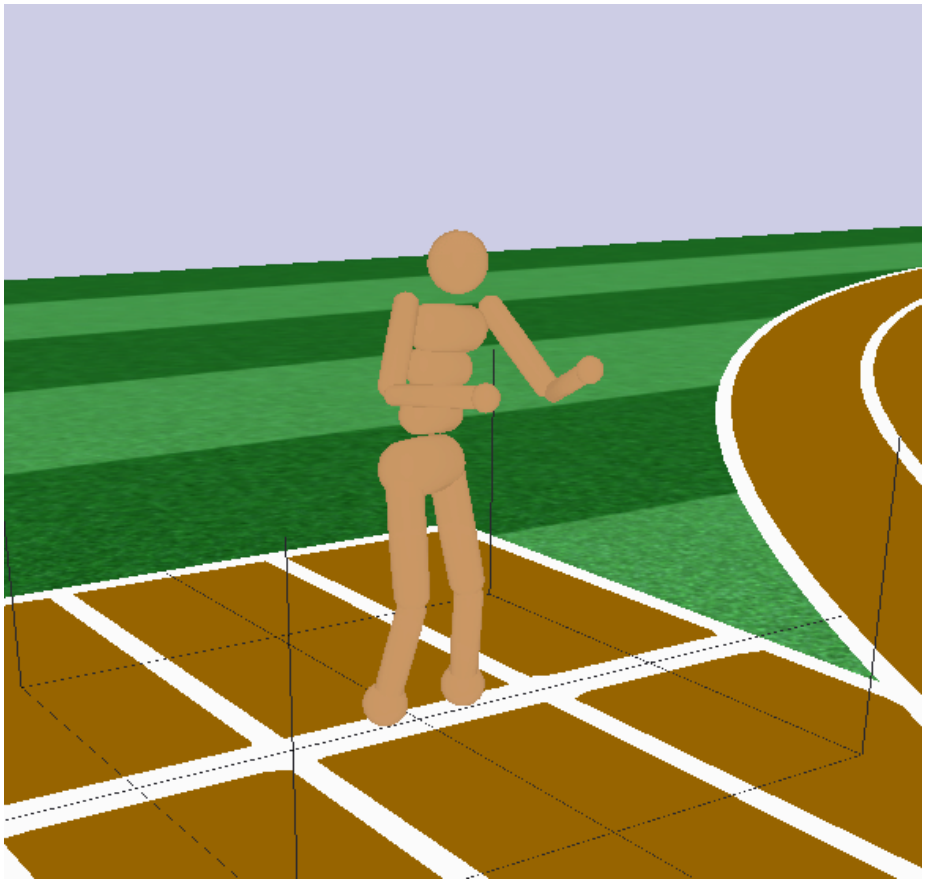
Capstone Report

Arise and walk, by Deep Deterministic Policy Gradient algorithm

Guitard Alan

June 15, 2018



1 Definition of the problem

1.1 Project Overview

With this project, I want to tackle an important task of robotics: the ability of a humanoid to walk. This is a really interesting problem since we, humans, learn to walk without any feedback from another agent, just by a training process. The parents doesn't teach anything to the kids, the kids just learns. I think this is very interesting to build such model of artificial intelligence in order to to understand our brain a little bit more.

The anthropologic reason is not the only reason that task is important, it is also important to build autonomous walking robots able to progress on every fields. There are several areas of actions: In war, such a robot will be able to reach wounded soldier to give medical treatment or bring the soldier back to a safe place. It will avoid a medic human to risk his life in order to do that. In hospital, we will gain time by giving task to the walking robot so doctors and nurses will have more time to give to the patient. In every day life, we will be able to get near from the future described by movies like "I-Robot". Even if it questions the place of the robot in human society, I think making this kind of robot is still a goal to reach in order to send the robot in places human cannot or with difficulty go, like to another planet.

I am talking about walking but a neural network able to make a robot walk sure will be able to tackle another difficult task. The difficulty in those task is their continuous observation spaces. Indeed, in order to make a robot walk, we have two possibilities. The first one is to learn over pixels of the environment. I didn't choose this one because I don't think a human learn to walk from its sight but rather from its body. That why I chose to watch the position of its joints. With pixels, we will able to train the humanoid to watch its steps but only after it is able to walk.

1.2 Problem Statement

The goal of this project, precisely, is to train a 3D avatar to walk forward on a plane fields. I will use OpenAI Gym [1] as an interface to the environment because it gives a simple and general way to use all kind of environment. For 3-dimensional avatar, it provides a binding for Mujoco library but since this library is paid, I will use a free similar one called Roboschool[1] which proposes few environments among the one I will use, RoboschoolHumanoid-v1. I am planning to use Reinforcement Learning to tackle this task with the Deep Deterministic Policy Gradient algorithm (Lillicrap et al.), an actor-critic algorithm. It is "Deep" because the actor and the critic is designed with deep neural networks. This algorithm is mostly used when the environment is a continuous space and since this environment never changes in our case (it is a plane field without changes), we can use a deterministic policy.

[1] https://github.com/openai/roboschool

Actor Critic algorithm In Reinforcement Learning, many algorithms uses an action-value function. It is a function which returns the value of an action a from a state s following a policy μ. It is defined like this:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))] \tag{1}$$

Besides that, the policy is the map of probabilities for an agent to go to state s_{t+1} from state s by the action a:

$$\mu(s, a) = P(a_t = a | s_t = s) \tag{2}$$

The idea of actor-critic algorithms is to use the action-value function as a critic which will give the Q value of the action taken by the agent at each time step. The higher the Q is, the more the reward the agent will get by taking that action in that state. The agent, here called the actor, will follow a policy μ in order to chose the action and that policy will be updated by the output of the critic. In Layman's terms, the actor is a child playing in the sandbox and the critic is the parent watching him. When the child make an action that may lead to a bad state, the parent gives him a bad reward in order to change his behaviour (policy).

1.3 Metrics

I will need two set of metrics because I have two kind of session for the body: it can walk and fail to stand or it can walk without falling. When the avatar will fall, I will restart the session, because to teach it to stand up is another kind of problem.

At the beggining of the training, the body will fall and fall again very quickly. So my metrics during that period will be the amount of time step it took before failing, the distance of the gravity center from the floor, the reward per action and the global average reward. Since the actor and critic are neural networks, I will also plot the loss of the critic network and their weights and biases, to check if it is changing over times. With these informations, I will be able to tell if my models is training well or if I have to adjust my parameters. When the body will start to have less fall, some of these metrics will not be informative anymore. I have to find metrics to evaluate the gait of the walk. For that, I will plot the angle of the current body position from the start position in respect to the axe the avatar will try to follow.

2 Analysis

2.1 Data Exploration: RoboschoolHumanoid-v1

2.1.1 Observation space

Definition An observation space, or state space, is the shape and the possible values an environment state could have. If the state is not continuous, we can calculate the state space by counting all possible values. For example, in a tic-tac-toe game, every square have 3 states so the state space is $3^9 = 19,683$. Here, our state is continuous, meaning that our possible values stands in a range. We can then just describe values and specify the range.

Our environment has an observation space of 44 float values in the range [-5, 5] which is a concatenation a three vectors described as follows:

- **more**: It is a vector of 8 values defined as follows:
  - The distance between the last position of the body and the current one.
  - The sinus of the angle to the target.
  - The cosinus of the angle to the target.
  - The three next values is the X, Y and Z values of the matrix multiplication between
    - $$\begin{pmatrix} \cos(-yaw) & -\sin(-yaw) & 0 \\ \sin(-yaw) & \cos(yaw) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
    - The speed vector of the body.
  - The roll value of the body
  - The pitch value of the body
- **j**: This is the current relative position of the joint described earlier and their current speed. The position is in the even position, and the speed in the odds (34 values).
- **feet_contact**: Boolean values, 0 or 1, for left and right feet, indicating if the respective feet is touching the ground or not.

2.1.2 Action space

Definition An action space is like observation space but for action an actor can take. In the tic-tac-toe example, the actor has 9 possible actions, which is playing in one of the square.

The action space is a vector of 17 float values in the range [-1, 1]. Each value corresponds to the joints of the avatar by this order in the XML file:

- abdomen_y
- abdomen_z
- abdomen_x
- right_hip_x
- right_hip_z
- right_hip_y
- right_knee
- left_hip_x
- left_hip_z
- left_hip_y
- left_knee
- right_shoulder1
- right_shoulder2
- right_elbow
- left_shoulder1
- left_shoulder2
- left_elbow

At each step, these values are applied to all the joints of the body by the code

```
for n,j in enumerate(self.ordered_joints):
    j.set_motor_torque( self.power*j.power_coef \
                        *float(np.clip(a[n], -1, +1)) )
```

in the `apply\_action` function in the class which extends the `gym.Env` class (`RoboschoolMujocoXmlEnv`) to set the torque value into the respective motor.

2.1.3 Reward

Definition A reward is a value given the information if the action was good or not given the state. The definition of the reward function is a critical aspect of reinforcement learning since it is the one who gives the most valuable information during the training.

The reward is a sum of 5 computed values:

- **alive**: -1 or +1 wether is on the ground or not
- **progress**: potential minus the old potential. The potential is defined by the speed multiplied by the distance to target point, to the negative.
- **electricity_cost**: The amount of energy needed for the last action
- **joints_at_limit_cost**: The amount of collision between joints of body during the last action
- **feet_collsion_cost**: The amount of feet collision taken during the last action

2.2 Exploratory Visualization

To visualize the exploration, we can simply use the render function of the gym environment which uses `Roboschool` library to create the 3D plane and humanoid such as show in the figure 1.

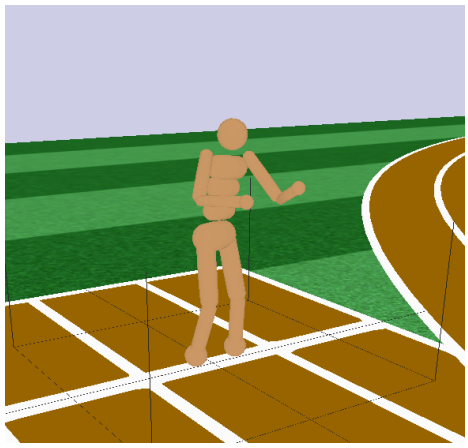


Figure 1: Roboschool environment

2.3 Algorithm and Techniques

Deep Deterministic Policy Gradient In DDPG, instead of compute manually the Q value and the μ, we use neural networks, one for the actor and one for the critic. The figure 2 shows how such a model can train.

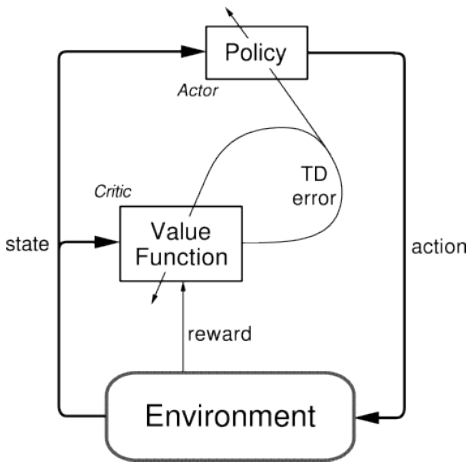


Figure 2: First, the environment gives the first state to the actor and it chose the action following the policy. The action is given back to the environment and a reward is computed, meaning how good the action was for that state, and sent to the critic network. The critic computes the Q value of that action by minimizing the loss and gives the error to the actor in order to let him train on it.

The Temporal Difference learning (TD Error in the figure) is a machine learning technique to, basically, let a network able to train with a little guess of the future. In our case of walking, the critic should not only see the next state but also estimate the chance the actor has to fall because of the action it took. To do that, we will use target actor and target critic network. It is, at the beggining, a copy of the

actor and the critic but we will use these networks to compute the target action for a state and Q_{t+1} and, at each time steps, we will update their weights $\theta^{Q'}$ with the actual network weights θ^{Q} by the following formula called "soft update":

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q} \tag{3}$$

Not using target networks will cause divergence and none of the model will learn a good policy or a good Q-value function.

We need to also design a Replay buffer to store transition (s_t, a, r, s_{t+1}) and train over minibatch of random transition. In that way, we break temporal correlations between transition and minimize variance between our possible bad predictions.

A last thing we need to introduce is exploration noise. We want to add some noise to the action the actor take during training in order to let it explore its world with some randomness. A good way to compute noise is the OrnsteinâĂŞUhlenbeck process, which is also known as Brownian motion.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q)^2)$
    Update the actor policy using the sampled gradient:

$$\nabla_{\theta^\mu}\mu|_{s_i} \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 3: Algorithm from Lillicrap et al. paper

The figure 3 describes the algorithm in a pseudo-code. The reader can see that the actor doesn't learn the classical way, i.e. by minimizing its loss. Instead of that, its weights is updated with the gradient of the critic network after it trained.

2.4 Benchmark

The most basic benchmark we have to test is the random benchmark. What will happen in a worse case scenario ?

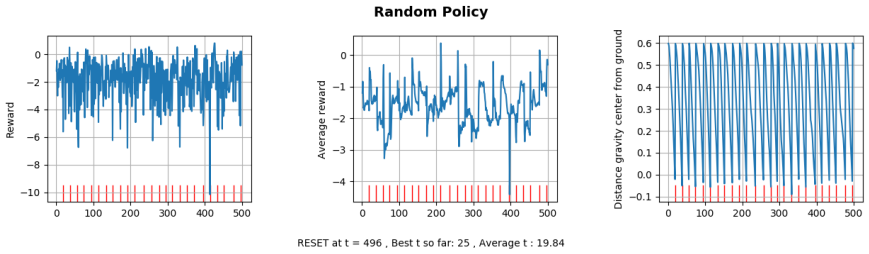
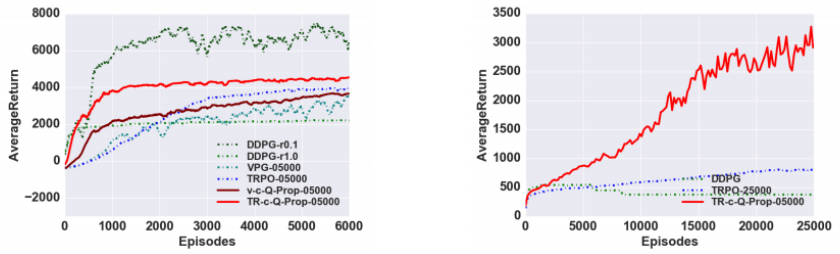


Figure 4: Random Policy

The figure 4 shows that the agent fall and fall again, without any chance to stand. The relevant information here is the average reward. The average reward of the random policy over 500 epochs is around 20 and the agent never lives more than 30 epochs.

Now we have a low benchmark, we have to dig into previous work of the community in order to find the state-of-the-art paper for that task. The work of Lillicrap et al. in their paper called *Continuous control with deep reinforcement Learning* does not show results with a humanoid environment but they did reach a good policy for 2d walker. Another more recent studies[2] shows results of different algorithm including DDPG.



(a) Comparing algorithms on HalfCheetah-v1.

(b) Comparing algorithms on Humanoid-v1.

Figure 5: Average return over episodes in HalfCheetah-v1 and Humanoid-v1 during learning, comparing Q-Prop against other model-free algorithms. Q-Prop with vanilla policy gradient outperforms TRPO on HalfCheetah. Q-Prop significantly outperforms TRPO in convergence time on Humanoid.

The figure 5 emphases that maybe DDPG is not a good choice to such a complex environment like Humanoid because DDPG is really sensitive about hyperparameters changes. I still want to try to teach a robot to walk with that algorithm by increasing the depth of the networks for example and see what happened.

3 Methodology

3.1 Data Preprocessing

The Roboschool environment is made for educational and research purpose. In that matter, a lot of useful data preprocessing is already made. For the observation vector, after building it as explained in section 2.1.1, the value is clipped between -5 and 5, meaning that if the environment provides a value outside of these bounds, the value is set to the nearest bounds. This is the same for action space but the value is clipped between -1 and 1.

3.2 Implementation

3.2.1 Project Structure

I present you below the structure of my project:

```
/
├── params.json
├── main_walk.py
└── walk
    ├── agents
    │   ├── abstract_env.py
    │   ├── ac_policy.py
    │   └── random_policy.py
    ├── models
    │   └── actor_critic.py
    └── utils
        ├── board.py
        ├── matplotboard.py
        ├── memory.py
        ├── noise.py
        ├── params.py
        └── tensorboard.py
```

In the `agents` folder, the `abstract_env.py` file is meant to be the super class of every policy I could implement, managing operation that every environment has to manage like instancing the plotting library or resetting the environment and its variables. In the `models` folder, we have our actor and critic networks definitions. In the `utils` folder, we have all classes used by our algorithm to make it work properly like visualization class, data structure and noise definition. The last thing I need to precise is that a lot of parameters in my program is written in `params.json` and read at runtime. By this way, I can search for the good set of parameters without entering into the code. I will explain in section 3.2.7 this file precisely after my implementation description.

3.2.2 Replay buffer

To implement that task, I started by writing the replay buffer. I created a class `Memory` which holds a queue as a member in order to store tuple of (s_t, a, r, s_{t+1}). I defined the first method as follow:

```
def remember(self, state, action, reward, next_state, done,
    state_range=None, action_range=None)
```

The `done` value is a boolean telling if the state is terminal or not. The state and the action range is a value to normalize the respective vector between 0 and 1. I think by this way neural networks will have better performance because input is less sparsed. If not `None`, these values must be equal to the range between low and high value (2 for action and 10 for state in our case).
The second needed method to define is the one to get samples from the queue randomly:

```
def samples(self, batch_size)
```

We need to get random sample, not last one or older one, but random, in order to break their correlations which would lead to decrease the performance of the model.

3.2.3 Neural networks: Actor and Critic

Keras vs Tensorflow In the beginning, to design my networks, I used Keras but this library is high-level API over Theano, Tensorflow and CNTK. It means that when we use it, we lose some control over the low-level API in order to be more readable and easy to implement. Since I will need to compute gradients and applying it to another networks, I understand that it will be more easy to do that in Tensorflow and it will be a good occasion to start to learn it.

The Actor The actor model is composed by fully-connected layers followed by Rectified Linear Unit activation function and then by batch normalization, because it gives better training speed and efficiency. The input is a shape of 44 float values for the state and the output is a shape of 17 float values activated by hyperbolic tangent because the range of the action values is equal to defintion range of the hyperbolic tangent, [-1, 1].

The Critic The critic model is composed by two inputs, one for the state and one for the action. The state input is followed by several dense layers activated by ReLU and batch normalization. The action input is followed by one layer with the same number of node than the last layer of the state network in order to be able to merge them. That layer is also followed by activation ReLU activation and batch normalization. These two layers is then added and followed by the output layer, a single-node one representing the Q-value of the state-action pair.

Target networks In our training process, when we have to do inferences from next actions or next states, we have to use actor target network and critic target

network. These identical network has been recently shown to increase stability during the training and decrease variance of outputs.

3.2.4 Training

The Actor TODO

The Critic I defined training operations for the critic with a mean squared error loss function and the ADAM optimizer, with learning rate defined in parameters and minimizing the loss.

Process Here comes the salt and pepper of Deep Deterministic Policy Gradients. I need here to train the critic network first, computing the gradients and apply it to the actor network. Here the complete code:

```
# Get samples of memory
states, actions, rewards, next_states, dones = \
    self.memory.samples(self.params.batch_size)

with tf.variable_scope("train_critic"):
    # Predicted actions
    next_actions = self.tf_session.run(
        self.target_actor_model.output,
        feed_dict={
            self.target_actor_model.input_ph: next_states
        })

    if self.params.action_range:
        next_actions = (next_actions +
                        self.params.action_range/2
        ) / self.params.action_range

    # Compute the Q+1 value with next s+1 and a+1
    Q_next = self.tf_session.run(
          self.target_critic_model.Q,
          feed_dict={
              self.target_critic_model.input_state_ph: next_states,
              self.target_critic_model.input_action_ph:
    next_actions
          })

    # gamma is the discounted factor
    Q_next = self.params.gamma * Q_next * (1 - dones)
    Q_next = np.add(Q_next, rewards)

    # Train the critic network and get gradients
    feed_critic = {
        self.critic_model.input_state_ph: states,
        self.critic_model.input_action_ph: actions,
        self.critic_model.true_target_ph: Q_next
    }
    self.critic_loss, _, critic_action_gradient = \
        self.tf_session.run(
            [self.critic_model.loss, self.critic_model.opt,
```

```
            self.critic_model.action_gradients],
        feed_dict=feed_critic)

    with tf.variable_scope("train_actor"):
        # Train the actor network with the critic gradients
        feed_actor = {
            self.actor_model.input_ph: states,
            self.actor_model.action_gradients: \
                critic_action_gradient[0]
        }
        self.actor_loss, _ = self.tf_session.run(
            [self.actor_model.loss,
             self.actor_model.opt],
             feed_dict=feed_actor)

    with tf.variable_scope("soft_update"):
        # Update target network
        self._update_target_network()
```

Here is the step by step explanation:

- We get temporally not-correlated batch from memory.
- We compute next actions with next states.
- We compute the expected reward Q_{next} with next states and next actions.
- We ponderate (or ignoring if the state is final) by the gamma parameters, i.e the discounted factor.
- We use it as a label to train the critic network and get gradients.
- We can then apply these gradients the actor to train it.
- Finally, we update target networks with soft update function.

```
self.update_critic_target =
   [self.ct_params[i].assign(
                tf.multiply(self.c_params[i],
                            self.params.tau) +
                tf.multiply(self.ct_params[i],
                            1. - self.params.tau))
        for i in range(len(self.ct_params))]

self.update_actor_target =
   [self.at_params[i].assign(
                tf.multiply(self.a_params[i],
                            self.params.tau) +
                tf.multiply(self.at_params[i],
                            1. - self.params.tau))
        for i in range(len(self.at_params))]

```

3.2.5 Noise

In my first implementation, I used a ϵ-greedy algorithm to explore the environment. It was a decaying asymptotic-to-0 function used to generate less and less random actions over time. But then I discovered noise function to reduce variance and increase stability over taken actions, like Ornstein-Uhlenbeck noise. It computes a vector of float equal to the shape passed in parameters and we can add it to the action given by the actor.

3.2.6 The Agent

Here is the main loop of the program:

```
for j in range(self.params.epochs):
    action = self.act(state)
    if self.params.noisy and j < self.params.noise_threshold:
        action += self.noise()

    new_state, reward, done, info = self.env.step(action)

    # Put the current environment in the memory
    # State interval is [-5;5] and action range is [-1;1]
    self.memory.remember(state, action, \
                         reward * self.params.reward_multiply, \
                         new_state, done, \
                         state_range=self.params.state_range, \
                         action_range=self.params.action_range)

    # Train the network
    self.train()

    # Reset the environment if done
    self.reset(done)

    # Render the environment
    self.render()

    # Plot needed values
    self.plotting(state=state, reward=reward, \
                  c_loss=self.critic_loss, \
                  a_loss=self.actor_loss)

    # Change current state
    state = new_state
```

3.2.7 Hyper Parameters

I present in this section every arguments of my programs. It was really usefull to quickly test many hyper parameters in order to see how does it affects the training.

- **reset**: `true` to reset the environment at final states, `false` otherwise.
- **render**: `true` to render the environment, `false` otherwise. Roboschool has a bug which lead to give vector of `inf` after the second reset when render, so

when I train the networkds, I don't render it.

- **plot**: `tensorflow` to visualize plot with tensorboard, `matplotlib` to render plot with matplotlib.
- **train**: `true` to train network, `false` otherwise.
- **noisy**: `true` to add noise to action output, `false` otherwise.
- **load_weights**: `true` to load previously saved weight from previous training session, `false` otherwise.
- **batch_size**: `integer`, number of samples to get from memory.
- **epochs**: `integer`, number of epochs to act.
- **noise_threshold**; `integer`, when epochs is hight than this number, the program stops adding noise to the action output.
- **actor_learning_rate**: `float`, learning rate for the actor network.
- **critic_learning_rate**: `float`, learning rate for the critic network.
- **actor_batch_norm**: `true` to add batch normamization after dense layers of the actor network, `false` otherwise.
- **critic_batch_norm**: `true` to add batch normamization after dense layers of the critic network, `false` otherwise.
- **tau**: `float`, used in the soft target update function.
- **gamma**: `float`, discounted factor.
- **actor_layers**: `list of integer`, define the structure of the actor network, the depth and the number of unit in each layer.
- **critic_layers**: `list of integer`, define the structure of the critic network, the depth and the number of unit in each layer.
- **action_range**: `integer`, size of the action range to normalize it between 0 and 1.
- **state_range**: `integer`, size of the state range to normalize it between 0 and 1.
- **reward_multiply**: `integer`, multiply every reward with to reward more when good action is taken and less when a bad one is taken.
- **dropout**: `float`, percent of dropout to use after dense layers. 0 to not add dropout.

3.2.8 Visualization

3.3 Refinement

4 Results

4.1 Model Evaluation and Validation

4.2 Justification

5 Conclusion

5.1 Free-Form Visualization

5.2 Reflection

5.3 Improvement

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E. Turner, and Sergey Levine. Q-prop: Sample-efficient policy gradient with an off-policy critic. In *Proceedings International Conference on Learning Representations 2017*. OpenReviews.net, April 2017. URL `https://openreview.net/pdf?id=rkE3y85ee`.

[3] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL `http://dblp.uni-trier.de/db/journals/corr/corr1509.html#LillicrapHPHETS15`.