

[Return to Classroom](#)[DISCUSS ON STUDENT HUB](#)

Continuous Control

REVIEW

CODE REVIEW 8

HISTORY

Meets Specifications

Congratulations

Great submission!

All functions were implemented correctly and the final algorithm seems to work quite well.

Algorithm used here is the deep deterministic policy gradient algorithm (ddpg).

DDPG is a model-free policy based learning algorithm in which the agent will learn directly from the unprocessed observation spaces without knowing the domain dynamic information.

Helpfull links to understand ddpq better:

- Fun part
- Deep Technical
- Great explanation

Training Code

The repository includes functional, well-documented, and organized code for training the agent.

You implemented a deep deterministic policy gradient algorithm (ddpg), a very effective reinforcement learning algorithm to solve the Reacher environment.

Your code was functional, well-documented, and organized for training the agent.


About Ornstein-Uhlenbeck process:

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.1): # sigma = 0.2
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in range(len(x))])
        self.state = x + dx
        return self.state
```



Comments about the idea of improving agent performance by adding noise:

- This implementation of Ornstein-Uhlenbeck process (`random.random()`) can have significant bias towards positive values and eventually can result in poor training.
- So, the use of uniform distribution is biased and since it tends to accumulate at around 0.6.
- You could try the replacement with a generator that draws samples from a **standard Normal distribution** (`np.random.standard_normal()`), and thus generate a different noise sample for each agent as opposed to applying same noise to all agents.

```
dx = self.theta * (self.mu - x) + self.sigma * (np.random.standard_normal(size=x.shape))
```

The code is written in PyTorch and Python 3.

The code was written in PyTorch and Python 3.

PyTorch and TensorFlow are the most extensively used frameworks in deep learning. It would be good to get some insight by comparing them.

Suggested reading about the discussion of what machine learning framework should be used.

- [TensorFlow vs. Pytorch.](#)
- [Tensorflow or PyTorch : The force is strong with which one?](#)
- [What is the best programming language for Machine Learning?](#)
- [PyTorch vs TensorFlow—spotting the difference](#)

Suggested reading:

- [Google Python Style Guide](#)
- [Python Best Practices](#)
- [Clean Code Summary](#)

The submission includes the saved model weights of the successful agent.

You included the saved model weights of the successful agent.

Suggested reads for further learning::

- [Saving and Loading Models](#)
- [Best way to save a trained model in PyTorch?](#)

README

The GitHub submission includes a `README.md` file in the root of the repository.

Well done! A detailed README file has been provided and is present in the repository.

The README.md file is important for many reasons

- It is an industry standard practice and helps to make the repository look professional.
- It make easy for the general audience to reuse the code.
- A README file shows:
 - Why the project is useful,
 - What to do with the project, and
 - How to use it.

That's what a README is for.

I recommend you to check

- This awesome [template to make good README.md](#).
- This article [Make a README - Because no one can read your mind \(yet\)](#), is a crash course of markdown READMEs.

The README describes the the project environment details (i.e., the state and action spaces, and when the environment is considered solved).

Good work with environment details.

The README described all the project environment details.

Suggested description of the project environment details

- **Environment:** How is it like?
- **Agent and its actions:** When is it considered resolved?; What are the possible actions the agent can take?
- **State space:** Is it continuous or discrete?
- **Reward Function:** How is the agent rewarded?
- **Task:** What is its task?; Is the task episodic or not?

The README has instructions for installing dependencies or downloading needed files.

The README described all instructions for installing

dependencies or downloading needed files in Getting Started item.

The README must describe all instructions for

- Installing [dependencies](#)
- Downloading needed files in [Getting Started instructions item](#).

The README describes how to run the code in the repository, to train the agent. For additional resources on creating READMEs or using Markdown, see [here](#) and [here](#).

Nice job!

The README correctly describes how to run the code in the repository.

I recommend you to check the [Mastering Markdown](#), there are great tips about how to use Markdown

Report

The submission includes a file in the root of the GitHub repository (one of `Report.md`, `Report.ipynb`, or `Report.pdf`) that provides a description of the implementation.

All required files were included

You included the report of the project with the complete description of the implementation. All the sections are detailed.

Well done!

The report content:

- Description of the learning algorithm
- The hyperparameters used.
- The model architecture of Actor and Critic.
- A plot showing the increase in average reward as the agent learns

- A plot showing the increase in average reward as the agent learns.
- The weakness found in the algorithm and how to improve it.

The report clearly describes the learning algorithm, along with the chosen hyperparameters. It also describes the model architectures for any neural networks.

Nice job! Your agent trains in a reasonable number of episodes.

The report described the learning algorithm, the chosen hyperparameters, the model architectures.

Learning algorithm:

The DDPG (Deep Deterministic Reinforcement Learning) is implemented, which consists of four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network. The Q network and policy network is alike the Advantage Actor-Critic, but the Actor directly maps states to actions (the output of the network directly the output) instead of outputting the probability distribution across a discrete action space. The target networks are time-delayed copies of their original networks that iteratively updated through training in each episode. Using the target value networks stabilizes the learning process.

I used Adam for learning the neural network parameters with a learning rate of $2e-4$ and $2e-3$ for the actor and critic respectively. The L2 weight decay is 0 and the discount factor $\gamma = 0.99$. For the soft target updates $\tau = 0.001$. The neural networks used the rectified non-linearity for two hidden layers. Also, a batch normalization is applied to the first hidden layer. The final output layer of the actor was a tanh layer, to bound the actions. Both the actor and critic networks had 2 hidden layers both with 128 units respectively. The final layer weights and biases of both the actor and critic were initialized from a uniform distribution $[-3 \times 10^{-3}; 3 \times 10^{-3}]$. This was to ensure the initial outputs for the policy and value estimates were near zero.

Hyperparameters:

```

BUFFER_SIZE = int(1e5)           # replay buffer size
BATCH_SIZE = 128 #64 #128        # minibatch size
GAMMA = 0.99                     # discount factor
TAU = 1e-3                      # for soft update of target parameters
LR_ACTOR = 2e-4 #1e-4            # learning rate of the actor
LR_CRITIC = 2e-4 # 1e-3         # learning rate of the critic
WEIGHT_DECAY = 0                 # L2 weight decay
LEARN_EVERY_N = 20              # learn every N intervals
ACTOR_FC1_UNITS = 128 #400 # 100 # 400
ACTOR_FC2_UNITS = 128 #300 # 100 #300
CRITIC_FC1_UNITS = 128 #400 # 100 #400
CRITIC_FC2_UNITS = 128 #300 # 100 #300

```

Model architecture:

```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=ACTOR_FC1_UNITS, fc2_units=ACTOR_FC2_UNITS):
        """Initialize parameters and build model.

        Params
        =====
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.bn1 = nn.BatchNorm1d(fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.bn1(self.fc1(state)))
        x = F.relu(self.fc2(x))
        return F.tanh(self.fc3(x))
```

```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fcs1_units=CRITIC_FC1_UNITS, fc2_units=CRITIC_FC2_UNITS):
        """Initialize parameters and build model.

        Params
        =====
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.bn1 = nn.BatchNorm1d(fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        xs = F.relu(self.bn1(self.fcs1(state)))
```

```
x = torch.cat((xs, action), dim=1)
x = F.relu(self.fc2(x))
return self.fc3(x)
```

A plot of rewards per episode is included to illustrate that either:

- [version 1] the agent receives an average reward (over 100 episodes) of at least +30, or
- [version 2] the agent is able to receive an average reward (over 100 episodes, and over all 20 agents) of at least +30.

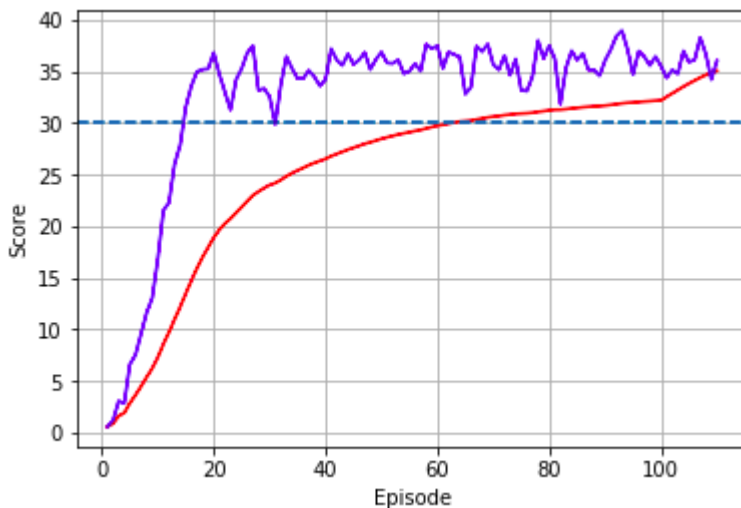
The submission reports the number of episodes needed to solve the environment.

The report informed the number of episodes needed to solve the environment:

Results

The graph below shows the training process, showing the score of each episode (blue) and the moving average (red). It shows that the moving average score is 30 at the episode 63. The complete set of results and steps can be found in [this notebook](#).

The report also included a plot of rewards per episode:



The implemented code for DDPG is correct. I didn't find any problem.

The variation in training is most likely due to the randomness in the environment.

The submission has concrete future ideas for improving the agent's performance.

Nice work suggesting the ideas for future improvement

Future Improvements

- Maybe try the STOA D4PG algorithm [Distributed Distributional Deterministic Policy Gradients \(D4PG\)](#).

You could spend some more time working on hyper-parameters tuning and optimizing the neural network structure.

- You can almost always obtain better parameter tuning.
- When you do, you can even reduce the number of layers needed.
 - Simpler architectures need less testing time.
- Therefore, they are faster, which is better for end-users.

You have used Batch normalization:

- Batch normalizer enhances training.
- However, it takes more computations and therefore more training time.

Here are some links that will help you:

- [Asynchronous Actor-Critic Agents \(A3C\)](#).
- [Trust Region Policy Optimization \(TRPO\) and Proximal Policy Optimization \(PPO\) and PPO implementation](#).
- [D4PG](#).

I hope this helps.

Keep it up.

Stay safe

 [DOWNLOAD PROJECT](#)

RETURN TO PATH

Rate this review

START