

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

Continuous Control

REVIEW

CODE REVIEW 6

HISTORY

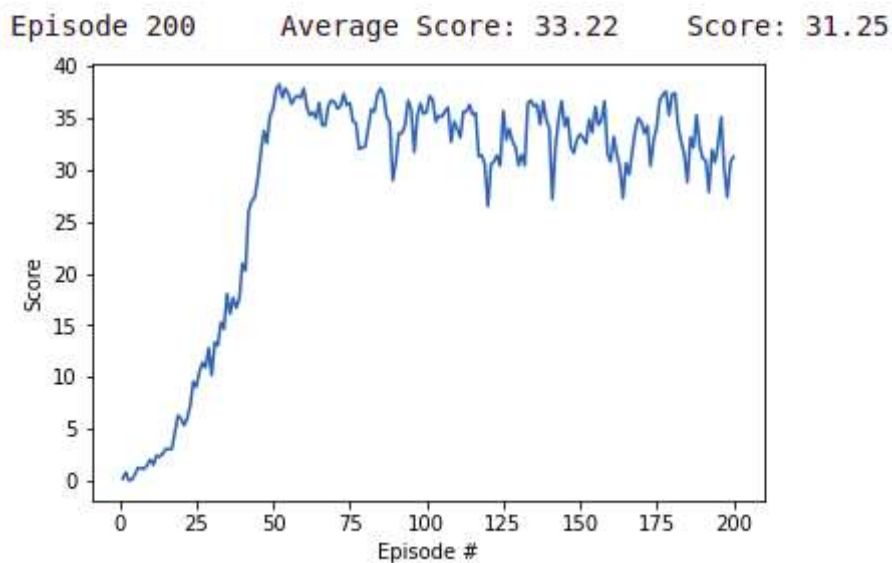
Meets Specifications

Congratulations

Great submission!

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

```
scores = ddpq()  
  
fig = plt.figure()  
ax = fig.add_subplot(111)  
plt.plot(np.arange(1, len(scores)+1), scores)  
plt.ylabel('Score')  
plt.xlabel('Episode #')  
plt.show()
```



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.

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

Suggested reading:

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

The code is written in PyTorch and Python 3.

The code was written in PyTorch and Python 3.

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?](#)

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.

The submission included a README.md file.

I recommend you to check this awesome [template to make good README.md](#).

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 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!

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 description of the implementation.

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.
- 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!

You described the learning algorithm, the chosen hyperparameters, the model architectures.

Hyperparameters:

```
BUFFER_SIZE = int(1e5) # replay buffer size
BATCH_SIZE = 1000      # minibatch size
GAMMA = 0.99           # discount factor
TAU = 1e-3             # for soft update of target parameters
LR_ACTOR = 1e-4        # learning rate of the actor
LR_CRITIC = 1e-3       # learning rate of the critic
WEIGHT_DECAY = 0       # L2 weight decay
```

Model architecture:

```

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

    def __init__(self, state_size, action_size, seed, fc1_units=264, fc2_units=132):
        """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.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

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

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

    def __init__(self, state_size, action_size, seed, fcs1_units=264, fc2_units=132):
        """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.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

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

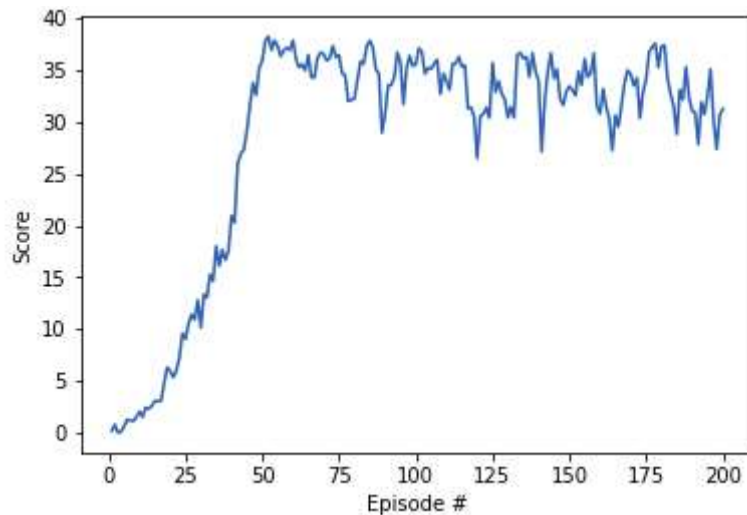
- *[version 1]* the agent receives an average reward (over 100 episodes) of at least +30, or
- *[version 2]* the agent is able to receive an average reward (over 100 episodes, and over all 20 agents) of at least +30.

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

You informed the number of episodes needed to solve the environment:

Episode 200 Average Score: 33.22 Score: 31.25

You also included a plot of rewards per episode:



With Reacher you can do parallel training with up to 20 agents in parallel.

- [First](#)
- [Second](#)

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

Nice work suggesting the ideas for future improvement

```
I have problem with noise and find a fix in class chat where i used np.random.standard_normal(size) instead  
np.array([random.random() for i in range(len(x))])
```




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

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""
        self.size = size
        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.random.standard_normal(self.size)
        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 have used the **standard normal distribution**.
- It was a reasonable choice
- A generator that draws samples from a **standard Normal distribution** (**np.random.standard_normal()**) will generate a different noise sample for each agent.

 [DOWNLOAD PROJECT](#)

6 [CODE REVIEW COMMENTS](#)



[RETURN TO PATH](#)

