UDACITY

‹ Return to Classroom

DISCUSS ON STUDENT HUB

Continuous Control

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Dear Udacian,

Great job getting acquainted with the Deep Deterministic Policy Gradients algorithm and successfully implementing it to solve the Reacher environment. The implementation is pretty good and the environment is solved in just 144 episodes. The architectures used for the actor and critic network are decent in size with two hidden layers each. Good work using `relu` activations and `batch normalization`. The report is extremely informative and covers all the important aspects of the implementation. 😄

I would suggest you to go through Deep Reinforcement Learning for Self Driving Car by MIT. You'd get to know more about reinforcement learning algorithms in broader and real-world perspective and, more importantly, how to apply these techniques to real-world problems.

All the best for future endeavors. ✨

Training Code

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

Awesome

- Good work implementing DDPG algorithm to solve robotic-arms Reacher environment.
- Implementation of the Actor and Critic networks is correct.
- Good work using the target networks for Actor and Critic networks. The original DDPG paper suggests it as well.
- Good work using soft updates for the target network.
- Good choice to use tau to perform soft update.
- Correct usage of replay memory to store and recall experience tuples.
- The implementation is easy to debug and easily extensible, good work keeping it highly modular.

The code is written in PyTorch and Python 3.

Awesome

The code is written in PyTorch and Python 3.

Lately, PyTorch and TensorFlow happen to be most extensively used frameworks in deep learning. It would be good to get some insight by comparing them, please see the following resources:

- Sebastian Thrun on TensorFlow
- PyTorch vs TensorFlow — spotting the difference
- Tensorflow or PyTorch : The Force is Strong with which One?

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

Awesome

- Saved model weights of the successful agent have been submitted.
- `checkpoint_actor_second_env.pth` and `checkpoint_critic_second_env.pth` files are present in the submission.

README

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

Awesome

- Great work documenting the project details and submitting the README file.

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

Awesome

- Great work providing the details of the project environment in the `Task Goals and Details` section of the README.
- The section describes the project environment by specifying the state space, action space, and the desired results.

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

Awesome

Great work providing all the necessary instructions in the `Environment Set Up` section

- `Step 1 Clone the DRLND Repository` subsection to install the dependencies.
- `Step 2: Download the Unity Environment` subsection to download the environment.

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.

Awesome

- Great work providing necessary instructions to run the code in the `Instructions to Run the Code` section.
- All the cells in `Reacher.ipynb` file should be executed to train the agent.

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.

Awesome

- Report for the project with all the details of the implementation has been provided in the submission.

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

Awesome

Great work providing the details of the implemented agent. Details of the learning algorithm used, hyper-parameters, and architectural information of the deep learning model have been provided.

- Good decision to choose DDPG algorithm for the continuous action space problem.

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 policy gradient:

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

    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

- Good work including model architecture in the report.

The **Actor Neural Networks** use the following architecture.

```
Input nodes (33)
  -> Fully Connected Layer (128 nodes, Relu activation)
    -> Batch Normlization
      -> Fully Connected Layer (128 nodes, Relu activation)
         -> Ouput nodes (4 nodes, tanh activation)
```

The **Critic Neural Networks** use the following architecture :

```
Input nodes (33)
  -> Fully Connected Layer (128 nodes, Relu activation)
    -> Batch Normlization
      -> Include Actions at the second fully connected layer
        -> Fully Connected Layer (128+4 nodes, Relu activation)
          -> Ouput node (1 node, no activation)
```

- Good decision choosing to use batch normalization.
- Hyperparameters you have used seem to be good.

  - `BATCH_SIZE = 128`: neural network mini-batch size
  - `GAMMA = 0.99`: the discount factor used in the discounted sum of rewards
  - `TAU = 1e-3`: the τ parameter used to soft update of target parameters
  - `LR_ACTOR = 1e-4`: the actor neural network learning rate use in gradient descent
  - `LR_CRITIC = 1e-4`: the critic neural network learning rate use in gradient descent
  - `WEIGHT_DECAY = 0.0`: L2 weight decay
  - `BUFFER_SIZE = 1e6`: replay buffer size (how much we store into the replay buffer)

Suggestions

To experiment more with the architecture and hyperparameters, you can check the following resources:

- Deep Deterministic Policy Gradients in TensorFlow
- Continuous control with Deep Reinforcement Learning

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.**

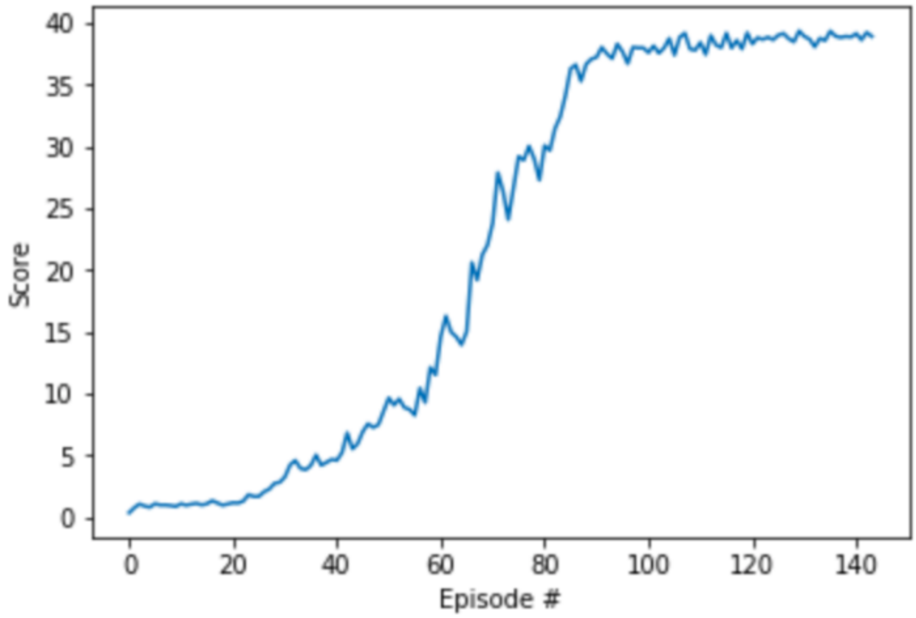
The submission reports the number of episodes needed to solve the environment.

Awesome

- Discussion for the rewards is provided in the report.
- The rewards plot seems to be good and average score of +30.01 is achieved in 144 episodes.

```
Episode 136     Overall Average Score: 27.29    Mean: 39.36     Total Time Per Episode: 14.97 seconds
Episode 137     Overall Average Score: 27.63    Mean: 38.95     Total Time Per Episode: 14.93 seconds
Episode 138     Overall Average Score: 27.98    Mean: 38.84     Total Time Per Episode: 14.91 seconds
Episode 139     Overall Average Score: 28.33    Mean: 38.94     Total Time Per Episode: 14.87 seconds
Episode 140     Overall Average Score: 28.67    Mean: 38.88     Total Time Per Episode: 14.92 seconds
Saving Weights in Episode 140   Overall Average Score: 28.67    Mean: 38.88     Mean Time Per 20 Episodes: 14.88 seconds
Episode 141     Overall Average Score: 29.01    Mean: 39.17     Total Time Per Episode: 14.96 seconds
Episode 142     Overall Average Score: 29.35    Mean: 38.65     Total Time Per Episode: 14.83 seconds
Episode 143     Overall Average Score: 29.67    Mean: 39.25     Total Time Per Episode: 15.03 seconds
Episode 144     Overall Average Score: 30.01    Mean: 38.93     Total Time Per Episode: 14.92 seconds
Environment solved in 144 episodes with an Average Score of 30.01
```

Figure 3: Learning evolution



Reinforcement learning algorithms are really hard to make work.
But it is substantial to put efforts in reinforcement learning as it is close to Artificial General Intelligence.
This article is a must read: Deep Reinforcement Learning Doesn't Work Yet.

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

Awesome

- Thanks for providing the following concrete ideas for improvement.

  - as suggested in the benchmark subsection of the continuous control project section, something to try latter is to implement the TRPO, TNPG and also PPO algorithms, they may yield better results
  - In the actor-critic subsection some other algorithms are shown: A2C,A3C,GAE.
  - i would really like to try D4PG!
  - also besides trying very hard I found that solving the environment with just one actor was harder than with two. I'd like to try implementing the start_learn function in the single environment to see it works as well as in the second one.
  - also the Ornstein-Uhlenbeck process to introduce noise in the parameters was an intersting idead. I'd like to explore more processes to see how they can change the agent learning, when everything else is "fixed".

- Please check the following resources also:
  - Prioritized Experience Replay

- Distributed Prioritized Experience Replay
- Reinforcement Learning with Prediction-Based Rewards
- Proximal Policy Optimization
- OpenAI Five
- Curiosity-driven Exploration by Self-supervised Prediction

DOWNLOAD PROJECT

RETURN TO PATH