Double-jointed Arm Reacher Project Solution

Udacity Deep Reinforcement Learning Project Number 02

Leonardo Uchoa Pedreira

7/28/2021

Contents

Description

This document is a report describing the learning algorithm and details of implementation, along with ideas for future work.

Algorithm: DDPG

Deep Deterministic Policy Gradients, or DDPG for short, is an Actor-Critic Reinforcement Learning method created to enable agents to better learn optimal policies to behave in continuous action spaces. DDPG borrows ideas from both `DQN` and `DPG` (DPG stands for Deterministic Policy Gradients).

The `DQN` is a method where a neural network is used is to implement the `Q-Learning` algorithm, which attempts to estimate action-value pairs in order to maximize the expected total reward and, therefore, to obtain the optimal policy for the given task. It belongs to class of value-based methods, whose goal is to solve the bellman equation. Solving the bellman equation gives us the optimal policy, given that our environment meets certain criteria in our Markov Decision Process setting. For `Q-Learning` in particular the equation we're trying to solve is as follows

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

So in a given time step `t` we search for the action that maximizes the action-value pair $Q(s_{t+1}, a)$ in order to estimate the optimal future value. DQN uses neural networks to step up from the traditional tabular method approach to the function approximation approach, which means that instead of storing the q-values in a table, we're going to encode it in a **parametrized** function approximator (parametrization what turn the q-value from $q(s, a)$ to $q(s, a; \theta)$). This gives us a lot flexibility to solve many other problems, specially those whose state space is continuous.

The figure bellow is the `DQN` implementation proposed in the nature paper. In that implementation there are two important additional tricks. Those tricks are employed in order to solve some instability that the neural network suffers in training. The first is experience replay and the second is the fixed q-targets.

Experience replay is mainly just creating a buffer to store some events (the (S_t, A_t, R_t, S_{t+1}) action-pairs) and then latter using mini-batches of them to run gradient descent and learn the network weights.

Fixed q-targets is a technique used to avoid a difficulty found in Q-Learning, where we update a guess with a guess, which can potentially lead to harmful correlations. In Q-Learning after each pass our neural network tries to get as close as possible to the target q-values, but the problem is that because we update the weights after each forward pass, our q-value target to calculate the loss function is always changing and that makes learning a bit unstable. To solve this problem the idea is to store the weights in another neural network $\hat{q}$ (called target network), freeze $\hat{q}$ the weights and update them every once in a while. Now when performing the learning step, our q-value loss function is the deviation from the forward pass calculated q and the target (fixed) $\hat{q}$ value. That means that out "ground-truth" response variable is the fixed q-value from the target network and that makes learning more stable.

The Deterministic Policy Gradients, or `DPG`, is an actor-critic method that attempts to combine both value-based and policy-based methods to learn an optimal policy IN continuous action spaces. Following the original paper, Deterministic Policy Gradient Algorithms, on the `DPG`:

> The deterministic policy gradient has a particularly appealing form: it is the expected gradient of the action-value function. This simple form means that the deterministic policy gradient can be estimated much more efficiently than the usual stochastic policy gradient.

The `DDPG` algorithm then is a mix of both `DPG` and `DQN`, where we have two networks: the actor and critic. The actor is responsible for estimating the value (or action-value) function while the critic is responsible for evaluating the value function. So the intuition behind actor-critic methods goes like this:

1. the actor takes in the current state and takes an action, based on the current policy $\pi(a|s; \theta_\pi)$.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 1: DQN paper implementation

2. collect the experience tuple (s, a, r, s') to feed the critic, whose job is to estimate $V(s; \theta_v)$.
3. use the critic output to calculate the advantage function $A(s, a) = r + \gamma V(s'; \theta_v) - V(s; \theta_v)$ and use it as the new baseline for actor to move on.
4. Repeat steps 1 through 3 until a stopping rule is achieved (end learning).

But the `DDPG` algorithm is a bit different from classic actor-critic methods. The actor in `DDPG` is used to approximate $\max_a Q(s_{t+1}, a)$, an expression found the original `DQN` implementation and not as a learn baseline. So instead of outputting $\pi(a|s; \theta_\pi)$, the actor outputs $\mu(s; \theta_\mu)$, an approximation of $\max_a Q(s_{t+1}, a)$ (notice that this is follows an deterministic policy, not a stochastic policy). Then the critic takes the actor output and learns to evaluate it, by estimating $Q(s, \mu(s; \theta_\mu); \theta_Q)$.

Besides that, `DDPG` is very much like what was described in steps 1-4. So in the `DDPG` algorithm implementation of steps 1 through 4 the actor will have two neural networks (a target and a policy network, like in the `DQN`) and for the critic we also have two neural networks (a target and a policy network, also like in the `DQN`). Also present in the `DDPG` algorithm is the replay buffer, in order improve learning by collecting some chunks of episodes. Also present are the fixed q targets used to help stabilize learning, as described in the `DQN` implementation. `DDPG` pretty much resembles the `DQN` algorithm, but addapted to support continuous action spaces. The `DDPG` is depicted in Figure 02.

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

Figure 2: DDPG paper implementation

Coding the Algorithm

The code implementation of the algorithm used here is mostly inspired the by "Pendulum" project notebook provided by `Udacity`, in the "Actor-Critic Methods" section. Here is a brief description of the files:

- **Reacher.ipynb**: a jupyter notebook that serves as a wrapper of smaller functions. This is the main file, responsible for: 1) starting the environment; 2) loading the neural network architecture in `model_second_env.py`; 3) loading the `ddpg_agent` implementation, running it and saving model weights and score results.
- **model_second_env.py**: this is where the neural network architectures are stored, written in `pytorch`.
- **ddpg_agent_second_env.py**: this is code that implements the `DDPG` high-level ideas of fixed q-targets, experience replay and Ornstein-Uhlenbeck noise.
- **checkpoint_actor_second_env.pth**: this is the `DDPG` actor model weights used to solve the environment.
- **checkpoint_critic_second_env.pth**: this is the `DDPG` critic model weights used to solve the environment.
- **second_env_model_results.sav**: this a pickled object that contains the scores, average scores and time spent training.

Comments on `model_second_env.py`

The `model_second_env.py` file contains the `pytorch` model architecture used to solve the environment. The input size that the network expects is the 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm, while the output layer is a simple linear layer with output size of 4 corresponding to torque applicable to two joints.

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)
```

The `hyperparameters` used are:

- the number of units in both first and second fully connected layer are 128.
- the learning rate parameter used in the `adam` optimizer for both actor and critic are `LR_ACTOR = LR_CRITIC = 1e-4` (this actually belongs to the `ddpg_agent_second_env.py` file, but it's also part of the neural network architecture)

Comments on `ddpg_agent_second_env.py`

The `ddpg_agent_second_env.py` file contains:

- the `hyperparameters` used:
  - `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)
- the `Agent` class:
  - implements the `step`, `act`, `learn` and `soft_updates` methods. Also has the `start_learn` to implement the idea to learn just after some episodes in order to help stabilize training (as suggested in the benchmark section)
  - initializes the policy and target networks for fixed q-target strategy for both actor and critic networks.
  - initializes the `ReplayBuffer` class.
- the `OUNoise` class:
  - implements the Ornstein-Uhlenbeck process in order to help exploration
  - hyperparameters used are the default already present in the udacity pendulum implementation. Namely:
    * `mu = 0`
    * `theta = 0.15`
    * `sigma = 0.2`
  - changed only to adapt the code the multiple agents environment
- the `ReplayBuffer` class: an API to
  - initialize storage
  - implement the `add` (add new experiences) and `sample` (sample experiences to mini-batch GD learning) methods

Results

The DDPG implementation was able to successfully solve the task of achieving an average score of +30 over 100 episodes in 144 episodes. Its average score was 30.01, as shown in the Figure 03 bellow. It's important to note that we used a early stopping technique not to go into the 2000 episodes, if not needed. The plot of the scores is shown in 'Figure 04.

```
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

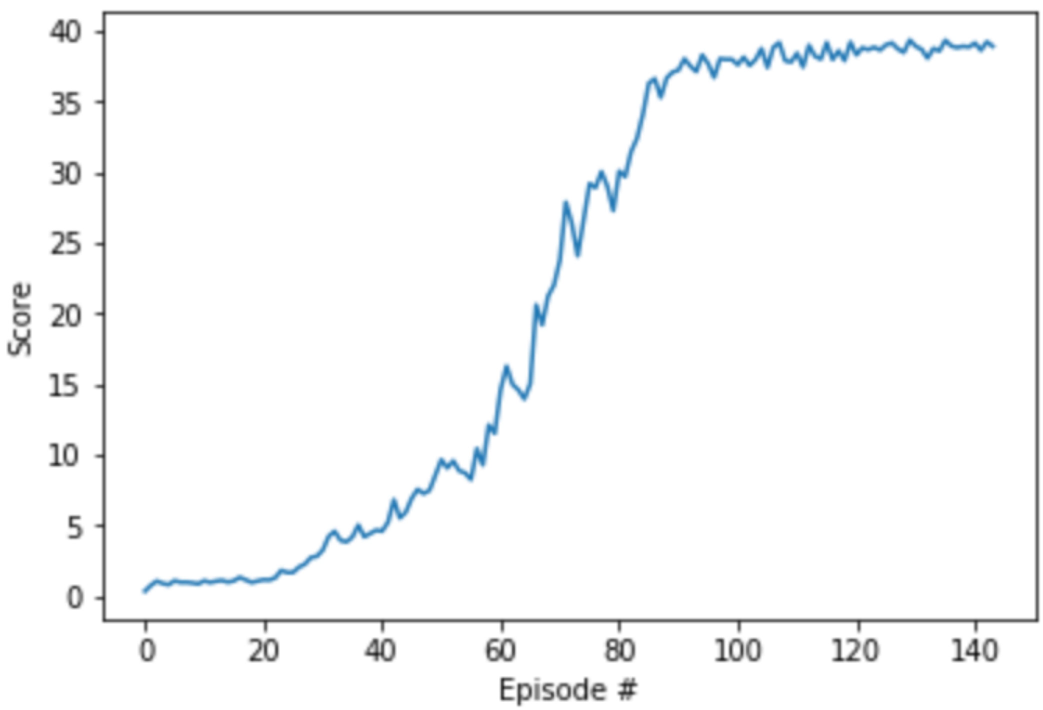

Figure 4: Learning evolution plot

Ideas to Explore Later

Some ideas that could lead to improvements in the learning process are trying to:

- 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".