Report on Continuous Control project

Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) algorithm is used to solve the Reacher Continuous Control environment.

Lillicrap et al. (2016) proposed a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional continuous action spaces. The algorithm combines the actor-critic approach with using the same key features applied in Deep Q Network (DQN):

1. The network is trained off-policy with samples from a replay buffer to minimize correlations between samples.
2. The network is trained with a target Q network to give consistent targets during temporal difference backups.

The same features are used in DDPG along with specific details for DDPG:

1. **Batch normalization:** This technique normalizes each dimension across the samples in a minibatch to have unit mean and variance. In addition, it maintains a running average of the mean and variance to use for normalization during testing (in our case, during exploration or evaluation). In deep networks, it is used to minimize covariance shift during training, by ensuring that each layer receives whitened input **[4]**. In this project, the batch normalization is used on the RELU activation output of the state input in both Actor and Critic networks.
2. **Adding noise to the Actor policy:** The performance of DDPG critically depends on a proper choice of exploration policy πe, which controls what data to add at each iteration. However, in high dimensional continuous action space, exploration is highly nontrivial. In the current practice of DDPG, the exploration policy πe is often constructed heuristically by adding certain type of noise to the actor policy to encourage stochastic exploration. A common practice is to add an uncorrelated Gaussian or a correlated Ornstein-Uhlenbeck (OU) process to the action selected by the deterministic actor policy **[5]**. If the noise generated at a given timestep is correlated to previous noise, it will tend to stay in the same direction for longer durations instead of immediately canceling itself out, which will consequently allow increasing velocity and unfreezing the position **[6]**. In this project, Ornstein-Uhlenbeck process is used with $\theta = 0.15$ and $\sigma = 0.2$.

The DDPG algorithm can be represented as follows:

1. The actor and critic use two separate neural networks.
2. The actor network will take the **state s** as an input and returns the **action a** as an output. The actor is used to approximate the optimal policy deterministically meaning that the output is the best action at any given state.

3. The critic network will take the **state s** and **action a** as an input and returns the Q value. The critic learns to evaluate the optimal action value function by using the actor's best action.
4. Define the target network for each of the Actor network and Critic network.
5. Update of Actors network weights with policy gradients and the Critics network weights with the gradients calculated from the TD error.
6. Add **an exploration noise N** to the action produced by the actor in order to select the correct action.
7. The agent selected an action in **state s** and the environment sends the agent the **next state s'**and the **reward r**.
8. This transition is saved to the experience replay buffer.
9. Sample the 128 experiences from the replay buffer and update the actor-critic network to get **states, actions, rewards, next_states, dones.**
10. Calculate the target Q value as **Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))** such that:
    a. Obtain the **actions_next** from the training of the actor target network with using the **next_states** as an input.
    b. Obtain the **Q_targets_next** from the training of the critic target network with using the **next_states** and **actions_next** as an input.
11. Calculate the expected Q value from the training of the local critic network with using the current **states** and **actions** as an input.
12. Compute the Temporal Difference error between **Q_expected** and **Q_targets**.
13. Perform the update of the Critic target network weights with gradients calculated from this loss *L* by minimizing the Mean Squared Error between **Q_expected** and **Q_targets**.
14. Update our policy network weights using a policy gradient by using:
    a. Obtain the **actions_pred** from the training of the local actor network with using the **current states** as an input.
    b. Obtain the **Q_values** from the training of the local critic network with using the **current states** and **actions_pred** as an input.
    c. Maximize the **actor loss** obtained from the average of **Q_values**.
15. The target networks are then updated using a **soft updates strategy**. A soft update strategy consists of slowly blending the regular network updates with the target network updates. So at every time step, a mix in **0.1% of regular/ local network weights** with **99.9% of the target network weights**. The regular network is the most up to date network because it is the one where training while the target network is the one used for prediction to stabilize training.
16. Update slowly the weights of the target networks (actor and critic) will provide greater stability with using the **soft updates strategy**.

DDPG Algorithm

DDPG algorithm can be shown as below:

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:

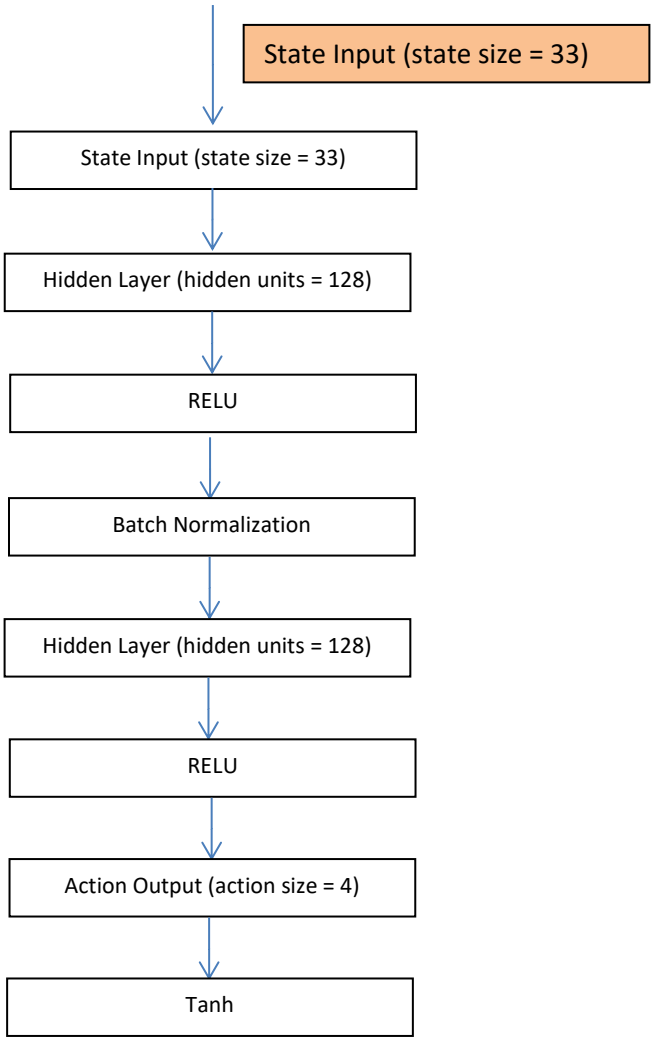
$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Model Architecture

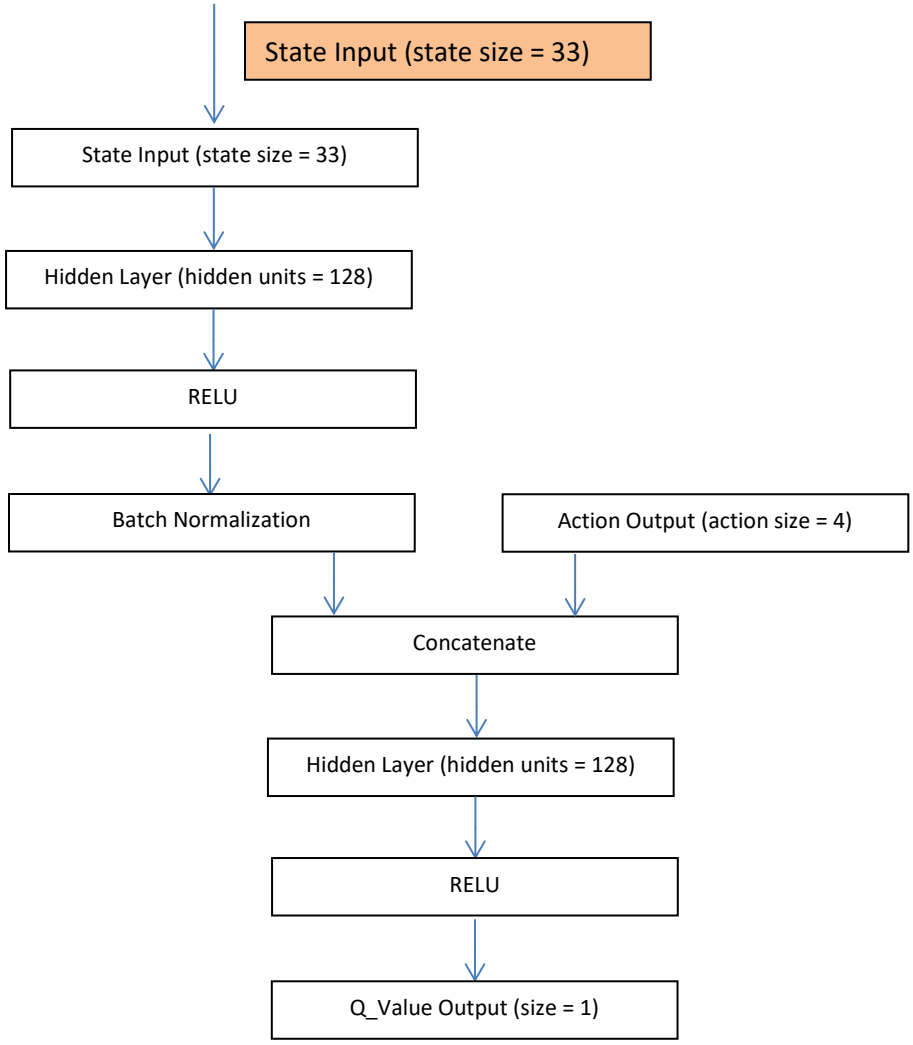
Actor Network

This represents the model architecture of the Actor network.



Critic Network

This represents the model architecture of the Critic network.

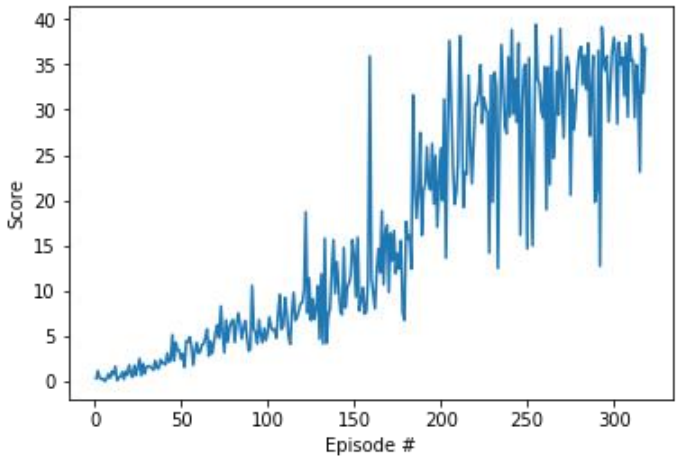


Hyperparameters

BUFFER_SIZE	1e6	replay buffer size
BATCH_SIZE	128	minibatch size
GAMMA	0.99	Discount factor
TAU	1e-3	for soft update of target parameters
LR_ACTOR	2e-4	learning rate of the actor
LR_CRITIC	2e-4	learning rate of the critic
WEIGHT_DECAY	0	Weight Decay
theta	0.15	how “fast” the variable reverts towards to the mean
Sigma	0.1	It is the degree of volatility of the process

Plot of Rewards

A plot of rewards per episode is included to illustrate **version 1** in which the agent receives an average reward (over 100 episodes) of at least +30.



Episode 100	Average Score: 3.19
Episode 200	Average Score: 12.62
Episode 300	Average Score: 29.66
Episode 318	Average Score: 31.05
Environment solved in 318 episodes!	**Average Score: 31.05**

Ideas for Future Work

- Implementing methods of Trust Region Policy Optimization (TRPO) and Truncated Natural Policy Gradient (TNPG) that may achieve better performance according to the Benchmark of a range of implemented reinforcement learning algorithms **[11]**.
- Implementation with using Proximal Policy Optimization (PPO) to get excellent results with relatively little hyperparameter tuning **[12]**.

References

1. https://spinningup.openai.com/en/latest/algorithms/ddpg.html
2. https://github.com/dennybritz/reinforcement-learning/tree/master/PolicyGradient
3. https://channel9.msdn.com/Events/Neural-Information-Processing-Systems-Conference/Neural-Information-Processing-Systems-Conference-NIPS-2016/Deep-Reinforcement-Learning-Through-Policy-Optimization
4. https://arxiv.org/pdf/1509.02971v5.pdf
5. https://arxiv.org/pdf/1803.05044.pdf
6. https://www.quora.com/Why-do-we-use-the-Ornstein-Uhlenbeck-Process-in-the-exploration-of-DDPG/answer/Edouard-Leurent?ch=10&share=4b79f94f&srid=udNQP

7. https://medium.com/@markus.x.buchholz/deep-reinforcement-learning-deep-deterministic-policy-gradient-ddpg-algoritm-5a823da91b43
8. https://ai-mrkogao.github.io/reinforcement%20learning/ActorCriticTensorflow/
9. https://bzdww.com/article/161901/
10. https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html
11. https://arxiv.org/abs/1604.06778
12. https://openai.com/blog/openai-baselines-ppo/