Deep Reinforcement Learning – Collaboration and Competition

Overview

The goal of this project is to train an agent to play tennis by controlling their rackets to bounce the ball over the net and keep it under play mode as long as possible without hitting the ball to ground.

In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.
This yields a single score for each episode.
The environment is considered solved, when the average (over 100 episodes) of those scores is at least +0.5.

Implementation

The project was solved using Deep Reinforcement Learning using the Multi-Agent version of DDPG(Deep Deterministic Policy Gradient) Algorithm.

The base code of the project is derived from my implementation of the previous project in Deep Reinforcement Learning Nanodegree for Continuous Control in a Reacher environment where I had used DDPG.

https://github.com/ManishaJhunjhunwala/Continuous_Control_DDPG_UdacityDRLND

The same solution was modified and updated for the new environment. In the previous environment there were 20 Agents which interacted with the environment independently. The same code was re-used since in this case, both the agents were trained while they interacted with eachother.

The same solution was modified and updated for the Unity ML Agents environment provided.

- The notebook **Tennis.ipynb** contains the implementation for training and visualising the untrained agent. Then the training code is implemented.
- **ddpg_agent.py** contains the code to understand and determine how the agent interacts with the environment and learns to optimize the reward.
- **model.py** contains the architecture of the deep learning model used in this implementation.

Learning algorithm

The ***Deep Deterministic Policy Gradient algorithm(DDPG)*** was chosen to solve the agent-environment interaction.

In general, the more commonly known Q-Learning method utilizes a state-action return value and from there chooses the actions which maximise the expected reward, hence reaching the optimal policy.

Policy gradients on the other hand tries to direcly map the states to the actions where a neural network can be used as a function approximator, by increasing the probabilities of the actions which yield higher returns, leading to the optimal policy.

DDPG combines these two ideas in the form of an actor-critic algorithm. It is based on the deterministic policy gradient that can operate over continuous action spaces.

This algorithm is outlined in this paper, Continuous Control with Deep Reinforcement Learning, by researchers at Google Deepmind. In this paper, the authors present "a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces." They highlight that DDPG can be viewed as an extension of Deep Q-learning for continuous tasks.

Actor-Critic Method

Actor-critic methods leverage the strengths of both policy-based and value-based methods.

Using a policy-based approach, the agent (actor) learns how to act by directly estimating the optimal policy and maximizing reward through gradient ascent. Meanwhile, employing a value-based approach, the agent (critic) learns how to estimate the value (i.e., the future cumulative reward) of different state-action pairs. Actor-critic methods combine these two approaches in order to accelerate the learning process. Actor-critic agents are also more stable than value-based agents, while requiring fewer training samples than policy-based agents.

You can find the actor-critic logic implemented in the file **ddpg_agent.py**. The actor-critic models can be found via their respective Actor() and Critic() classes in **model.py**.

In the algorithm, local and target networks are implemented separately for both the actor and the critic.

```
    # Actor Network (w/ Target Network)
    self.actor_local = Actor(state_size, action_size, random_seed).to(device)
    self.actor_target = Actor(state_size, action_size, random_seed).to(device)
    self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

    # Critic Network (w/ Target Network)
    self.critic_local = Critic(state_size, action_size, random_seed).to(device)
    self.critic_target = Critic(state_size, action_size, random_seed).to(device)
    self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=LR_CRITIC,
weight_decay=WEIGHT_DECAY)
```

Exploration vs Exploitation

One challenge is choosing which action to take while the agent is still learning the optimal policy. Should the agent choose an action based on the rewards observed thus far? Or, should the agent try a new action in hopes of earning a higher reward? This is known as the exploration-exploitation dilemma.

In the Navigation project, this is addressed by implementing an ε-greedy algorithm. This algorithm allows the agent to systematically manage the exploration vs. exploitation trade-off. The agent "explores" by picking a random action with some probability epsilon ϵ. Meanwhile, the agent continues to "exploit" its knowledge of the environment by choosing actions based on the deterministic policy with probability (1-ϵ).

However, this approach won't work for controlling a robotic arm. The reason is that the actions are no longer a discrete set of simple directions (i.e., forward, backward, left, right). The actions driving the movement of the arm are forces with different magnitudes and directions. If we base our exploration mechanism on random uniform sampling, the direction actions would have a mean of zero, in turn cancelling each other out. This can cause the system to oscillate without making much progress.

Instead, we'll use the Ornstein-Uhlenbeck process, as suggested in the previously mentioned paper by Google DeepMind (see bottom of page 4). The Ornstein-Uhlenbeck process adds a certain amount of noise to the action values at each timestep. This noise is correlated to previous noise, and therefore tends to stay in the same direction for longer durations without canceling itself out. This allows the arm to maintain velocity and explore the action space with more continuity.

You can find the Ornstein-Uhlenbeck process implemented in the OUNoise class in ddpg_agent.py.

In total, there are five hyperparameters related to this noise process.

The Ornstein-Uhlenbeck process itself has three hyperparameters that determine the noise characteristics and magnitude:

mu: the long-running mean
theta: the speed of mean reversion
sigma: the volatility parameter

The final noise parameters were set as follows:

```
OU_SIGMA = 0.1        # Ornstein-Uhlenbeck noise parameter
OU_THETA = 0.15       # Ornstein-Uhlenbeck noise parameter
```

Experience Replay

Experience replay allows the RL agent to learn from past experience.

DDPG also utilizes a replay buffer to gather experiences from each agent. Each experience is stored in a replay buffer as the agent interacts with the environment. In this project, there is one central replay buffer utilized by all 20 agents, therefore allowing agents to learn from each others' experiences.

The replay buffer contains a collection of experience tuples with the state, action, reward, and next state (s, a, r, s'). Each agent samples from this buffer as part of the learning step. Experiences are sampled randomly, so that the data is uncorrelated. This prevents action values from oscillating or diverging catastrophically, since a naive algorithm could otherwise become biased by correlations between sequential experience tuples.

Also, experience replay improves learning through repetition. By doing multiple passes over the data, our agents have multiple opportunities to learn from a single experience tuple. This is particularly useful for state-action pairs that occur infrequently within the environment.

The following sections describe important concepts related to the Multi Agent DDPG algorithm.

Policy-Based vs. Value-Vased Methods

There are two key differences between this project's Tennis environment and the first project's Banana Collector environment.:

Multiple agents — The Tennis environment has 2 agents, while the Navigation project had a single agent.

Continuous action space — The action space is now continuous, which allows each agent to execute more complex and precise movements. Essentially, there's an unlimited range of possible action values to control the racket's movement toward (or away from) the net, and jumping, whereas the agent in the Navigation project was limited to four discrete actions: forward, backward, left, right.

Given the complexity of this environment, the value-based method we used in the first project - the Deep Q-Network (DQN) algorithm - is not suitable. We need an algorithm that allows the agents' rackets to utilize its full range of movement. We'll need to explore a different class of algorithms called policy-based methods.

Some advantages of policy-based methods:

- **Continuous action spaces** — Policy-based methods are well-suited for continuous action spaces.
- **Stochastic policies** — Both value-based and policy-based methods can learn deterministic policies. However, policy-based methods can also learn true stochastic policies.
- **Simplicity** — Policy-based methods directly learn the optimal policy, without having to maintain a separate value function estimate. With value-based methods, the agent uses its experience with the environment to maintain an estimate of the optimal action-value function, from which an optimal policy is derived. This intermediate step requires the storage of lots of additional data since you need to account for all possible action values. Even if you discretize the action space, the number of possible actions can be quite high. For example, if we assumed only 10 degrees of freedom for both joints of our robotic arm, we'd have 1024 unique actions (210). Using DQN to determine the action that maximizes the action-value function within a continuous or high-dimensional space requires a complex optimization process at every timestep.

Neural Network

As implemented in the file model.py, both Actor and Critic (and local & target for each) consists of :

1. Three (3) fully-connected (Linear) layers.
2. The input to fc1 is state_size, while the output of fc3 is action_size.
3. There are 400 and 300 hidden units in fc1 and fc2, respectively, and
4. **batch normalization (BatchNorm1d) **is applied to fc1.
5. ReLU activation is applied to fc1 and fc2,

6. tanh is applied to fc3.

MODEL ARCHITECTURE

There are two model architectures used here defined in the file **model.py** :

Actor

- The model has 2 fully connected layers with 400 and 300 nodes respectively.
- It takes in an input equal to the state size provided which in this case is 24
- Relu activation function is used betweeen the two layers
- Batch normalisation has been applied since it has been proven that it speeds up the learning in case of DDPG
- The output node of this layer is of size equal to the action size

Critic

- The model has 2 fully connected layers with 400 nodes in the first layer and (300+action_size) in the second.
- It takes in an input equal to the state size provided which in this case is 24
- Relu activation function is used betweeen the two layers
- Batch normalisation has been applied since it has been proven that it speeds up the learning in case of DDPG
- The output node of this layer is of size equal to 1 which means that it outputs the expected return value
- torch.cat operation is applied between the first two layers to denote the mapping between the states and actions as defined in the algorithm

Agent

The agent is defined in the agent.py file. It is the Deep Q-Learning agent which interacts with the environment. It references the local and target network from the model defined in model.py.

It contains four methods:

Step: Here the agent saves the experiences in the replay memory. After a certain set of predefined intervals, it also causes the network to learn from the replay buffer a certain number of times.

Act: Here the agent returns the action determined by the local Actor Network. The output is of size 4 corresponding to each actions but in the range of -1 to 1 as expected by the network. Furthermore, noise is added via the Ornstein-Uhlenbeck process to encourage exploration.

Learn: This is where the agent actually learns.

Critic network learning

We randomly sample a batch from the experience buffer in the form of (states, actions, rewards, next_states, dones) and pass on the next_states to the actor target network to determine the next set of actions which in return is passed on to the critic target network.

The return from the critic target network is actually the Q values determined by the target network.

Again, like in the DQN Algorithm, we compute the expected Q values from these next states and compute the Mean Squared Error loss between Q_targets and Q_expected and update the critic target network accordingly.

Actor network learning

Here we get the predicted actions from the local actor network based on the current states. The loss is computed as the mean of the Q values corresponding to the different state action pairs. We use the negative sign here because we want to maximise the gradients and hence using gradient ascent.

Then we run the soft_update function to update the target network with the local network parameters
Soft_update: Here we update the target networks with the local networks parameters using the formula

$$\theta\_target = \tau*\theta\_local + (1 - \tau)*\theta\_target$$

Modification from the original DDPG implementation for Multi-Agent Scenario¶

In our current environment the agent learns by interacting with eachother. In this case I am reusing the same Actor-Critic Network with the mindset that we learn both positive or negative outcomes from mutually opposing agents who have in other words 'learnt' the from eachother's experiences.

Hyperparameters

The hyperparameters used to train the agent are:

- BUFFER_SIZE = int(1e6) # replay buffer size
- BATCH_SIZE = 256 # minibatch size
- GAMMA = 0.99 # discount factor
- TAU = 2e-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 # L2 weight decay
- LEARN_EVERY=1 # Update interval
- LEARN_NUM=1 # Number of learning steps after every predefined interval in LEARN_EVERY

NOTE: The files ddpg_agent.py was taken from the 'Bipedal' gym environment and model.py was taken from the 'Pendulum-v0' gym environment.

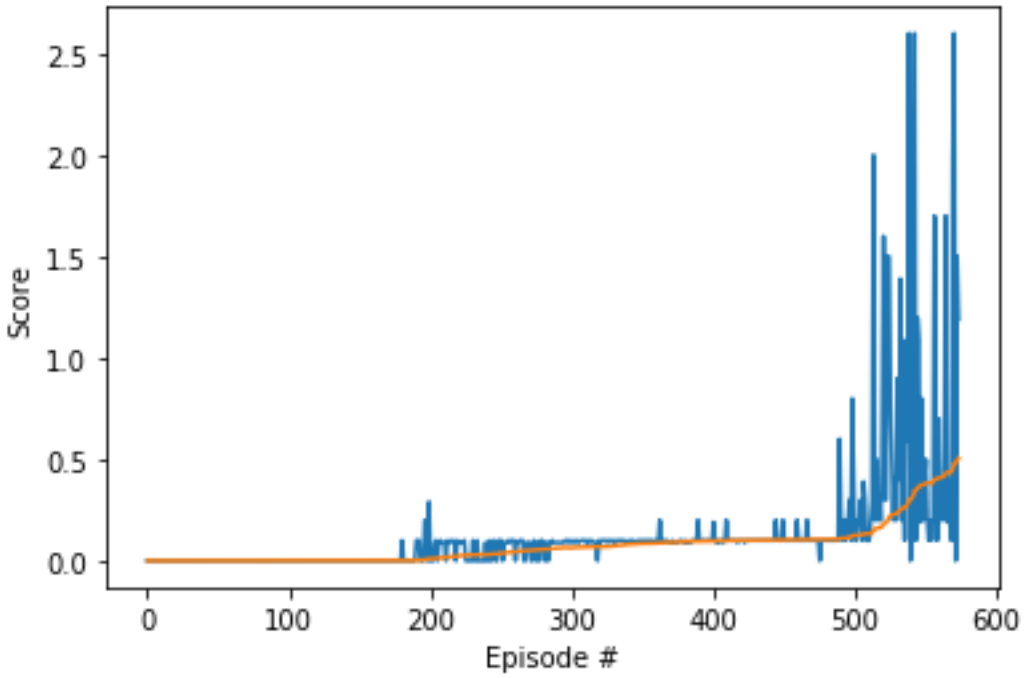
Replay Buffer

One of the breaks through with the DQN architecture was the introduction of Experience replay. It stores experiences in a replay buffer so that learning is separated and can take place on both the current experience and past experiences. This allows us to use Supervised Deep Learning techniques. This is done by randomly sampling the stored tuples in the replay buffer for training the model. The code is under the ReplayBuffer Class in **agent.py .**

Results

The results from the DDPG were impressive. It easily achieved an average score of 0.51 in 574 episodes.

```
Episode 100     Average Score: 0.00     Score: 0.00
Episode 200     Average Score: 0.01     Score: 0.29
Episode 300     Average Score: 0.06     Score: 0.10
Episode 400     Average Score: 0.10     Score: 0.10
Episode 500     Average Score: 0.13     Score: 0.39
Episode 574     Average Score: 0.51     Score: 1.19
Environment solved in 574 episodes!     Average Score: 0.51
```



Improvements

While results achieved were impressive, there is always room for improvement. This was a very interesting project because it needed a lot of hyperparameter tuning to find the right mix of hyperparams to achieve a score of 0.5 in a decent number of episodes.

The Reinforcement Learning agent was trained using Deep Deterministic Policy Gradients extended to work over multiple agents which are interacting over multiple agents

- Here in this case I have used Random Sampling as in the previous project, but Prioritized Experience Replay might show better results in comparison.
- I might need to optimize the code further to improve the stability since the agent does start to learn quite later in the journey and as shown in the graph, gives huge spikes in final scores.

- The agent currently converges after variable number of steps even with the same hyperparameters. Further modification to them might yield more consistent results.
- I stopped training when the target goal was reached, but further training for a longer period of time might lead to more insights on how long the current agent maintains its stability

Conclusion

The project was a fantastic learning experience. Troubleshooting various parts of the agent and playing around with endless variation of hyperparameters and Neural Network architectures was particularly useful. Understanding how small changes can make a significant difference in performance and seeing the algorithms working to solves problems is incredible.

More is not always better. Too many layers or nodes in a neural network can rapidly slow down training and cause it to overfit.

Modifying the TAU and learning rate can change the outcome and speed of training. I recommend playing with those parameters while you run this project.