Report

October 21, 2019

See the `README` for background information of the project and how to setup Unity ML-Agents Reacher environment used. This report contains detail of **Actor-Critic** algorithm and the model used for training the agent

1 1. The Learning Algorithm

The learning algorithm used in this project is called **Actor-Critic Method**.

1.0.1 Actor - Critic Summary

On a high level: The actor in actor-critic method does its best to respond to the environment to maximize its reward The critic responds immediately on how well actor is doing

Detailed explanation with math Basic RL models are either **value-based** or **policy-based**.

- **Value Based Methods:** Methods where we try to learn a value function that will map each state-action pair to a value. Using this, we can find for an agent which action will maximize the value given a state, which will be the action with the biggest value. E.g - Q-learning, Deep Q-learning
- **Policy Based Methods:** Methods where we directly try to optimize the policy without a value function. This is useful when an action-space is continuous or stochastic. Main problem is finding a score function of how good the policy is. We use total rewards of the episode. E.g - REINFORCE with policy gradients

Actor-critic is a hybrid of both of these methods. It uses two neural networks:

- An actor that controls how our agent behaves (policy-based)
- A critic that measures how good action taken is (value-based)

The Actor-Critic model differs from more elementary learning processes in that instead of waiting until the end of the episode as in Monte Carlo REINFORCE, an update is made at each time step (TD Learning):

$ Policy Update Rule $

$$\Delta\theta = \alpha * \nabla_\theta * (log\pi(S_t, A_t, \theta)) * R(t)$$

$ Actor-Critic Update Rule: $

$$\Delta\theta = \alpha * \nabla_\theta * (log\pi(S_t, A_t, \theta)) * Q(S_t, A_t)$$

Here, in actor-critic, we changed the reward with the Q-function, which will be given by critic. It'll be treated as a proxy for reward. As we do an update at each time step, we can't use `R(t)`. Thats why we use critic that approximates value function, which replaces the reward in policy gradient method

1.0.2 Model Architecture

The model architecture is an actor and a critic neural network with 3 fully-connected layers. The model code is in `model.py`

- Parameters:
  - state_size
  - action_size
  - seed
  - fc1_units
  - fc2_units
  - leakiness = 0.01
- Layer 1 took `state_size` as input and output `fc1_units`
- Layer 2 took `fc1_units` and output `fc2_units`
- Layer 3 took `fc2_units` and output `action_size` for actor and `1` for the critic
- Activation function used was `leaky_relu` with leakiness = 0.001
- Gradient clipping was applied to stabilize learning
- Batch Norm was also applied on the first layer of the network
- Forward returns via `tanh` for actor and directly for critic

2 Agent architecture and hyperparameters

Agent code is in `ddpg_agent.py`. The agent: * utilizes the actor and critic neural networks, which it will train using the `Adam` optimizer * has an experience replay buffer * Buffer contains experience tuples in form $(state, action, reward, nextstate)$ * Repeated passes from buffer allow agent to learn from experience * adds Ornstein-Uhlenbeck noise so that agent can learn better

The hyperparameters of the model are also defined in `ddpg_agent.py`. These hyperparameters were also used by other people who got pretty good results.

Here are the various hyperparamters used:

```
BUFFER_SIZE             = int(1e6)                    # Replay buffer size
BATCH_SIZE              = 128                         # Mini-batch size
GAMMA                   = 0.99                        # Discount factor
TAU                     = 1e-3                        # Soft-update target parameters
LR_ACTOR                = 1e-3                        # Learning Rate of Actor
LR_CRITIC               = 1e-3                        # Learning Rate of Critic
WEIGHT_DECAY            = 0                           # L2 Weight Decay
LEAKINESS               = 0.01                        # Leakiness of leaky_relu
```

```
LEARN_EVERY            = 20                    # Learning timestep interval
LEARN_NUM              = 10                    # Number of learning passes
GRAD_CLIPPING          = 1.                    # Gradient Clipping

# Ornstein-Uhlenbeck noise
OU_SIGMA               = 0.2
OU_THETA               = 0.15

EPSILON                = 1.                    # for epsilon in noise
EPSILON_DECAY          = 1e-6
```

3 2. The Reacher Agent's state and action space

In this environment, a double-jointed arm can move to target locations, which consist of moving balls. A reward of `+0.1` is provided for each step that the agent's hand is in the goal location. Thus, the goal of the agent is to maintain its position at the moving target location for as many time steps as possible.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector must be a number between `-1` and `1`.

The version of the model I trained on has 1 target/agent pair. This agent takes more time to converge than the 20 target/agent pair one as it doesn't have the advantage of multiple agents which can learn from other's experiences

4 3. The training loop

The DRLND project was accompanied by information gained from a Udacity benchmark implementation of the environment, where it was described how to implement a training loop to cater for the agent, which was based on prior work performed on a single agent Unity ML-Agents environment

- Training is complete when the average of the last 100 episodes reaches 30+.
- Gradient clipping was employed to stabilize the training.
- Further clues for success came from fellow students on the Slack channel dedicated to this nanodegree, some of which was very useful, some was confusing!

The loop

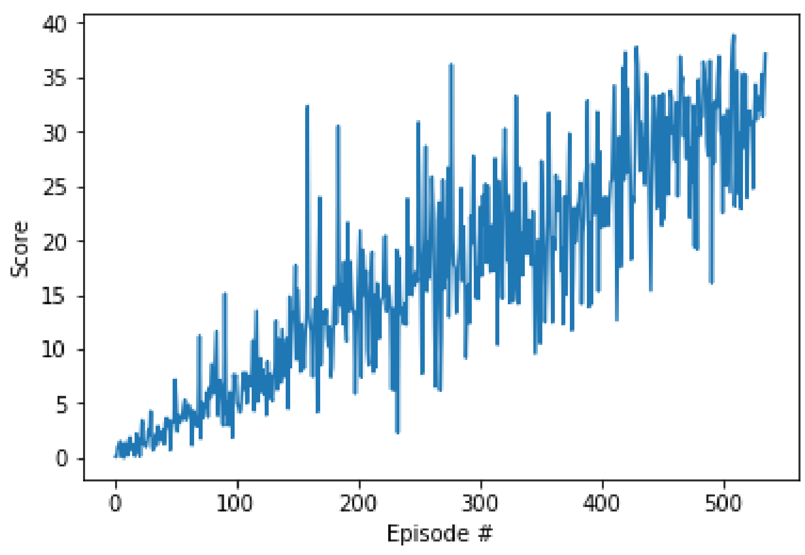
```
def ddpg(n_episodes=1000, max_t=5000, print_every=100):
    scores = []
    scores_deque = deque(maxlen=print_every)

    for i_episode in tqdm(range(1, n_episodes+1)):
        env_info = env.reset(train_mode=True)[brain_name]
        agent.reset()
        state = env_info.vector_observations[0]
        score = 0

        for t in range(max_t):
            action = agent.act(state)

            env_info = env.step(action)[brain_name]
            next_state = env_info.vector_observations[0]
            reward = env_info.rewards[0]
            done = env_info.local_done[0]

            accumulate = num_agents > 1

            agent.step(state, action, reward, next_state, done, t)
            score+=reward

            state = next_state

            if done: break
        scores_deque.append(score)
        scores.append(score)

        if i_episode % print_every == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_deque))
            torch.save(agent.actor_local.state_dict(), 'checkpoint_actor.pth')
            torch.save(agent.critic_local.state_dict(), 'checkpoint_critic.pth')

        if np.mean(scores_deque)>=30.:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episo
            torch.save(agent.actor_local.state_dict(), 'checkpoint_actor.pth')
            torch.save(agent.critic_local.state_dict(), 'checkpoint_critic.pth')
            break

    return scores
```

5 4. Plot of rewards

An average score of `30.02` was achieved in 535 episodes. This score can be further improved by optimizing the hyperparameters

```
scores = ddpg(n_episodes=600)
```

```
 16%|█          | 99/600 [14:47<1:13:58,  8.86s/it]

Episode 100    Average Score: 3.47

 33%|███        | 200/600 [29:55<1:00:11,  9.03s/it]

Episode 200    Average Score: 10.84

 50%|█████      | 300/600 [45:14<47:21,  9.47s/it]

Episode 300    Average Score: 16.62

 67%|██████▋    | 400/600 [1:01:27<33:18,  9.99s/it]

Episode 400    Average Score: 20.91

 83%|████████▎  | 500/600 [1:18:20<17:25, 10.45s/it]

Episode 500    Average Score: 28.49

 89%|████████▉  | 534/600 [1:24:25<11:45, 10.69s/it]


Environment solved in 535 episodes!     Average Score: 30.02
```



6 5. Ideas for Future work

- **Prioritized Experience Replay:** Replace selection of random experiences with prioritized experience replay which could help the agent learn much better
- **Try other architectures:** DDPG is a good algorithm no doubt, but many other algorithms have been much more significant. PPO and D4PG may be the architectures which i will try.
- **Solve the crawler environment:** This environment will be beneficial to apply the new skills i learned

```
In [ ]:
```