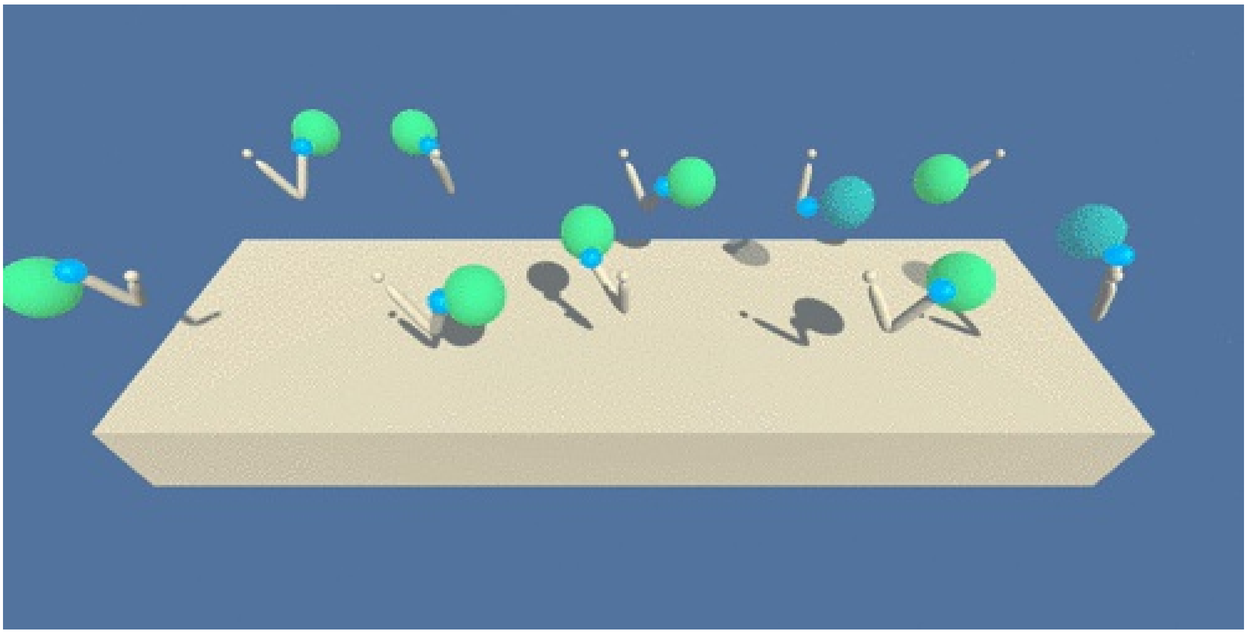
report

Environment

The Reacher Environment with 20 agents. It has a continuous action space [-1,1] and 33 state size. It is a double jointed arm which can move in all directions and the environment considered solved when it is able to follow a specific trajectory (follow the ball) and keep it. For this project we consider it solved when it reaches an average of 30 points for 100 consecutive episodes.



First Experiment: Deep Deterministic Policy Gradient Algorithm (DDPG)

I have used the implementation found in the lessons which is easy to understand but not so much modular. DDPG is an off-policy actor critic method which is a modification of the DQN algorithm for continuous action spaces. DDPG is basically the older DPG algorithm but with DQN for the function approximation instead of the bellman equations.

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

Some explanations:

Critic network: $Q\left(s, a | \theta^{Q}\right)$, Actor Network: $\mu\left(s | \theta^{\mu}\right)$

Weights Critic: θ^{Q} Weights Actor θ^{μ}

Update critic $y_{i}=r_{i}+\gamma Q^{\prime}\left(s_{i+1}, \mu^{\prime}\left(s_{i+1} | \theta^{\mu^{\prime}}\right) | \theta^{Q^{\prime}}\right)$

```
actions_next = self.actor_target(next_states)
Q_targets_next = self.critic_target(next_states, actions_next)
# Compute Q targets for current states (y_i)
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
```

$L=\frac{1}{N} \sum_{i}\left(y_{i}-Q\left(s_{i}, a_{i} | \theta^{Q}\right)\right)^{2}$

```
critic_loss = F.mse_loss(Q_expected, Q_targets)
```

$\nabla_{\theta^{\mu}} J \approx \mathbb{E}{s{t} \sim \rho^{\beta}}\left[\nabla_{\theta^{\mu}}

Q\left.\left(s, a | \theta^{Q}\right)\right|{s=s{t}, a=\mu\left(s_{t} | \theta^{\mu}\right)}\right]$

$\nabla_{\theta^{\mu}} J \approx \frac{1}{N} \sum_{i} \nabla_{a} Q\left.\left(s, a | \theta^{Q}\right)\right|{s=s{i}, a=\mu\left(s_{i}\right)} \nabla_{\theta^{\mu}} \mu\left.\left(s | \theta^{\mu}\right)\right|{s{i}}$

```
actions_pred = self.actor_local(states)
actor_loss = -self.critic_local(states, actions_pred).mean()
```

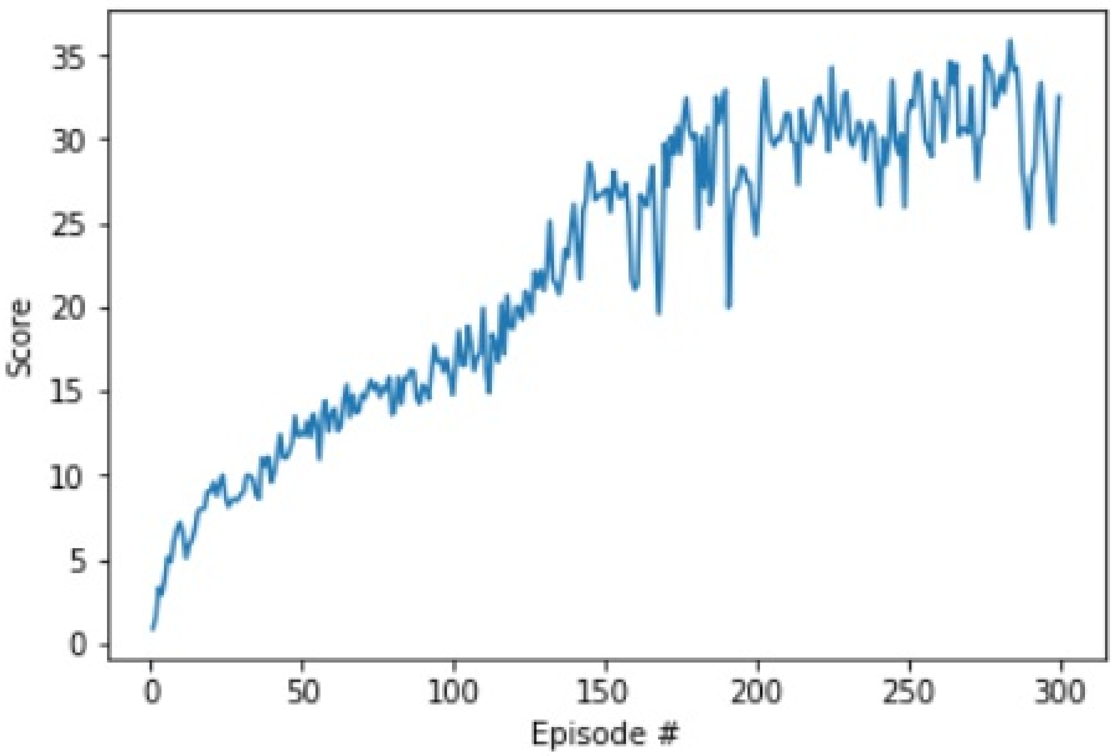
The network is the same from the lessons. An actor with two FC layers (256) and a critic with four FC layers of (256,256,128)

Hyperparameters and details

- Every 20 timesteps I update the network 10 times.
- I'm not explicitly using a `max_t`. When any of the 20 agents terminates I finish the iterations and move on to the next episode.

###Results

The algorithm takes more than 5 hours to finish on a moderate GPU and it converges slowly at around 300 episodes.



Second Experiment: Proximal Policy Optimization algorithm (PPO)

About the PPO Algorithm

PPO or Proximal Policy Optimization algorithm is an Open AI algorithm released in 2017 that gives improved performance and stability against DDPG and TRPO.

Algorithm 1 PPO, Actor-Critic Style

for iteration$=1,2,\ldots$ **do**
    for actor$=1,2,\ldots,N$ **do**
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
        Compute advantage estimates $\hat{A}_1,\ldots,\hat{A}_T$
    end for
    Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$ $L^{CLIP}(\theta) = \hat{E}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\right]$
end for

and with words:

1. First, collect some trajectories based on some policy π_θ, and initialize theta prime $\theta'=\theta$
2. Next, compute the gradient of the clipped surrogate function using the trajectories
3. Update θ' using gradient ascent $\theta'\leftarrow\theta' +\alpha \nabla_{\theta'}L_{\rm sur}^{\rm clip}(\theta', \theta)$
4. Then we repeat step 2-3 without generating new trajectories. Typically, step 2-3 are only repeated a few times
5. Set $\theta=\theta'$, go back to step 1, repeat.

I have used the very modular implementation of https://github.com/ShangtongZhang/DeepRL and I applied it for our current unity environment which
is somewhat different from the Open AI ones.

PPO uses an Actor Critic network. In each step the agent executes a two rollout steps and a learning step. First the actor network generates the actions and the the critic network generates the predictions for those actions.

In the learning step and for a number of epochs, the agent performs training and optimizes the objective function.

The network used is an actor-critic network that is similar to the paper.

```
GaussianActorCriticNet(
        config.state_dim, config.action_dim, actor_body=FCBody(config.state_dim),
        critic_body=FCBody(config.state_dim))
```

You can find more information about it in the `deep_rl` repository.

Hyperparameters

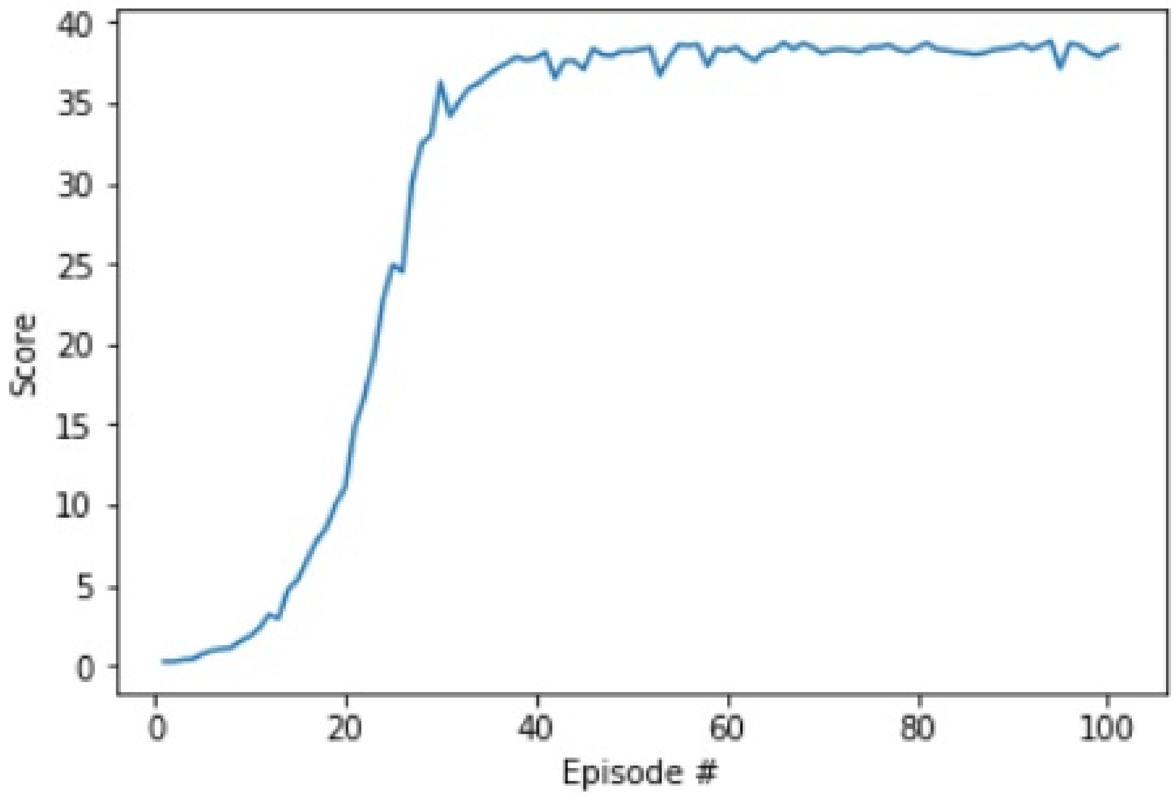
```
discount_rate = 0.99
gae_tau = 0.95
gradient_clip = 5
rollout_length = 2048*4
optimization_epochs = 10
mini_batch_size = 32
ppo_ratio_clip = 0.2
```

Some explanations:

- The `gradient_clip` parameter is to limit the magnitude of the gradient avoiding exploding gradients
- The `gae_tau` parameter is related to the general advantage estimation

###Results

The algorithm converges much faster than the DDPG (around 100 epochs) and displays lower variance.



Ideas for Future Work

I haven't experimented much with the chosen model architectures. For future work I would test simpler architectures and more complex ones. Also, I would try different hyper-parameters (although I have tried many).

I would also test the `A3C`, and `D4PG` algorithms to witness if they perform better.