Udacity Deep Reinforcement Learning Project 2 - Continuous Control

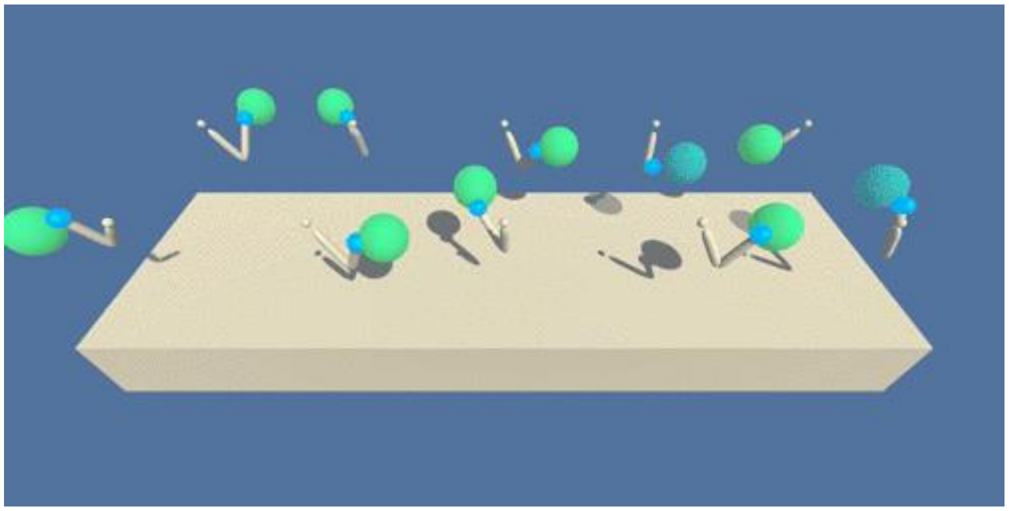
Jonathan Salfity | November, 2018

Continuous Control

Description

This project featured two simulation environments that the deep reinforcement learning student could choose from. Both environments featured a double-jointed arm trained to move to target locations. One simulation environment allowed only for a single agent to learn while the other agent allowed for 20 identical agents, each with its own copy of the environment, to learn. The second simulation environment was chosen for this project.

The agents were trained in a reinforcement learning framework, where each agent learned a policy to move to the target. Unity was used for the simulation environment and OpenAI Gym was used for the Reinforcement Learning implementations. The agents were trained using Proximal Policy Gradients in a manner very similar to the 'Proximal Policy Optimization Algorithms', J. Schulman et al, 2017.



Observation Space, Reward Structure, Action Space

There are 20 agents in this environment. Each agent has a continuous observation space with 33 dimensions corresponding to position, rotation, velocity, and angular velocities of the arg. Each agent has a continuous action space of four dimensions, corresponding to torque applicable to two joints. A reward of +0.1 is provided for each step of the agent's hand. The goal of each agent is to maintain its position at the target location for as many steps as possible.

Learning Algorithm

The learning algorithm used was Proximal Policy Optimization (PPO), a policy gradient Reinforcement Learning algorithm. The novel objective proposed in PPO is clipped probability rations, which form a lower bound on the performance of the policy. During gradient ascent of the policy gradient, there is a limit on how far the policy can change in each iteration through the KL-divergence. KL-divergence measures the difference between two data distributions. In contrast to Trust Region Policy Optimization (TRPO), PPO is simpler to implement, more general and have better sample complexity.

Software Framework

A Jupyter notebook hosted on a Udacity server with GPU support was used to write code. PyTorch was used to build and train a deep neural network.

A large part of this project was built leveraging the beautiful repository from Shantong Zhang's github repository consisting, Deep RL, A Modularized implementation of popular deep RL algorithms by PyTorch.

Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a policy gradient method of Reinforcement Learning. The goal of reinforcement learning is to come up with a policy which maps states to actions. Policy gradient methods learn by computing an estimator of the policy gradient, testing the policy gradient during simulation, and update the weights of the policy gradient to maximize the cumulative sum of discounted future rewards.

A key difference between PPO and other policy gradients is that PPO changes its learned policy slowly, i.e. updates its policy with a policy in a near proximity. This may lead to a slower training, but seemingly more robust and efficient models.

In a similar fashion to PPO, Trust Region Policy Optimization (TRPO) determines a maximum step size to explore within, then chooses the optimal point within the trust region and resumes the search from there.

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] \\ \text{subject to} \quad & \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \leq \delta. \end{aligned}$$

The two above equations express the objective function and constraints. The objective, or surrogate function, is maximized subject to a constraint on the size of the policy update. $\theta_{old}$is the vector of policy parameters before the update and θ is the vector of policy parameters after the update.

PPO may use a Clipped Surrogate Objective function, where the distance between θ_{old} and θ is kept in a region by a hyperparameter, ϵ.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\Big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\Big]$$

where:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, \text{ so } r(\theta_{\text{old}}) = 1.$$

In addition, PPO may limit of how far the policy changes during each iteration is measured through the KL-divergence. The KL-divergence measures the different between two data distributions.

$$D_{KL}(P||Q) = \mathbb{E}[log\frac{P(x)}{Q(x)}]$$

PPO uses this KL divergence as a penalty and adapts the penalty coefficients so that the algorithm achieves some target value of the KL divergence each policy update.

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t - \beta\,\text{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]\right]$$

β is the hyperparameter in this case.

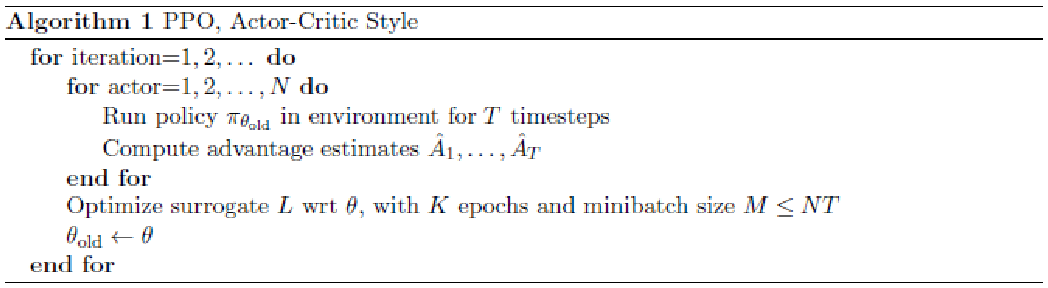
Deep Neural Network Model Architecture

The neural network was a Gaussian Actor-Critic structure. The Actor and Critic network contained 2 layers, each with 128 hidden units with ReLU activation.

```
CategoricalActorCriticNet(
  (network): ActorCriticNet(
    (phi_body): FCBody(
      (layers): ModuleList(
        (0): Linear(in_features=4, out_features=64, bias=True)
        (1): Linear(in_features=64, out_features=64, bias=True)
      )
    )
    (actor_body): DummyBody()
    (critic_body): DummyBody()
    (fc_action): Linear(in_features=64, out_features=2, bias=True)
    (fc_critic): Linear(in_features=64, out_features=1, bias=True)
  )
)
```

Training the Network

The PPO algorithm used fixed-length trajectory segments. Each iteration was run using N number of parallel actors collecting T timesteps of data. The surrogate loss on the NT timesteps of data is constructed and optimized with minibatch SGD for K epochs. Below is the algorithm from the PPO paper.



Algorithm 1 PPO, Actor-Critic Style

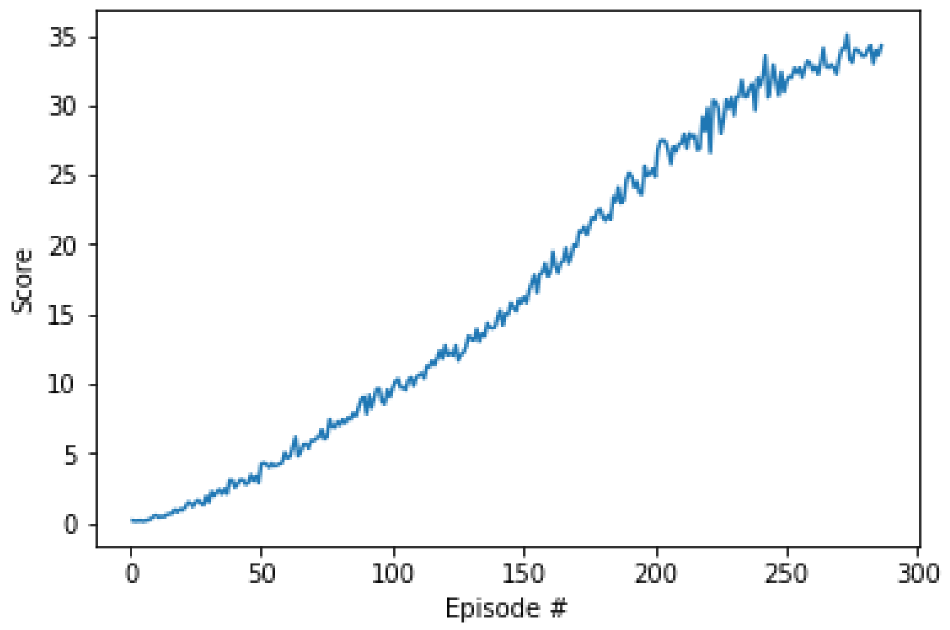
```
for iteration=$1, 2, \ldots$ do
    for actor=$1, 2, \ldots, N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

HyperParameters

Hyperparameters were configured using the Config() class:

```
config.num_workers = 20
config.discount = 0.99
config.use_gae = True
config.gae_tau = 0.95
config.gradient_clip = 5
config.rollout_length = 4096
config.optimization_epochs = 10
config.num_mini_batches = 256
config.ppo_ratio_clip = 0.2
config.log_interval = 2048
config.max_steps = 2e7
```

Results



PPO does converge slower than some other policy gradient methods, due to the limit in policy update. Because this learning used 20 agents in parallel, the policy was able to converge around 300 episodes.

Ideas for Future Work

Network architecture and hyperparamters are always the first parameters to tune when optimizing a reinforcement learning agent. The large input state space could warrant more hidden layers, to see if the agent can learn more representations of information. Also, the state vector could include previous states, similar to how DQN by DeepMind included previous Atari frames.

Appendix

Proximal Policy Optimization, https://arxiv.org/abs/1707.06347

https://blog.openai.com/openai-baselines-ppo/

https://medium.com/@jonathan_hui/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12

https://github.com/ShangtongZhang/DeepRL