1. A short Review of Policy-Based Method

- difference between value-based and policy-based method
- value-based need to establish Q table, and build the optimal policy based on maximum q value
- policy-based method is to directly estimate $\pi(s, a)$
- The **KEY** to optimize policy-based methods is the gradient ascent due to expected return.
- Gradient here is expected return w.r.t policy parameters, i.e. weights of deep neural network

1.2 Estimate Expected Return

- There are three ways to estimate expected return.
- Monte-Carlo learning, we use expected return from episodes to update gradient.
- Temporal Difference (TD) learning, which use current reward and a future estimation to update gradient during episode
- N-step bootstrap, which is general form of Monte-Carlo learning and TD learning. TD is a one step bootstrap, and Monte-Carlo is a infinite step bootstrap.

1.3 REINFORCE

- REINFORCE is a very straightforward policy-based method.
- REINFORCE is a Monte-Carlo learning, update policy per episode.
- REINFORCE use stochastic policy.
- we solve the policy distribution, and get action from sampling of this policy distribution.

Model

- **one** trajectory τ is state-action sequence

$$\tau = s_0, a_0, s_1, a_1, \ldots, s_H, a_H, s_{H+1}$$

- H is horizon
- total reward for **one** trajectory

$$R(\tau) = r_1 + r_2 + \cdots + r_H + r_{H+1}$$

- expected return for **multiple** trajectories

$$U(\theta) = \sum_{\tau} P(\tau, \theta) R(\tau)$$

- define problem as

$$\max_{\theta} U(\theta)$$

- update policy with gradient ascent

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} U(\theta)$$

Additional explanation

- for a **single** trajectory, the gradient is equivalent to derivative on cross-entropy loss

$$\begin{aligned} \nabla_{\theta} U(\theta) &= \nabla_{\theta} P(\tau, \theta) R(\tau) \\ &= \sum_{t=0}^{H} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) R(\tau) \end{aligned}$$

- Pseudocode

```
policy_loss = []
for log_prob in saved_log_probs:
policy_loss.append(-log_prob * R)
policy_loss = torch.cat(policy_loss).sum()
```

Algorithm

1. initialize a random $\pi_{\theta}(a|s)$
2. collect m trajectories $\{\tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(m)}\}$ with current policy π_{θ}
3. compute $R^{(i)} = r_1^{(i)} + r_2^{(i)} + ... + r_H^{(i)}$ with or without discounter
4. computer the gradient of expected reward U. e.g. for 1 trajectory

$$g^{(i)} = R^{(i)} \sum_{t}^{H} \nabla \log \pi_{\theta}(a_t^{(i)} | s_t^{(i)})$$

5. for m trajectories

$$\hat{g} = \frac{1}{m} \sum_{i=1}^{m} g^{(i)}$$

6. update with gradient ascent $\theta \leftarrow \theta + \alpha \hat{g}$
7. repeat 2

1.4 Importance Sampling

literally important !

Motivation

- we know that calculating expected return is critical.
- With TD learning, we update policy each step instead of each episode.
- But the training data is sampled from the policy. It means the sampled data changed after a step.
- We can not re-sample bunch of trajectories after each update step. It is too timely expensive.
- So we introduce importance sampling to solve this problem, to use old **SIMILAR** samples to calculate new expected returns from new policy, to save sampling time.

Model

- I agree this is too small to read, so I prefer understand this with an example. :-)

an Example instead of Explanation

- we have $f(1) = 2, f(2) = 3, f(3) = 4$ with $x \sim p, p(x = 1) = \frac{1}{3}, p(x = 2) = \frac{1}{3}, p(x = 3) = \frac{1}{3}$
- we denote $x \sim p$ as old policy, and we have the expectation of **old** samples as
$$\mathbb{E}_p[f(x)] = \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 3 + \frac{1}{3} \cdot 4 = 3$$
- Now we $x' \sim q$ as new policy, and we know $q(x = 1) = 0, q(x = 2) = \frac{1}{3}, q(x = 3) = \frac{2}{3}$
- So we want to use old samples to estimate the expectation of new policy q.

$$\begin{aligned}\mathbb{E}_q[f(x)] &= \mathbb{E}_p[f(x)\frac{q(x)}{p(x)}] \\ &= \mathbb{E}_p[f(x=2)]\frac{q(x=2)}{p(x=2)} + \mathbb{E}_p[f(x=3)]\frac{q(x=3)}{p(x=3)} \\ &= \frac{1}{3} \cdot 3 \cdot 1 + \frac{1}{3} \cdot 4 \cdot 2 = \frac{11}{3}\end{aligned}$$

- which is the same with
$$\mathbb{E}_q[f(x)] = \frac{1}{3} \cdot 3 + \frac{2}{3} \cdot 4 = \frac{11}{3}$$

- that is, important sampling
- this section was learned from this post

1.5 Trust region policy optimization (TRPO)

- Importance sampling and literally constrained by KL-divergence

1.6 Proximal Policy Optimization (PPO)

- Importance sampling and use clip function to approximately ensure the similarity.

2. Actor-Critic

Actor-Critic is architecture which combined policy-based method and value-based method.

2.1 Asynchronous Advantage Actor-Critic (A3C)

- I plan to write this section in next project

2.2 synchronous Advantage Actor-Critic (A2C)

- I plan to write this section in next project

2.3 Deep Deterministic Policy Gradient (DDPG)

DDPG was used for this Unity-Reacher project. I must elaborate DDPG.

- DDPG has a Actor-Critic Architecure
- Actor Network in DDPG is deterministic **NOT** stochastic. We can get action directly, no need to sample from policy.
- Actor Network is $\text{argmax}_a Q_\theta(s)$, given s, output a, where a let $Q(s)$ is max $Q(s, a)$
- Critic Network is $V_\theta(s, a)$, given (s, a), output action value $V_\theta(s, a)$, the same with notation $q(s, a)$

Train Process

- start a trajectory
- use s_t as input for Actor Network, then we get a_t from Actor Network, i.e. $\text{argmax}_a\, Q_\theta(s)$
- feedback a_t to Env, then we have r_{t+1}, s_{t+1}
- train Critic Netwrok, input $(s_t, a_t, r_{t+1}, s_{t+1})$ to Critic Network, train Critic Network like DQN. Let Critic Network evaluate better $V_\theta(s, a)$.
- the difference between DQN and Critic Network is, DQN outputs discrete $Q[s][a]$, we need a max operation to find $\max_a Q(s)$. But Critic Network outputs a with $\text{argmax}_a\, Q(s)$ directly.
- more explanation with codes

```
# DQN use torch.max to find max Q
Q_next = target_net(s_next).detach()
Q_target = r + gamma * torch.max(Q_next, 1)[0].view(-1, 1)

# DDPG use ActorNetwork to find a w.r.t max Q,
# put a back to CriticNetwork to calculate q(s,a)
actions_next = actor_target(next_states)
Q_targets_next = critic_target(next_states, actions_next)
# Compute Q targets for current states (y_i)
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
```

- Train ActorNetwork
- use output of CriticNetwork $V_\theta(s, a)$ as baseline or target
- just use the basic gradient ascent to optimize

$$\min_{\theta_{\text{actor}}} |\,\text{actor}(s) - V(s, a)|$$

- let the output of ActorNetwork closer to the baseline from CriticNetwork
- Which is $a_{\text{actor}} = \text{actor}(s)$ let $Q = \text{critic}(s, a)$ is max $Q(s)$
- Note, Train ActorNetwork and CriticNetwork both with local, target copy just like DQN

2. Project Walk-through

This project was solve by DDPG with batch sampling from 20 agents.

2.1 Dummy test with 2D Arm env

Motivation

- I built a 2D n-bar arm environment to test my algorithms.

- My implementation is based on the orginal 2D 2-bar arm program from Morvan Zhou'post
- I extended it to n-bars arm system, and add some utility functions, such as frame image collector, and GIF export.
- this 2D arm is similar but **not the same** with Unity Reacher
- 2D arm can sense both ego parameters and target ball's distance from states.
- Unity Reacher can only sense ego parameters **as claimed** from state.
  - I can't find a good documented definition for thie Reacher Env.
  - It is a motivation that I made the 2D arm env, which I know clearly about the env parameters.
- So the two 2D n-bar arm learns to **track** the target ball.
- Unity Reacher learns **a gesture**, e.g. rotate with a gesture with some constant movement pattern.
- Tracking task from 2D n-bar arm has more straightforward rewards.
- Gesture learning task has a more indirect/sparse reward.
- The reward system is critical for RL models.
- The tracking task is easier in my opinion.
- an example animation of training 3-bars arm
- an example animation of results

Setup

- With parameters

```
env = ArmEnv(n_bar=3, bar_length=80)
agent = DDPG(state_size=env.state_size,
         action_size=env.action_size,
         lr_actor=0.001,
         lr_critic=0.001,
         gamma=0.9,
         tau=0.01,
         memory_size=30000,
         batch_size=32)
```

Observation

- The tracking task from 2D n-bar arm began to converge with about 20-50 trajectories averagely.

- and converged well with about 500 trajectories averagely.

Replay

- dependency: pyglet, imageio (optional)

```
python continuous_control_2d.py
```

Some Thinking

- for simple task, we could set a large learning rate to accelerate training
- hard copy between local/target network can NOT be too frequent, we need to update local network at least e.g. 5-10 times, then copy to target network.
- my intuitive understanding is that we update target network for every 10 steps of local network can reduce the bias error.

2.2 Implementation of DDPG to solve Unity Reacher

- After solved 2D arm system, the Reacher environment is similar, but with a much large solution space to explore.
- The agent need to find a certain constant dynamic gesture with random exploration.
- Without sensing ability, both reward and states are a bit implicit.
- since the solution space is much hard, we can encounter gradient explode/vanish more probably.
- we can either clip the gradient `python self.critic_optimizer.zero_grad() critic_loss.backward() torch.nn.utils.clip_grad_norm(self.critic_local.parameters(), 1) self.critic_optimizer.step()`
- or add batchnorm between the layer
- my intuitive thinking. clip the critic network is enough, since actor network take output of critic network as baseline.
- In this repository, I used BN solution.

Setup

- With parameters

```
agent = DDPG(state_size=33,
             action_size=4,
             sample_batch_size=20,
             lr_actor=0.0001,
             lr_critic=0.0001,
             gamma=0.99,
             tau=0.01,
```

```
                memory_size=int(1e6),
                batch_size=128)
```

- I used a large memory size, to save almost recent 50 episodes, since I didn't implement Prioritized Experience Replay similar algorithms. For this task this lazy solution is reasonable, sine the target gesture do not change, the later experience is guaranteed better than previous'. When we collect the most recent trajectories, we are indeed improve the quality of the learning experience.

Observation

- converged well from about 20000 trajectories base on lucky.