Proximal Policy Optimization Algorithm Implementation

Nima Tavassoli

August 2019

1 Introduction

In light of the limitations of TRPO, PPO introduced a modified algorithm to the class of the policy gradient methods. The algorithm is built on TRPO concepts in order to strike a balance between easy implementation, sample efficiency, and easy parameter tuning.

Considering the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, TRPO maximizies a surrogate objective:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta\left(a_t|s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t|s_t\right)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right]$$

However, PPO's main objective is:

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

The expectation is taken over a minimum of two terms, the first one is the standard policy gradient objective, and the second one is the clipped version of the first item. The final objective, as a result, is a lower bound on the unclipped objective, and restricts the range that the new policy can vary from the old one.

In this project, PPO is implemented in Python and executed on Mujoco environments to train the locomotion agents. All codes are available here.

2 Implementation

Algorithm 1: Main training loop

Gym and Scaler initialization
while *episode* < *num_episodes* **do**
    trajectories = run_policy (env, policy, scaler, stats ,episodes=batch_size);
    episode += len (trajectories);
    insert_value_estimate (trajectories, val_func);
    insert_disc_sum_rew (trajectories, gamma);
    insert_GAE (trajectories, gamma, lamb);
    obs, act, adv, disc = create_train_set (trajectories);
    policy.update (obs, act, adv);
    val_func.fit (obs, disc);
end

2.1 Train

This file is the start-point of the project, and implements the environment initialization as well as the main training loop. It includes some helper functions which are used in the training section.

2.1.1 run_policy method

- It is responsible for running the policy and collecting the data.

```
def run_policy(env, policy, scaler, stats, episodes, index):
    # Run policy and collect data
    total_steps = 0
    trajectories = []
    for e in range(episodes):
        observations, actions, rewards, unscaled_obs = run_episode(env, policy, scaler)
        total_steps += observations.shape[0]
        trajectory = {'observations': observations,
                      'actions': actions,
                      'rewards': rewards,
                      'unscaled_obs': unscaled_obs}
        trajectories.append(trajectory)
    unscaled = np.concatenate([t['unscaled_obs'] for t in trajectories])
    scaler.update(unscaled)  # update running statistics for scaling observations
    mean_reward = np.mean([t['rewards'].sum() for t in trajectories])
    print("M_Reward", mean_reward)
    stats.episode_rewards[index] = mean_reward

    return trajectories
```

2.1.2 insert_disc_sum_rew method

- Calculates the discounted sum of rewards, and afterwards adds them to all time-steps of all trajectories
- The discounted forward sum of a sequence obtains through a signal filter function.

```
def insert_disc_sum_rew(trajectories, gamma):
    # Inserts discounted sum of rewards to all time steps of all trajectories
    for trajectory in trajectories:
        if gamma < 0.999:  # Don't scale for gamma ~= 1
            rewards = trajectory['rewards'] * (1 - gamma)
        else:
            rewards = trajectory['rewards']
        disc_sum_rew = discount(rewards, gamma)
        trajectory['disc_sum_rew'] = disc_sum_rew
```

2.1.3 insert_value_estimate method

- Adds the estimated value function to all time-steps of all trajectories.

```
def insert_value_estimate(trajectories, val_func):
    # Inserts estimated value to all time-steps trajectories
    for t in trajectories:
        obs = t['observations']
        values = val_func.predict(obs)
        t['values'] = values
```

2.1.4 insert_GAE method

- Calculates the advantage using a temporal difference method.

```
def insert_GAE(trajectories, gamma, lamb):
    for trajectory in trajectories:
        if gamma < 0.999:  # Don't scale for gamma ~= 1
            rewards = trajectory['rewards'] * (1 - gamma)
        else:
            rewards = trajectory['rewards']
        values = trajectory['values']
        # TD
        td = rewards + np.append(gamma * values[1:],0) - values
        advantages = discount(td, gamma * lamb)
        trajectory['advantages'] = advantages
```

2.2 Value Function

Value Function approximated using 3 hidden-layer neural network and is trained based on the current batch + the previous one. All hidden layers use a tanh activation.

- Layer1_size = observation_dimension × 10
- Layer2_size = geometric_mean (Layer1_size, Layer2_size)
- Layer3_size = 5

2.3 Policy

Modeled by a multivariate Gaussian with a diagonal covariance matrix. The distribution approximator is implemented with a 3-layer neural network.

- Layer1_size = observation_dimension × 10
- Layer2_size = geometric_mean (Layer1_size, Layer2_size)
- Layer3_size = action_dimension × 5

2.3.1 _loss_train_op method

- Sets the clipping objective loss function
- clipping_range [0] = clipping_range [1] = 0.2, according to the PPO paper
- 'pg_ratio' is the very $r_t(\theta)$
- ADAM optimizer is used

```
def _loss_train_op(self):
    print('loss with clipping objective')
    pg_ratio = tf.exp(self.logp - self.logp_old)
    clipped_pg_ratio = tf.clip_by_value(pg_ratio, 1 - self.clipping_range[0], 1 + self.clipping_range[1])
    surrogate_loss = tf.minimum(self.advantages* pg_ratio,
                                self.advantages * clipped_pg_ratio)
    self.loss = -tf.reduce_mean(surrogate_loss)
    optimizer = tf.train.AdamOptimizer(self.learning_rate_ph)
    self.train_op = optimizer.minimize(self.loss)
```

2.3.2 update method

- Updates the policy based on observations, actions, and advantages.

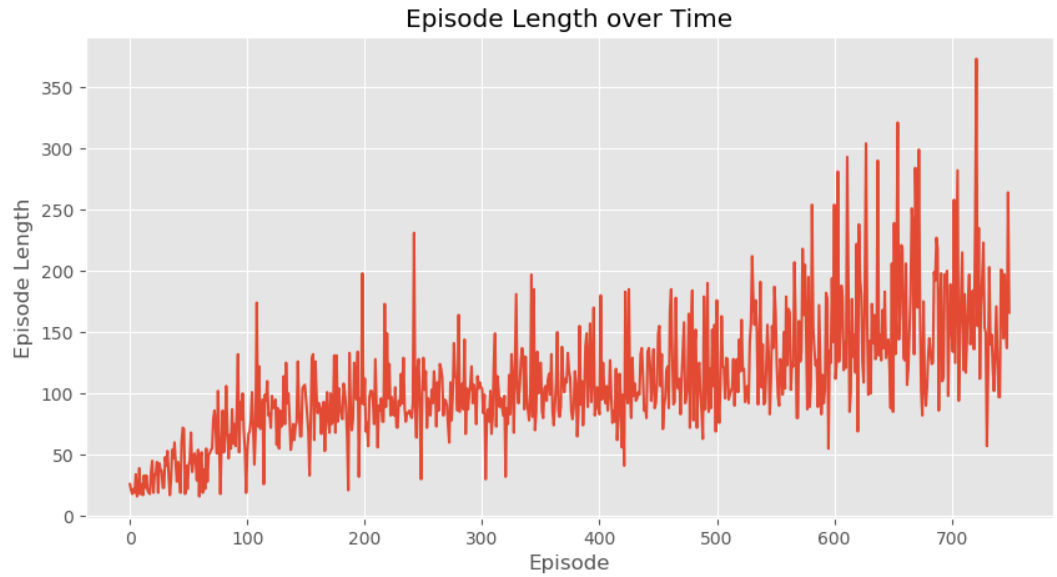
```
def update(self, observes, actions, advantages):
    feed_dict = {self.state: observes,
                 self.action: actions,
                 self.advantages: advantages,
                 self.beta_ph: self.beta,
                 self.eta_ph: self.eta,
                 self.learning_rate_ph : self.learning_rate * self.learning_rate_mult}
    old_means_np, old_log_vars_np = self.sess.run([self.means, self.log_vars],
                                                  feed_dict)
    feed_dict[self.old_log_vars_ph] = old_log_vars_np
    feed_dict[self.old_means_ph] = old_means_np
    loss, kl, entropy = 0, 0, 0
    for e in range(self.epochs):
        self.sess.run(self.train_op, feed_dict)
        loss, kl, entropy = self.sess.run([self.loss, self.kl, self.entropy], feed_dict)
        if kl > self.kl_targ * 4:  # Early stopping if D_KL diverges badly
            break
    if kl > self.kl_targ * 2:
        self.beta = np.minimum(35, 1.5 * self.beta)  # Clip beta
        if self.beta > 30 and self.learning_rate_mult > 0.1:
            self.learning_rate_mult /= 1.5
    elif kl < self.kl_targ / 2:
        self.beta = np.maximum(1 / 35, self.beta / 1.5)  # Clip beta
        if self.beta < (1 / 30) and self.learning_rate_mult < 10:
            self.learning_rate_mult *= 1.5
```

3 Results

I have tested the same algorithm on such Mujoco environments as 'InvertedPendulum-v2' and 'Humanoid-v2'; the improvement, during the training, was noticeable.

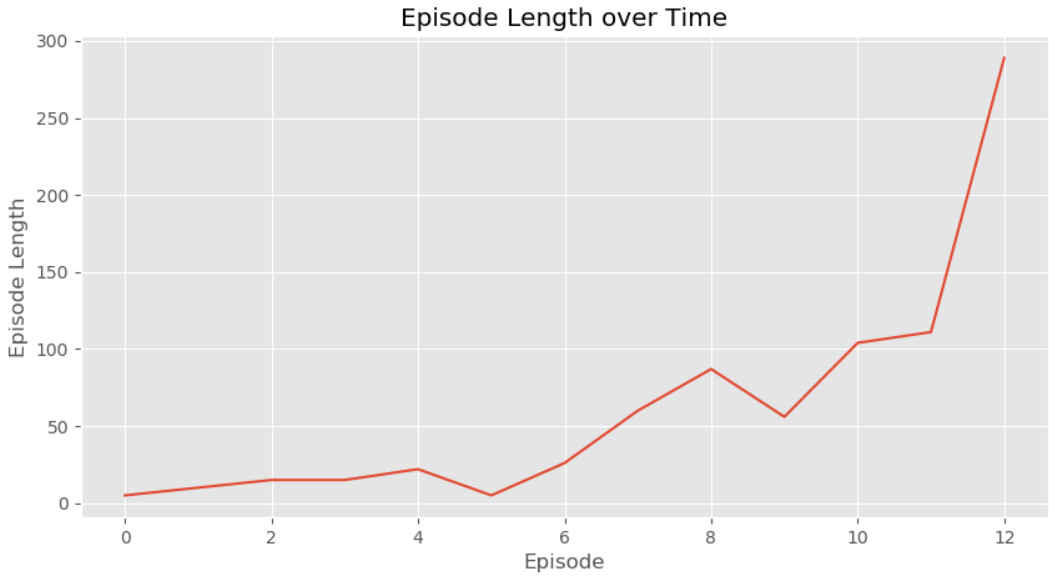
3.1 Humanoid-v2

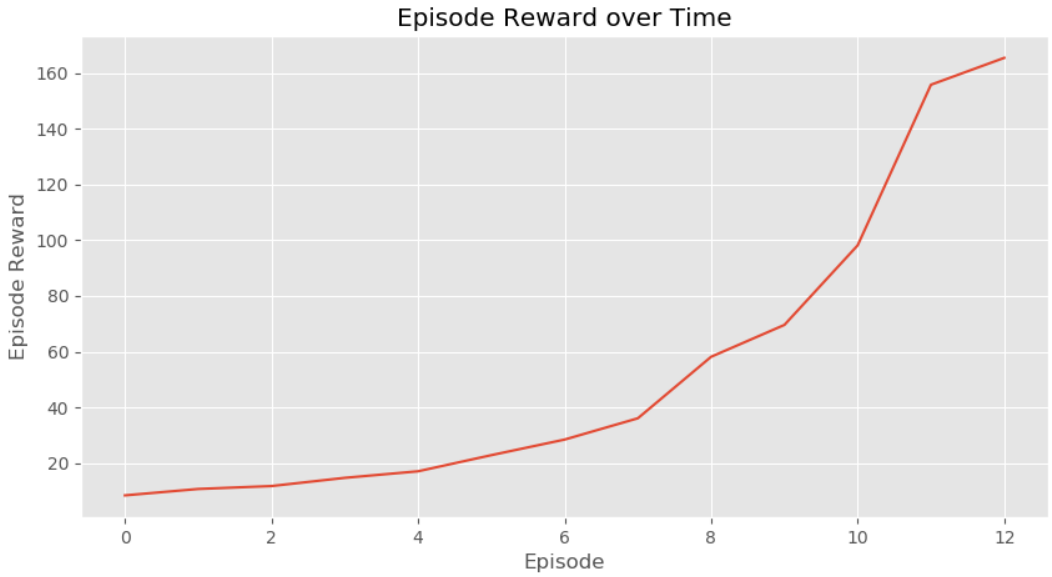
- 15000 episodes
- 750 training batch (20 episodes each)
- Max reward $\sim$ 1000



3.2 InvertedPendulum-v2

- 260 episodes
- 13 training batch (20 episodes each)
- Max reward $\sim$ 165





4 References

- Proximal Policy Optimization (PPO)
- UC Berkeley RL course
- UC Berkeley RL Bootcamp
- Implementation reference