Day 4. Trust Region Policy Optimization & Proximal Policy Optimization

NPEX Reinforcement Learning

July 29, 2021

Jaeuk Shin, Minkyu Park

CORE
Control + Optimization Research Lab

TRPO - Review

TRPO - Review

policy gradient (with importance sampling)

$$\nabla_\phi J(\phi) = \mathbb{E}_{s\sim\rho_{\phi_{\text{old}}},\ a\sim\pi_{\phi_{\text{old}}}}\left(A^{\pi_\phi}(s,a)\frac{\nabla_\phi \pi_\phi(a|s)}{\pi_{\phi_{\text{old}}}(a|s)}\right)$$

To estimate A^{π_ϕ}, we use a separate value function approximator $V(s;\theta)$, and apply **generalized advantage estimation(GAE)**!

Furthermore, we use a stochastic policy $\pi_\phi(a|s)$ (Gaussian in our case).

Bad news: in TRPO, we have a lot of extra stuff to implement...
(Hessian-vector product, line search, etc.)

TRPO - Review

GAE?

Given a trajectory $(s_0, a_0, r_0, s_1, a_1, r_1, \cdots s_T)$ generated by executing the current policy, we first compute

$$\delta_t = r_t + \gamma V(s_{t+1}; \theta) - V(s_t; \theta)$$

Then, GAE is computed as follows:

$$\hat{A}(s_t, a_t) = \sum_{\tau=t}^{T-1} \gamma^{\tau-t} \delta_\tau$$

TRPO - Review

training $V(s;\theta)$?

given a trajectory $(s_0, a_0, r_0, s_1, a_1, r_1, \cdots s_T)$ generated from π_ϕ,
we can compute Monte-Carlo esimtates of $V(s_0), V(s_1), \cdots V(s_T)$
as follows:

$$V(s_t) = \sum_{\tau=t}^{T-1} \gamma^{\tau-t} r_\tau + \gamma^T V(s_T)$$

This is the **target** for value function update!

TRPO - Implementation

TRPO - Implementation

What info do we need when we implement **on-policy algorithms**?

1. (s_j, a_j, s'_j, r_j)
2. generalized advantage estimation(GAE)
3. MC estimates of value $V^{\pi_\phi}(s)$
4. probability $\pi_{\phi_{\text{old}}}(a_j|s_j)$

TRPO - Implementation

```python
self._obs_mem = np.zeros(shape=(lim, dimS))
self._act_mem = np.zeros(shape=(lim, dimA))
self._rew_mem = np.zeros(shape=(lim,))
self._val_mem = np.zeros(shape=(lim,))
self._log_prob_mem = np.zeros(shape=(lim,))
```

collected during agent-env interaction

```python
# memory of cumulative rewards which are MC-estimates of the current value function
self._target_v_mem = np.zeros(shape=(lim,))
# memory of GAE($\lambda$)-estimate of the current advantage function
self._adv_mem = np.zeros(shape=(lim,))
```

computed when sampling a single episode is done

TRPO – Complete Implementation

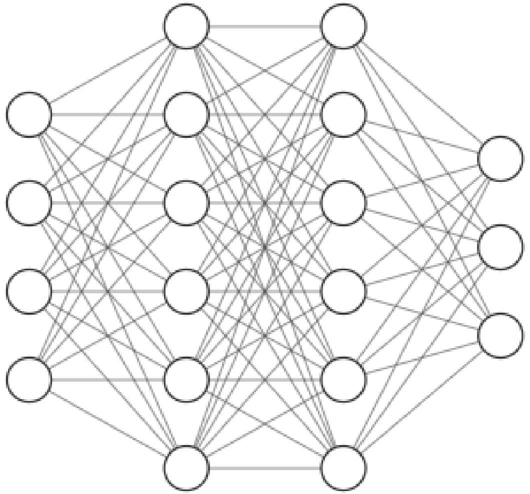
TRPO – Network Architecture

Key components of TRPO agent:

- policy network
  - can handle both discrete/continuous actions
  - network ouput : **probability distribution** over action space (cf. DDPG)

ex) discrete action space

$s \Longrightarrow$



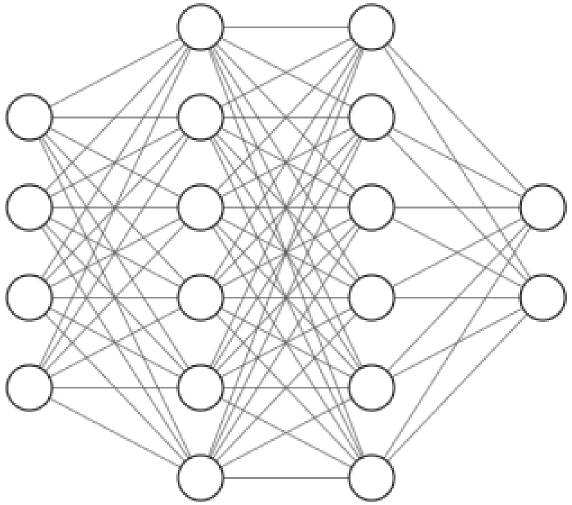
$\pi(a_1|s)$

$\pi(a_2|s)$

$\pi(a_3|s)$

ex) continuous action space

$s \Longrightarrow$



$\mu(s)$

$\sigma(s)$

$a \sim \mathcal{N}(\mu(s), \sigma(s))$

TRPO : Network Architecture

```
class Actor(nn.Module):
    def __init__(self, obs_dim, act_dim, hidden1, hidden2):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(obs_dim, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, act_dim)  # for \mu
        self.fc4 = nn.Linear(hidden2, act_dim)  # for \sigma

    def forward(self, obs):
        x = torch.tanh(self.fc1(obs))
        x = torch.tanh(self.fc2(x))
        mu = self.fc3(x)
        log_sigma = self.fc4(x)
        sigma = torch.exp(log_sigma)
        return mu, sigma

    def log_prob(self, obs, act):
        mu, sigma = self.forward(obs)
        act_distribution = Independent(Normal(mu, sigma), 1)
        log_prob = act_distribution.log_prob(act)
        return log_prob
```

1. Actor Network (By actor we just mean policy)

Why $\log \sigma(s)$ instead of $\sigma(s)$?
$\Longrightarrow$ We have a constraint $\sigma(s) > 0$!

gives $\log \pi(a^i | s^i)$'s for (s^i, a^i) pairs

CORE
Control + Optimization Research Lab

TRPO : Network Architecture

```python
class Critic(nn.Module):
    # critic V(s ; \theta)
    def __init__(self, obs_dim, hidden1, hidden2):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(obs_dim, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, 1)

    def forward(self, obs):
        x = torch.tanh(self.fc1(obs))
        x = torch.tanh(self.fc2(x))

        return self.fc3(x)
```

2. Critic Network : super-easy!

TRPO : Action Selection by Agent

```python
class TRPOAgent:
    def __init__(self, obs_dim, act_dim, hidden1=64, hidden2=32):
        ...

    def act(self, obs):
        obs = torch.tensor(obs, dtype=torch.float).to(device)
        with torch.no_grad():
            mu, sigma = self.pi(obs)
            act_distribution = Independent(Normal(mu, sigma), 1)
            action = act_distribution.sample()
            log_prob = act_distribution.log_prob(action)
            val = self.V(obs)
        action = action.cpu().numpy()
        log_prob = log_prob.cpu().numpy()
        val = val.cpu().numpy()

        return action, log_prob, val
```

Gaussian distribution $\mathcal{N}(\mu(s^i), \sigma(s^i))$

$$a^i \sim \pi_\phi(s^i) := \mathcal{N}(\mu(s^i), \sigma(s^i))$$

$$\log \pi_\phi(a^i | s^i)$$

$$V(s^i; \theta)$$

TRPO : Policy & Value Update

Unfortunately, the parameter updates in TRPO...

1. Fisher-vector product

2. conjugate gradient descent

3. backtracking line search

TRPO – Complete Implementation

1. Fisher-vector product

```
def fisher_vector_product(v, actor, obs_batch, cg_damping=1e-2):
    v.detach()
    kl = torch.mean(kl_div(actor=actor, old_actor=actor, obs_batch=obs_batch))
    kl_grads = torch.autograd.grad(kl, actor.parameters(), create_graph=True)
    kl_grad = torch.cat([grad.view(-1) for grad in kl_grads])
    kl_grad_p = torch.sum(kl_grad * v)
    Iv = torch.autograd.grad(kl_grad_p, actor.parameters()) # product of Fisher information I and v
    Iv = flatten(Iv)
    return Iv + v * cg_damping
```

Basically, Fish information matrix is **Hessian** of a function.
$\Longrightarrow$ costly (requires n^2 numbers of 2^{nd}-order differentiation)

However, computation of Ax can easily be done using automatic differentiation: use

$$(\nabla_\theta^2 f)v = \nabla_\theta(\underbrace{\nabla_\theta f \cdot v}_{\text{scalar}}).$$

TRPO – Complete Implementation

2. conjugate gradient method

What is conjugate gradient for?

$\Longrightarrow$ To efficiently solve $As = g$

```
def cg(f_Ax, b, actor, obs_batch, cg_iters=10, residual_tol=1e-10):
    p = b.clone()
    r = b.clone()
    x = torch.zeros_like(b)
    rdotr = r.dot(r)
    for i in range(cg_iters):
        z = f_Ax(p, actor, obs_batch)
        v = rdotr / p.dot(z)
        x += v*p
        r -= v*z
        newrdotr = r.dot(r)
        mu = newrdotr/rdotr
        p = r + mu*p
        rdotr = newrdotr
        if rdotr < residual_tol:
            break
    return x
```

works if computation of Ax is possible for any x (avoids taking A^{-1})

TRPO – Complete Implementation

3. backtracking linesearch

```python
def backtracking_line_search(...):
    backtrac_coef = 1.0
    alpha = 0.5
    beta = 0.5
    flag = False
    expected_improve = (actor_loss_grad * maximal_step).sum(0, keepdim=True)
    for i in range(10):
        new_params = params + backtrac_coef * maximal_step
        update_model(actor, new_params)
        new_actor_loss = surrogate_loss(actor, adv, states, old_policy.detach(), actions)
        loss_improve = new_actor_loss - actor_loss
        expected_improve *= backtrac_coef
        improve_condition = loss_improve / expected_improve
        kl = kl_div(actor=actor, old_actor=old_actor, obs_batch=states)
        kl = kl.mean()
        if kl < max_kl and improve_condition > alpha:
            flag = True
            break
        backtrac_coef *= beta
    if not flag:
        params = flat_params(old_actor)
        update_model(actor, params)
```

$\theta \longleftarrow \theta + \beta s$ where s : search direction found through CG

check if new policy stays within **trust region**

decrease β and repeat the verification

TRPO – Complete Implementation

```python
def train(env, agent, max_iter, gamma=0.99, lr=3e-4, lam=0.95, delta=1e-3, steps_per_epoch=4000, eval_interval=4000):
    ...
    for epoch in range(num_epochs):
        state = env.reset()
        step_count = 0
        ep_reward = 0
        for t in range(steps_per_epoch):
            action, log_prob, v = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            memory.append(state, action, reward, v, log_prob)
            ep_reward += reward
            step_count += 1
            if (step_count == max_ep_len) or (t == steps_per_epoch - 1):
                s_last = torch.tensor(next_state, dtype=torch.float).to(device)
                v_last = agent.V(s_last).item()
                memory.compute_values(v_last)
            elif done:
                v_last = 0.0
                memory.compute_values(v_last)
```

TRPO – Complete Implementation

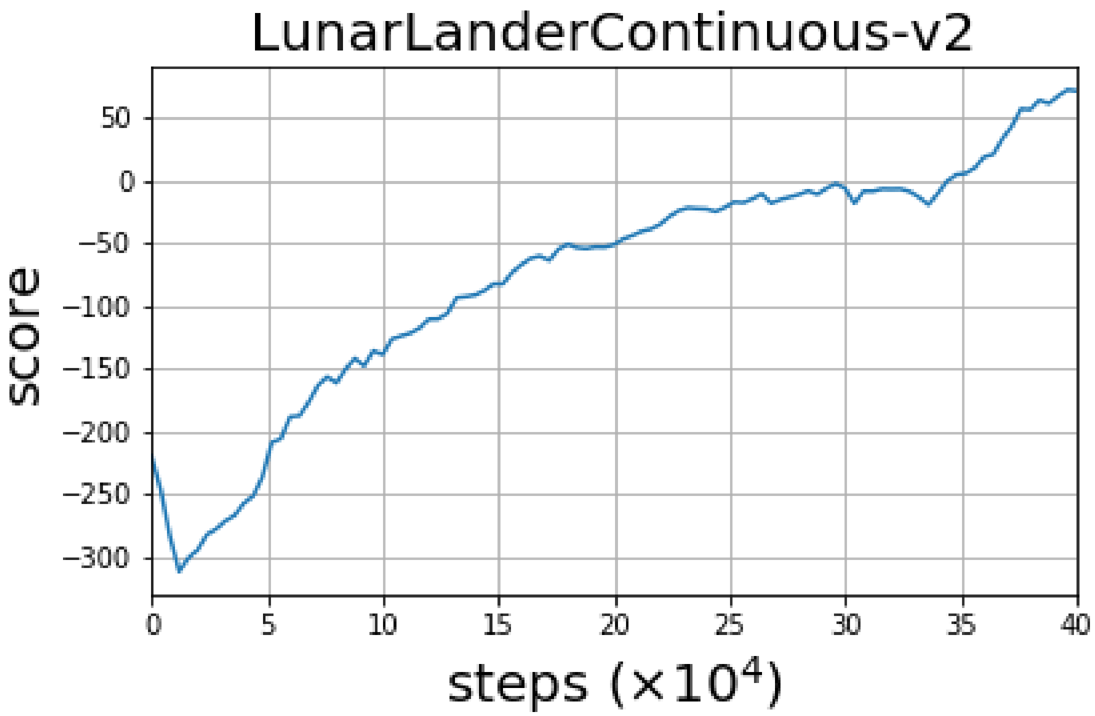
```
            state = next_state
            if done:
                state = env.reset()
                step_count = 0
                ep_reward = 0
            total_t += 1
        update(agent, memory, critic_optim, delta, num_updates=1)
    return
```

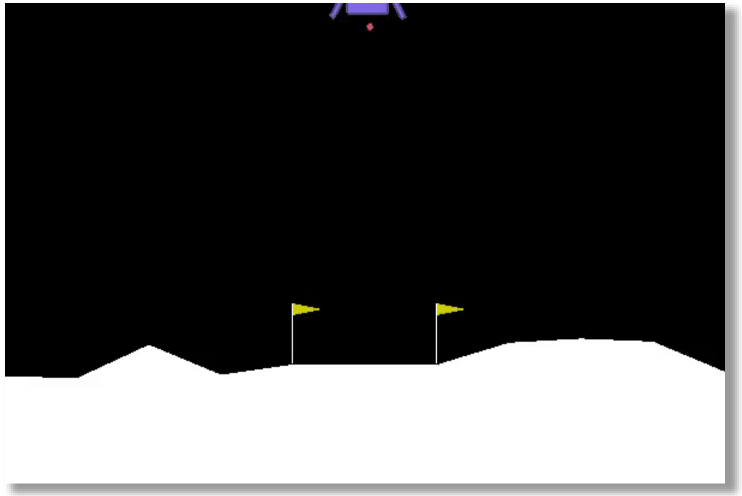
TRPO - Experiment

OpenAI Gym LunarLanderContinuous-v2

Training DDPG on the task was extremely unstable...

What about TRPO?





Control + Optimization Research Lab

Proximal Policy Optimization(PPO) : What's different?

```
for i in range(num_iter):
            log_probs, ent = self.pi.compute_log_prob(states, actions)
            r = torch.exp(log_probs - old_log_probs)
            clipped_r = torch.clamp(r, 1 - self.epsilon, 1 + self.epsilon)
            single_step_obj = torch.min(r * A, clipped_r * A)
            pi_loss = -torch.mean(single_step_obj)
            v = self.V(states)
            V_loss = torch.mean((v - target_v) ** 2)
            ent_bonus = torch.mean(ent)
            loss = pi_loss + 0.5 * V_loss - 0.01 * ent_bonus
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
            if i == num_iter - 1:
                kl = torch.mean(old_log_probs - log_probs).item()
                if kl > 1.5 * self.kl_threshold:
                    break
        return
```

simple, surrogate loss function for policy network parameters!

$\Longrightarrow 1^{st}$-order optimization (no Hessian needed)