Lecture 14: Actor Critic Methods

1 Introduction

Admin:

- Sit in front of class. If there are 2 consecutive empty seats in a row in front of you, move to one of those seats. Repeat.
- Project proposal due yesterday. Check-in with me/TAs due on Mar 31. We'll get feedback on projects posted in the next couple days.
- If you are thinking of trying to publish your course project (yes, this happened last year), come talk to me during office hours. Can help scope project to maximize likelihood of this.

1.1 Review: PPO

Throughout this course we've been going back and forth between two key objects: the policy and the value function. We started with the policy: imitation learning, REINFORCE. We then turned to the value function to think about estimating the policy gradient with lower variance. We're now in the part of the semester where we get to combine these two methods into *actor-critic* methods.

PPO Trick 1: clipped surrogate objective

$$\mathbb{E}_{\rho^{\pi_{\theta_{\text{old}}}}(s_t)\pi_{\theta_{\text{old}}}(a_t|s_t)}\left[\min(r(s_t,a_t;\theta)A(s_t,a_t), \text{clip}(r(s_t,a_t;\theta),1-\epsilon,1+\epsilon)A(s_t,a_t))\log\pi_\theta(a_t \mid s_t)\right]. \tag{1}$$

PPO Trick 2: generalized advantage estimation We will learn a value function by regression to the empirical returns:

$$\min_\theta \frac{1}{2}(V_\theta(s_t) - \sum_{t'=t}^{\infty}\gamma^{t'-t}r_{t'})^2. \tag{2}$$

Then estimate the advantages with GAE:

$$A(s_t,a_t) = -V_t + r_t + (1-\lambda)\gamma V_{t+1} + (1-\lambda)(\underbrace{\lambda+\lambda^2+\cdots}_{=\frac{\lambda}{1-\lambda}})\gamma r_{t+1} + \lambda(1-\lambda)V_{t+2}\gamma^2 + (1-\lambda)(\underbrace{\lambda^2+\lambda^3+\cdots}_{=\frac{\lambda^2}{1-\lambda}}\gamma^2 r_{t+2} + \cdots \tag{3}$$

$$= \sum_{t'=t}^{\infty}(\lambda\gamma)^{t'-t}(r_{t'} + \gamma V_{t'+1} - V_{t'}). \tag{4}$$

2 Activity: PPO Implementation

Form groups of 4. Together with the person next to you, spend the next 10 min talking over HW5 and trying to complete as much as possible. I'll come around and answer questions.

3 Activity: Deciphering DDPG [1]

In the same groups of 4, read through this code and answer the questions. I'll randomly call on groups.

4 Connection with Amortized Optimization

Seen in another way, again thinking about amortized optimization.

Let's start by reviewing the policy improvement step, this time considering a deterministic policy: $\pi : s \mapsto a$. The loss is then

$$\max_\theta \mathbb{E}_s[Q(s, a = \pi_\theta(s))]. \tag{5}$$

The gradient w.r.t. the policy parameters θ are

$$\nabla_\theta = \mathbb{E}_s \left[\nabla_a Q(s, a) \nabla_\theta \pi(s)\right]. \tag{6}$$

This is closely related to the *deterministic policy gradient*, but we'll ignore that for now. Note that we can't use the regular $\log \pi(a \mid s)$ trick for deterministic policies as the probability is either 0 or infinity for deterministic policies in continuous spaces.

Now, I want to consider an alternative approach to policy optimization, what you might do if you hadn't taken this course. In regular supervised learning, we have (x, y) pairs and we learn to map one to the other. For the purpose of policy improvement, we ideally would have examples $(s, a^* = \arg\max_a Q(s, a))$.

However, actually computing these optimal actions a^* can be quite expensive. One way to do this would be to run gradient descent on the Q-function. But, this is expensive; it requires many steps to converge. So, we could consider a heuristic – what if we just take one gradient update? That is, we pretend that $a^* = \pi(s) + \nabla_a Q(s, a)$. Then, let's do supervised learning on these data, using a MSE:

$$\min_\theta \tfrac{1}{2} \mathbb{E}_{(s,a^*)} \left[(\pi_\theta(s) - a^*)^2\right]. \tag{7}$$

Let's take the gradient of this, substituting our approximation for a^*:

$$\nabla_\theta \tfrac{1}{2} \mathbb{E}_{(s,a^*)} \left[(\pi(s) - a^*)^2\right] = \mathbb{E}_{(s,a^*)} \left[(\pi(s) - a^*) \nabla_\theta \pi(s)\right] \tag{8}$$

$$= \mathbb{E}_s \left[(\cancel{\pi(s)} - (\cancel{\pi(s)} + \nabla_a Q(s, a))) \nabla_\theta \pi(s)\right] \tag{9}$$

$$= -\mathbb{E}_s \left[\nabla_a Q(s, a) \nabla_\theta \pi(s)\right] \tag{10}$$

Note that this is exactly the same as the deterministic policy gradient! The sign is different just because the objective in Eq. 5 is a maximization problem while Eq. 7 is a minimization problem.

5 Thinking in terms of code.

Neural networks:

- Actor (policy)
- Critic (Q-function)

Modern deep learning libraries are magical, making it very easy to compute all sorts of policy improvement losses.

Policy gradient. Let's start with the regular policy gradient. We'd like to compute the gradient $\mathbb{E}_\pi[Q(s, a) \nabla_\theta \pi(a \mid s)]$. But, it's often easier to think in terms of losses than in terms of gradients; doing so let's us feed the loss function to some optimizer (e.g., Adam, SGD, Adagrad) and let it do the optimization from there. So, how do we construct a loss function whose gradient is the policy gradient?

To do this, we can simply sample states and actions from the policy π, and then do Q-weighted behavioral cloning

```
s = env.reset()
loss = 0
```

```
while not done:
  a = policy(a).sample().detach()
  loss -= Q(s, a) * pi(s).log_prob(a)
  s, r, done, _ = env.step(a)
policy_gradient = torch.grad(loss, policy.parameters)
```

We have previously talked about the connections between behavioral cloning and the policy gradient. In particular, we can think about the policy gradient as doing behavioral cloning *on the policy itself*, except that each sample is weighted by the Q-function. Said in other words, the policy gradient is equivalent to imitating optimized data. We now see that these are not only useful mental models for understanding the mechanics of the the policy gradient, but they also directly correspond to how you'd implement these methods!

Policy improvement. In turns out we can go a step further, and write down this loss in an even simpler form by exploiting the fact that modern deep learning libraries do automatic differentiation.

```
def actor_loss(s):
  a = policy(s).sample()
  return -Q(s, a)
```

Mathematically, this loss is the same as the one above. Under the hood, the deep learning library will compute the correct gradients. However, this loss can actually work better in practice because of the reparametrization trick. Let's assume that $\pi(a \mid s)$ is a Gaussian. Sampling from that Gaussian is typically implemented as

```
def gaussian_sample(loc, scale):
  std = standard_normal.sample()
  return loc + std * scale
```

Note that this function is differentiable w.r.t. the location and scale. This is what's happening under the hood in many deep learning libraries.

6 Practical Actor-Critic Algorithms

There are many practical and widely-used algorithms that are instantiations of this general recipe:

- A2C
- A3C
- DDPG
- SVG(0)
- SAC
- NAF
- D4PG

These are good papers to refer to to learn more.

In the latter half of the course, we'll see that these algorithms differ in some design decisions that end up being important:

- Replay buffers – keeping around old and new data, doing both value and policy udpates on both. Note that this is easier than doing importance weighting.
- Min Q trick – avoid OOD actions

References

[1] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

```python
class Args:
  ...
  # Algorithm specific arguments
  env_id: str = "Hopper-v4"
  """the environment id of the Atari game"""
  total_timesteps: int = 1000000
  """total timesteps of the experiments"""
  learning_rate: float = 3e-4
  """the learning rate of the optimizer"""
  buffer_size: int = int(1e6)
  """the replay memory buffer size"""
  gamma: float = 0.99
  """the discount factor gamma"""
  tau: float = 0.005
  """target smoothing coefficient (default: 0.005)"""
  batch_size: int = 256
  """the batch size of sample from the reply memory"""
  exploration_noise: float = 0.1
  """the scale of exploration noise"""
  learning_starts: int = 25e3
  """timestep to start learning"""
  policy_frequency: int = 2
  """the frequency of training policy (delayed)"""
  noise_clip: float = 0.5
  """noise clip parameter of the Target Policy Smoothing Regularization"""


# ALGO LOGIC: initialize agent here:
class QNetwork(nn.Module):
  def __init__(self, env):
    super().__init__()
    self.fc1 = nn.Linear(np.prod(env.action_space.shape), 256)
    self.fc2 = nn.Linear(256, 256)
    self.fc3 = nn.Linear(256, 1)

  def forward(self, x, a):
    x = torch.cat([x, a], 1)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x


class Actor(nn.Module):
  def __init__(self, env):
    super().__init__()
    self.fc1 = nn.Linear(np.array(env.observation_space.shape).prod(), 256)
    self.fc2 = nn.Linear(256, 256)
    self.fc_mu = nn.Linear(256, np.prod(env.action_space.shape))
    # action rescaling
    self.action_scale = torch.tensor((env.action_space.high - env.action_space.low) / 2.0)
    self.action_bias = torch.tensor((env.action_space.high + env.action_space.low) / 2.0)

  def forward(self, x):
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = torch.tanh(self.fc_mu(x))
    return x * self.action_scale + self.action_bias


if __name__ == "__main__":
  ...
  random.seed(args.seed)
  np.random.seed(args.seed)
  torch.manual_seed(args.seed)
```

```
device = torch.device("cuda" if torch.cuda.is_available() and args.cuda else "cpu")

# env setup
envs = make_env(args.env_id, args.seed)

actor = Actor(envs).to(device)
qf1 = QNetwork(envs).to(device)
qf1_target = QNetwork(envs).to(device)
target_actor = Actor(envs).to(device)
...
q_optimizer = optim.Adam(list(qf1.parameters()), lr=args.learning_rate)
actor_optimizer = optim.Adam(list(actor.parameters()), lr=args.learning_rate)

rb = ReplayBuffer(
    args.buffer_size,
    envs.observation_space,
    envs.action_space,
)

obs, _ = envs.reset(seed=args.seed)
for global_step in range(args.total_timesteps):
  if global_step < args.learning_starts:
    actions = np.array([envs.action_space.sample() for _ in range(envs.num_envs)])
  else:
    with torch.no_grad():
      actions = actor(torch.Tensor(obs).to(device))
      actions += torch.normal(0, actor.action_scale * args.exploration_noise)
      actions = actions.cpu().numpy().clip(envs.action_space.low, envs.action_space.high)

  next_obs, rewards, terminations, truncations, infos = envs.step(actions)
  ...
  rb.add(obs, next_obs, actions, rewards, terminations, infos)
  obs = next_obs

  # ALGO LOGIC: training.
  if global_step > args.learning_starts:
    data = rb.sample(args.batch_size)
    with torch.no_grad():
      next_state_actions = target_actor(data.next_observations)
      qf1_next_target = qf1_target(data.next_observations, next_state_actions)
      next_q_value = data.rewards.flatten() + (1 - data.dones.flatten()) * args.gamma * (qf1_next_target)

    qf1_a_values = qf1(data.observations, data.actions)
    qf1_loss = F.mse_loss(qf1_a_values, next_q_value)

    # optimize the model
    q_optimizer.zero_grad()
    qf1_loss.backward()
    q_optimizer.step()

    ...
    actor_loss = -qf1(data.observations, actor(data.observations)).mean()
    actor_optimizer.zero_grad()
    actor_loss.backward()
    actor_optimizer.step()

    # update the target network
    for param, target_param in zip(actor.parameters(), target_actor.parameters()):
      target_param.data.copy_(args.tau * param.data + (1 - args.tau) * target_param.data)
    for param, target_param in zip(qf1.parameters(), qf1_target.parameters()):
      target_param.data.copy_(args.tau * param.data + (1 - args.tau) * target_param.data)
  ...
```

7 Group Problems

1. When would you prefer this method over DQN, and vice versa?

2. When would you prefer this method over PPO, and vice versa?

3. What is the loss function for the actor?

4. Compute (with calculus) the gradient of the actor objective.

   (a) How is this related to the policy gradient?

5. What is the loss function for the critic?

   (a) Is the critic estimating $Q^\star$ (like DQN), Q^{π_t} (like expected SARSA), or $Q^{\pi_{\text{old}}}$ (like SARSA and MC regression)?

6. What does the target actor do?

7. How would you modify this code to implement something akin to double Q-learning?