Lecture 15: Advanced Actor Critic Methods

1 Introduction

Logistics:

- HW5 (PPO): Due tomorrow.
- HW6 (TD3): released, will implement TD3 (which we'll talk about today), will be due a week from tomorrow.
- HW7 (SAC): released next week
- HW8: canceled? HWs will account for same fraction of grade.
- Final project: peer feedback and instructor check-in due on Mon.

1.1 Review of DDPG [2]

Actor loss:

$$\max_{\phi} \mathbb{E}_s[Q(s, a = \pi_\phi(s))]. \tag{1}$$

Critic loss:

$$\max_{\theta} \mathbb{E}_{\substack{(s,a,r,s')\sim\mathcal{B},\\ a'\sim\pi_{\text{target}}(a'|s')}} \left[\left(Q_\theta(s,a) - (r + \gamma Q_{\text{target}}(s', a'))\right)^2 \right]. \tag{2}$$

- Lifts key limitation of DQN: can work with continuous actions.
- Estimates the Q value of π_{target}, not $Q^\star$ or Q^β.
- Increased training stability using target Q network and target policy.

Looking forward Spectrum of how "off-policy" a method is:

- REINFORCE
- DDPG (and TD3, SVG(0), SAC)
- DQN
- PPO
- Offline RL (next week)

Next week will wrap up this penultimate chapter of the course by looking at MaxEnt RL (SAC) and offline RL:

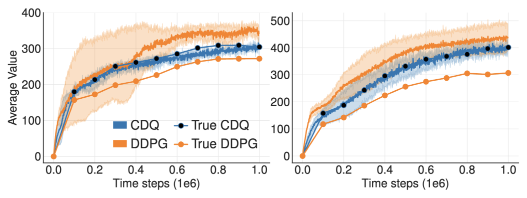
1. What is RL?
2. Do you really need RL? (bandits, MPC, imitation)
3. The policy gradient
4. Value functions
5. Actor-critic methods
6. Where do rewards come from?

1.2 Learning objectives.

By the end of today's class, you'll be able to implement some of the most widely-used off-policy, actor-critic RL algorithms (DDPG, TD3, SVG). You'll be able to explain why certain designs are made in these algorithms.

2 TD3 [1]

One of the most widely used variants of DDPG today is TD3. The TD3 paper was published in 2018, 3 years after the DDPG paper. The main challenge that TD3 sought to address was critic overestimation: the Q-values predicted more returns than the agent actually received.

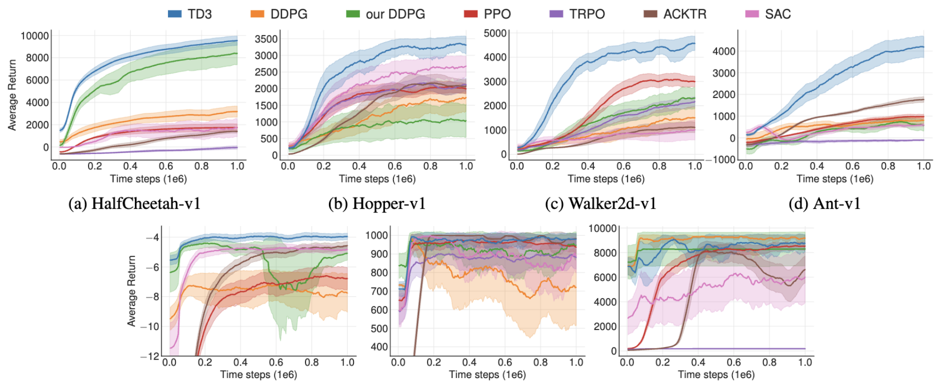


(a) Hopper-v1 (b) Walker2d-v1

Figure 1. Measuring overestimation bias in the value estimates of DDPG and our proposed method, Clipped Double Q-learning (CDQ), on MuJoCo environments over 1 million time steps.

Figure 1: Figure from the TD3 paper showing how DDPG overestimates the future returns

By fixing this issue (and we'll see how in today's class), TD3 can achieve significantly higher returns:



(a) HalfCheetah-v1 (b) Hopper-v1 (c) Walker2d-v1 (d) Ant-v1

(e) Reacher-v1 (f) InvertedPendulum-v1 (g) InvertedDoublePendulum-v1

Figure 2: Figure from the TD3 paper showing how TD3 outperforms prior methods.

It did this by adding a few tricks:

1. additive clipped noise on actions
2. double critics and actors
3. delayed actors update

Exercise: form groups of about 4, which are wholly distinct from your groups on Tuesday. We'll spend 40 – 60 min working on the problems in groups, and then reconvene to go over the answers together.

References

[1] Fujimoto, S., Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR.

[2] 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*.

```
class Args:
  env_id: str = "Hopper-v4"
  """the id of the environment"""
  total_timesteps: int = 1000000
  """total timesteps of the experiments"""
  learning_rate: float = 3e-4
  """the learning rate of the optimizer"""
  num_envs: int = 1
  """the number of parallel game environments"""
  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"""
  policy_noise: float = 0.2
  """the scale of policy noise"""
  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"""

class QNetwork(nn.Module):
  def __init__(self, env):
    super().__init__()
    self.fc1 = nn.Linear(
      np.array(env.observation_space.shape).prod() + 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))
    self.action_scale = (env.action_space.high - env.action_space.low)
    self.action_bias = (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__":
  args = tyro.cli(Args)
  run_name = f"{args.env_id}__{args.exp_name}__{args.seed}__{int(time.time())}"
```

```
random.seed(args.seed)
...

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

actor = Actor(envs).to(device)
qf1 = QNetwork(envs).to(device)
qf2 = QNetwork(envs).to(device)
qf1_target = QNetwork(envs).to(device)
qf2_target = QNetwork(envs).to(device)
target_actor = Actor(envs).to(device)
target_actor.load_state_dict(actor.state_dict())
qf1_target.load_state_dict(qf1.state_dict())
qf2_target.load_state_dict(qf2.state_dict())
q_optimizer = optim.Adam(list(qf1.parameters()) + list(qf2.parameters()), lr=args.learning_rate)
actor_optimizer = optim.Adam(list(actor.parameters()), lr=args.learning_rate)

envs.observation_space.dtype = np.float32
rb = ReplayBuffer(
  args.buffer_size,
  envs.observation_space,
  envs.action_space,
)
start_time = time.time()

# TRY NOT TO MODIFY: start the game
obs, _ = envs.reset(seed=args.seed)
for global_step in range(args.total_timesteps):
  # ALGO LOGIC: put action logic here
  if global_step < args.learning_starts:
    actions = envs.action_space.sample()
  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

  if global_step > args.learning_starts:
    data = rb.sample(args.batch_size)
    with torch.no_grad():
      clipped_noise = (torch.randn_like(data.actions, device=device) * args.policy_noise).clamp(
        -args.noise_clip, args.noise_clip
      ) * target_actor.action_scale

      next_state_actions = (target_actor(data.next_observations) + clipped_noise).clamp(
        envs.action_space.low[0], envs.action_space.high[0]
      )
      qf1_next_target = qf1_target(data.next_observations, next_state_actions)
      qf2_next_target = qf2_target(data.next_observations, next_state_actions)
      min_qf_next_target = torch.min(qf1_next_target, qf2_next_target)
      next_q_value = data.rewards.flatten() + (1 - data.dones.flatten()) * args.gamma * (min_qf_next_target)

    qf1_a_values = qf1(data.observations, data.actions).view(-1)
    qf2_a_values = qf2(data.observations, data.actions).view(-1)
    qf1_loss = F.mse_loss(qf1_a_values, next_q_value)
    qf2_loss = F.mse_loss(qf2_a_values, next_q_value)
    qf_loss = qf1_loss + qf2_loss

    # optimize the model
```

```
q_optimizer.zero_grad()
qf_loss.backward()
q_optimizer.step()

if global_step % args.policy_frequency == 0:
  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)
  for param, target_param in zip(qf2.parameters(), qf2_target.parameters()):
    target_param.data.copy_(args.tau * param.data + (1 - args.tau) * target_param.data)
```

3 Group Problems

1. How many Q networks are there in total? What is the function of each Q network?

2. What is the loss function for the critic?

3. Compared with DDPG, why would this critic loss for TD3 help prevent critic overestimation?

4. Is the actor or critic updated more frequently?

5. Is the actor stochastic or deterministic?

6. What is the loss function for the actor?

7. Let's say that you directly wanted to learn a stochastic actor, which predicts a mean and standard deviation for a Gaussian distribution.
   (a) Modify the code to implement your stochastic actor and its associated loss function.

   (b) In math, what have you used as the actor objective for this stochastic policy?

   (c) In math, what is the gradient of this objective?

   (d) Is PyTorch computing the same gradient under the hood?