DS 598
Introduction to RL

Xuezhou Zhang

Announcements

- Homework 2 will be out this week and due next Friday.
- Office hour changing from Thursday to Friday 1:00-2:00PM.
- Please go to the discussion sections.

Chapter 4: Value-based RL (Continued)

Recap from last time

- When using function approximation,
  1) Value-based RL can converge to the wrong solution (Bellman-completeness)
  2) Value-based RL may not even converge (Q-learning)

- Heuristic methods to combat divergence
  1) Target network
  2) Double Q-learning
  3) Replay Buffer
  4) Multi-step Return

A quick overview for Deep RL coding

- Environment
- Agent

Environment

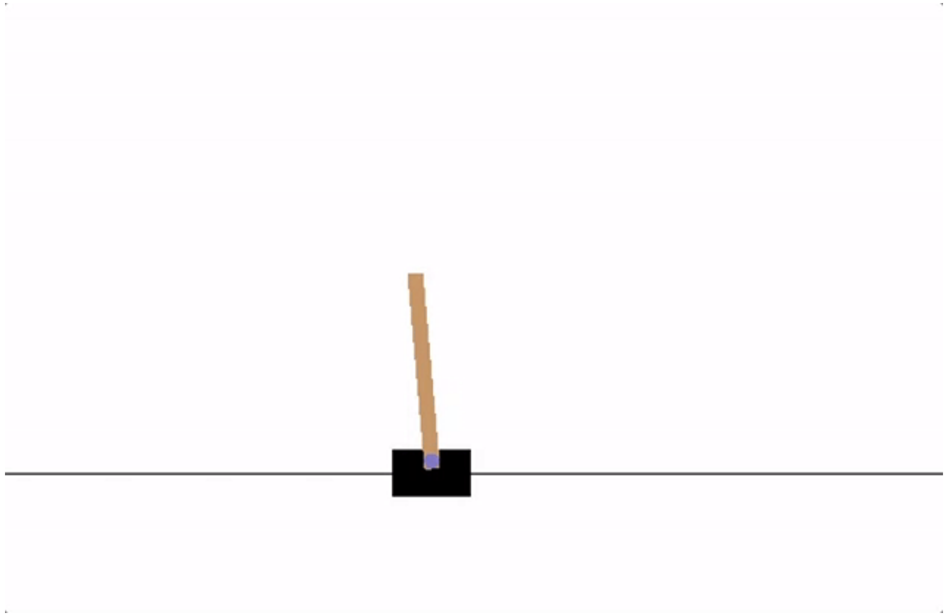
- **The OpenAI Gym format:**

```
class YourEnv(gym.Env):
        def __init__(self, parameters):
                ## Set parameters for the environment.
        def step(self, action):
                self.state, reward, terminate = transition(self.state, action)
                return self.state, reward, terminate
        def reset(self):
                self.state = sample_initial_state()
                return self.state
```

Environment

- Common benchmark environments are already included, e.g.

- **env = gym.make(**"CartPole-v1"**)**



Agent

- Replay buffer
- Network
- Training

Replay Buffer

```
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))


class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Network

```python
class DQN(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)
```

Training (DQN)

- Initializations

```
policy_net = DQN(n_observations, n_actions).to(device)
target_net = DQN(n_observations, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())

optimizer = optim.AdamW(policy_net.parameters(), lr=LR, amsgrad=True)
memory = ReplayMemory(10000)
```

Training (DQN)

- Sample from replay buffer

"classic" deep Q-learning algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update ϕ': copy ϕ every N steps

```
state_action_values = policy_net(state_batch).gather(1, action_batch)
```

```
transitions = memory.sample(BATCH_SIZE)
batch = Transition(*zip(*transitions))
state_batch = torch.cat(batch.state)
action_batch = torch.cat(batch.action)
reward_batch = torch.cat(batch.reward)
```

Training (DQN)

- Updating the Q-network

"classic" deep Q-learning algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update ϕ': copy ϕ every N steps

```
next_state_values = torch.zeros(BATCH_SIZE, device=device)
with torch.no_grad():
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1).values
# Compute the expected Q values
expected_state_action_values = (next_state_values * GAMMA) + reward_batch

# Compute Huber loss
criterion = nn.SmoothL1Loss()
loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

# Optimize the model
optimizer.zero_grad()
loss.backward()
# In-place gradient clipping
torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
optimizer.step()
```

Training (DQN)

- Updating the target network

"classic" deep Q-learning algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update ϕ': copy ϕ every N steps

```
        # Soft update of the target network's weights
        # θ′ ← τ θ + (1 −τ )θ′
        target_net_state_dict = target_net.state_dict()
        policy_net_state_dict = policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key]*TAU + 
target_net_state_dict[key]*(1-TAU)
        target_net.load_state_dict(target_net_state_dict)
```

DQN

- Jupyter Notebook/Colab example at https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Chapter 5: Policy-based RL

Policy-based RL

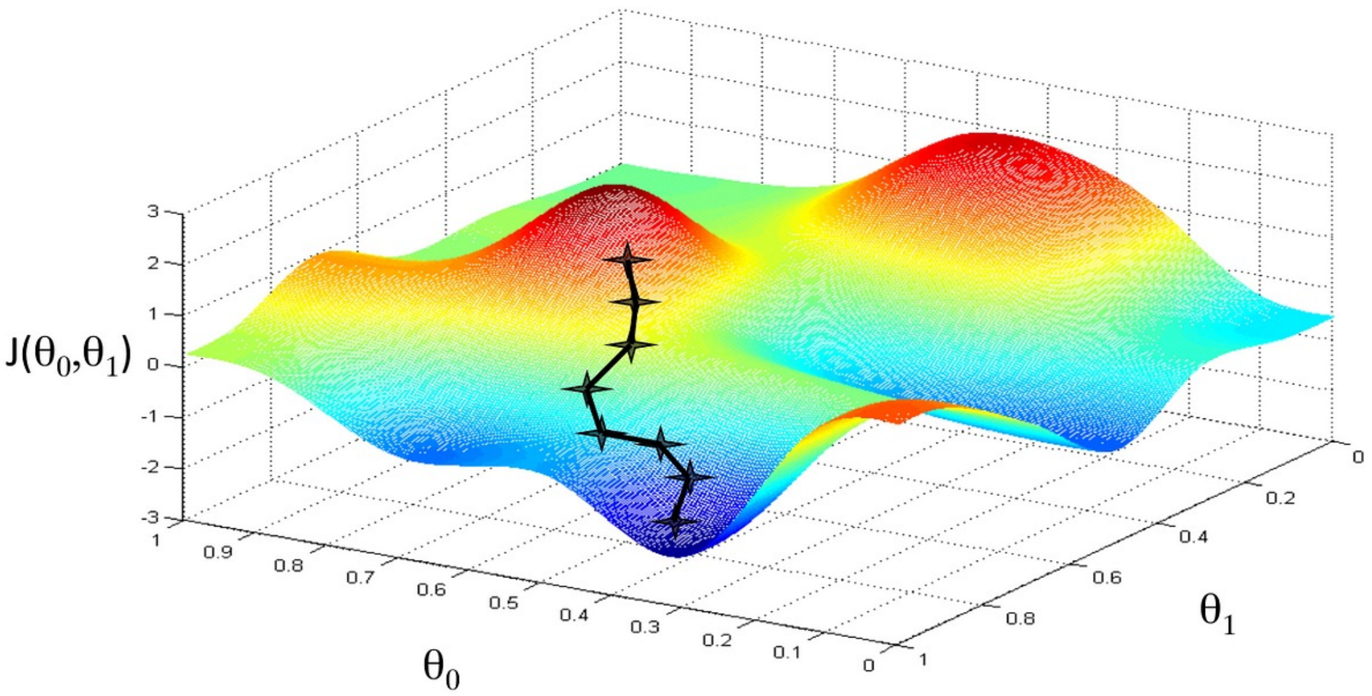
- Let $\tau = (s_0, a_0, s_1, a_1, \ldots)$ denotes the trajectory, the goal of RL is to maximize

$$J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi}[R_\theta(\tau)]$$

Policy-based RL

- Why not just treat it as any other optimization problem and run **gradient ascent**?

$$\theta_{t+1} = \theta_t + \alpha_t \nabla_\theta J\left(\pi_{\theta_t}\right)$$



Policy Gradient

- Parameterized Policy $\pi_\theta(a|s) = \pi_\theta(a|s;\theta)$.

- **Core question**: How to compute $\nabla_\theta J(\pi_\theta)$, where $J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[\sum_{h=0}^{\infty} \gamma^h r_h]$?

Policy Gradient

- A general change of measure trick

$$\nabla_\theta \mathbb{E}_{x \sim p_\theta(\cdot)}\left[f(x)\right] = \nabla_\theta \int_x p_\theta(x) f(x) dx$$

Policy Gradient

$$\nabla_\theta \mathbb{E}_{x\sim p_\theta(\cdot)}\left[f(x)\right] = \mathbb{E}_{x\sim p_\theta(\cdot)}\left[\nabla_\theta \log p_\theta(x) f(x)\right]$$

- Applying to our problem,

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log p_\theta(\tau) R(\tau)\right]$$

The Policy Gradient Theorem

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{h=0}^{\infty} \nabla_\theta \log \pi(a_h | s_h) R(\tau)\right]$$

- Estimate PG from sample trajectories $\tau_1, \ldots, \tau_n \sim \pi_\theta$

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{n} \sum_{i=1}^{n} \left[\sum_{h=0}^{\infty} \nabla_\theta \log \pi(a_{i;h} | s_{i;h}) R(\tau_i)\right]$$

The REINFORCE algorithm

1. Initialize θ_0
2. For iteration t = 0,…,T
   1) Run π_{θ_t} and collect trajectories $\tau_1, \dots, \tau_n$
   2) Estimate the PG by

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \left[\sum_{h=0}^{\infty} \nabla_\theta \log \pi(a_{i;h} | s_{i;h}) R(\tau_i) \right]$$

   3) Do SGD update $\theta_{t+1} = \theta_t + \alpha_t g_t$

The REINFORCE algorithm

- REINFORCE is an “on-policy” algorithm
- i.e. it only uses data collected by π_t to update π_t.

- Not using of historical data $\Rightarrow$ sample inefficient!
- However, REINFORCE is stable
- i.e. always converge to a local optimal solution.

Pytorch Implementation Snippet

- Save data as trajectories instead of individual transitions in the replay buffer.

```
for log_prob, R in zip(policy.saved_log_probs, returns):
    policy_loss.append(-log_prob * R)
optimizer.zero_grad()
policy_loss = torch.cat(policy_loss).sum()
policy_loss.backward()
optimizer.step()
```

The REINFORCE algorithm

- Pros:
  - ✓Convergence
  - ✓Conceptually simple

- Cons:
  - ❖Only works with stochastic policies
  - ❖On-policy -> Sample inefficient
  - ❖High Variance $\mathbb{E}\left[\left|\left|g_t - \nabla_\theta J(\pi_\theta)\right|\right|_2^2\right]$

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{h=0}^{\infty} \nabla_\theta \log \pi(a_h|s_h) R(\tau)\right]$$

The next few lectures…

- Solve each of the cons of REINFORCE.