CS6700
Reinforcement Learning

Programming Assignment 2

Archish S
me20b032@smail.iitm.ac.in

Vinayak Gupta
ee20b152@smail.iitm.ac.in

Indian Institute of Technology, Madras

Saturday 30th March, 2024

Contents

Scripts

The code for this assignment (and other assignments) and output files are logged in this GitHub repository.

1 Introduction

Deep Reinforcement Learning (Deep RL) utilizes Deep Learning to enhance traditional Reinforcement Learning algorithms. The principal idea is to make use of a neural network to parameterize the Q function (and possibly the policy π).
In this assignment, we study two methods of Deep RL: Dueling DQN 3.1 add Monte-Carlo REINFORCE 4.1.

2 Environment Description

We base our study on the CartPole-v1 and the Acrobot-v1 environments of the Open AI's gymnasium library.

2.1 CartPole

A pole is attached by an unactuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart, and the goal is to balance the pole by applying force in the left and right directions.

Action Space

The action can take values $\{0, 1\}$ (0 for left and 1 for right), indicating the direction of the fixed force that the cart is pushed with.

State Space

The observation space consists of the values corresponding to the Cart Position $(-4.8, 4.8)$, the Cart Velocity $(-\infty, \infty)$, the Pole Angle $(-24°, 24°)$ and the Pole Angular Velocity $(-\infty, \infty)$.

Rewards

The goal is to keep the pole upright for as long as possible. A reward of $+1$ is allotted for every step taken.

Termination

The episode terminates if

- Cart Position is not in the range $[-2.4, 2.4]$ (or)
- Pole Angle is not in the range $[-12°, 12°]$ (or)
- Episode length is greater than 200.

2.2 Acrobot

The system consists of two links connected linearly to form a chain, with one end of the chain fixed. The joint between the two links is actuated. The goal is to apply torques on the actuated joint to swing the free end of the linear chain above a given height while starting from the initial state of hanging downwards.

Action Space

The action can take values $\{-1, 0, 1\}$ indicating torque applied on the actuated joint.

State Space

The observation space consists of the values corresponding to the two rotational joint angles, Sine and Cosine of angles θ_1 and θ_2, and the angular velocity of θ_1 and θ_2 in $(-4\pi, 4\pi)$, where θ_1 is the angle of the first joint with the $y-$axis and θ_2 is the relative angle to the first link.

Rewards

The goal is to have the free end reach a designated target. A reward of -1 is allotted for all the steps that do not reach the goal.

Termination

The episode terminates if

- Reaching the target height $-\cos(\theta_1) - \cos(\theta_2 + \theta_1) > 1$ (or)
- Episode length is greater than 500.

3 Deep Q-Learning

This strategy focuses on learning the parameterized Q function by comparing the Q-value predicted with a Q-target and uses gradient descent to update the parameters. The Deep Q-Learning algorithm has two phases

- Perform actions and store the observed experience tuples in a replay buffer,
- Use a randomly selected mini-batch of tuples and update the parameters using gradient descent.

Algorithm 3.1: Deep Q-Learning

Input: discount factor γ; update steps c; replay buffer size N

Initialize: replay memory $\mathcal{D}$ with capacity N; state-action value function Q with random weights Θ; target function $\hat{Q}$ with weights $\hat{\Theta} = \Theta$

for episode $= 1$ to M **do**
  Initialize sequence s_1
  for $t = 1$ to T **do**
    With probability ϵ select a random action a_t otherwise select $a_t = \underset{a \in \mathcal{A}(s_t)}{\arg\max}\, Q(s_t, a; \Theta)$
    Take action a_t, observe reward r_t and next state s_{t+1} and store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random mini-batch $\mathcal{D}_b \subseteq \mathcal{D}$ and compute $\boldsymbol{y}$ as

$$y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max\limits_{a' \in \mathcal{A}(s_{j+1})} \hat{Q}(s_{j+1}, a'; \hat{\Theta}) & \text{otherwise} \end{cases} \quad \forall\, j = \{|\mathcal{D}_b|\}$$

    Perform gradient descent step on $(\boldsymbol{y} - \boldsymbol{Q}(\cdot\,; \Theta))^2$ for $Q_j(\cdot\,; \Theta) = Q(s_j, a_j; \Theta)\, \forall\, j = \{|\mathcal{D}_b|\}$
    if $t \mod c = 0$ **then**
      reset $\hat{\Theta} \leftarrow \Theta$

Output: Parameterized Q function $Q(\cdot\,; \Theta)$

3.1 Dueling DQN

Dueling DQN is an extension of the Deep Q-Learning Algorithm 1, designed to improve learning efficiency by decomposing the Q-value function into two separate streams - one estimating the state value and the other estimating the advantage of each action. The update equation for the dueling network is modified as

$$Q(s, a; \Theta) \doteq V(s; \Theta) + \left(A(s, a; \Theta) - \frac{1}{|\mathcal{A}(s)|} \sum_{a' \in \mathcal{A}(s)} A(s, a'; \Theta) \right) \qquad \text{(Type 1)}$$

$$Q(s, a; \Theta) \doteq V(s; \Theta) + \left(A(s, a; \Theta) - \max_{a' \in \mathcal{A}(s)} A(s, a'; \Theta) \right) \qquad \text{(Type 2)}$$

where $Q(\cdot\,; \Theta)$ represents the parameterized dueling Q function.

3.2 Experiments

We simulate a total of 2×2 experiments on the CartPole 2.1 and Acrobot 2.2 environments using both the update types (Type 1, Type 2). We repeat each experiment 5 times, with each run consisting of 1000 episodes (until termination). To ensure that the experiments are consistent, we re-initialize the seed to be 184 before the start of every experiment.

We iterate through a combination of hyperparameters for each experiment to find the optimal values through grid search. We fix the value of the discount factor $\gamma = 0.99$ and use a linear scheduler for the exploration factor ϵ as $\epsilon_t = \max\left(\epsilon_{\min}, \epsilon_{\max} - (\epsilon_{\max} - \epsilon_{\min}) \times \frac{t}{\text{num\_episodes}}\right)$ for $\epsilon_{\min} = 10^{-4}$, $\epsilon_{\max} = 0.1$.

Parameter	Values
`replay_size`	[1000, 5000, 10000]
`update_steps`	[4, 10, 20]
`hidden_size`	[32, 48, 64]

Table 3.1: Experiment Configurations

Optimality Measure

We use the following notion of optimality to filter out the "best" hyperparameter configuration for each experiment.

Asymptotic Behaviour At the end of 1000 episodes, we evaluate the policy and utilize the average reward over 10 runs as the metric for comparison. However, we acknowledge that several hyperparameter configurations lead to asymptotic convergence.

Implementation Details

We use a shared feature extractor for the value and the advantage functions. At the start of every run, the weights of the neural network are re-initialized according to Xavier Normal. The network is trained using AdAM optimizer with a learning rate of 10^{-4}.

Note: The code snippets are beautified for demonstration purposes. Actual code may vary.

```python
    class QNetwork(nn.Module):
        def __init__(self, state_size, action_size, hidden_size):
            super(QNetwork, self).__init__()

            self.fc = nn.Sequential(
                nn.Linear(state_size, hidden_size),
                nn.ReLU(),
                nn.Linear(hidden_size, 4*hidden_size),
                nn.ReLU(),
            )
            self.value = nn.Sequential(
                nn.Linear(4*hidden_size, 1)
            )

            self.advantage = nn.Sequential(
                nn.Linear(4*hidden_size, action_size)
            )

        def forward(self, x, rule):
            x = self.fc(x)
            value = self.value(x)
            advantage = self.advantage(x)

            if rule == 'type1':
                Q = value + advantage - advantage.mean(dim=1, keepdim=True)
            elif rule == 'type2':
                Q = value + (advantage - torch.max(advantage, dim=1, keepdim=True)[0])
            return Q

        def select_action(self, state):
            with torch.no_grad():
                Q = self.forward(state)
                action = Q.argmax(dim=1)
            return action
```

Listing 3.1: Q Network Definition

```python
class Memory:
    @property
    def size(self):
        return len(self.buffer)

    def __init__(self, size):
        self.buffer = deque(maxlen=size)

    def add(self, experience):
        state, action, reward, next_state, done = experience
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size, continuous=True):
        batch_size = min(batch_size, self.size)

        if continuous:
            idx = np.random.randint(0, self.size - batch_size)
            return [self.buffer[i] for i in range(idx, idx + batch_size)]
        else:
            idx = np.random.choice(range(self.size), batch_size, replace=False)
            return [self.buffer[i] for i in idx]

    def clear(self):
        self.buffer.clear()
```

Listing 3.2: Replay Buffer

```python
with torch.no_grad():
    onlineQ_next = self.online_Qnetwork(next_state)
    targetQ_next = self.target_Qnetwork(next_state)
    online_action = onlineQ_next.argmax(dim=1, keepdim=True)
    y = batch_reward + gamma * targetQ_next.gather(1, online_action) * (1 - done)

loss = F.mse_loss(self.online_Qnetwork(state).gather(1, action), y)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Listing 3.3: Learning

3.2.1 on CartPole

3.2.2 on Acrobot

4 Policy Gradient Methods

In Policy Gradient methods, the policy π is parameterized as it offers an advantage over state action value based methods for continuous action spaces. The policy is parameterized as an exponential softmax distribution

$$\pi(a|\, s;\, \Theta) \doteq \frac{e^{h(s,\, a;\, \Theta)}}{\sum\limits_{a' \in \mathcal{A}(s)} e^{h(s,\, a';\, \Theta)}}$$

where $h(\cdot;\, \Theta)$ maybe a neural network. Parameterizing policies according to the softmax in action preferences enables the selection of actions with arbitrary probabilities. With continuous policy parameterization, the action probabilities change smoothly as a function of the learned parameter.
In this assignment, we focus on episodic Policy Gradient method, for which the performance measure is defined as the value of the start state of each episode

$$J(\Theta) = v_{\pi_\Theta}(s_0)$$

where v_{π_Θ} is the true value function for π_Θ. The policy gradient theorem for the episodic case establishes that

$$\nabla J(\Theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,\, a) \nabla \pi(a|\, s;\, \Theta)$$

where $\mu(\cdot)$ is the on-policy distribution under π.

Algorithm 4.1: (Episodic) Policy Gradient

Input: learning rate α; discount factor γ
Initialize: policy function π with random weights Θ
for episode $= 1$ to M **do**
  Initialize trajectory $\mathcal{T}$
  for $t = 1$ to T (or till termination) **do**
    Take action a_t following $\pi(\cdot|\, s;\, \Theta)$, observe reward r_t and next state s_{t+1} and store transition $(s_t,\, a_t,\, r_t)$ in $\mathcal{T}$
  for $t = 1$ to T **do**
    Compute $\nabla J_t(\Theta)$ for start state s_t from $\mathcal{T}$
    $\Theta \leftarrow \Theta + \alpha \gamma^t \nabla J_t(\Theta)$
Output: Parameterized policy $\pi(\cdot\,;\, \Theta)$

4.1 Monte-Carlo REINFORCE

The MC-REINFORCE algorithm utilizes Monte Carlo sampling to estimate gradients for policy optimization.

$$\begin{aligned}
\nabla J(\Theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s,\, a) \nabla \pi(a|\, s;\, \Theta) \\
&= \mathbb{E}_\pi \left[\sum_a q_\pi(s_t,\, a) \nabla \pi(a|\, s_t;\, \Theta) \right] \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a|\, s_t;\, \Theta) q_\pi(s_t,\, a) \frac{\nabla \pi(a|\, s_t;\, \Theta)}{\pi(a|\, s_t;\, \Theta)} \right] \\
&= \mathbb{E}_\pi \left[q_\pi(s_t,\, a_t) \frac{\nabla \pi(a_t|\, s_t;\, \Theta)}{\pi(a_t|\, s_t;\, \Theta)} \right] \\
&= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(a_t|\, s_t;\, \Theta)}{\pi(a_t|\, s_t;\, \Theta)} \right] \quad (\text{because } \mathbb{E}_\pi[G_t|\, s_t,\, a_t] = q_\pi(s_t,\, a_t))
\end{aligned}$$

The update equation of its policy parameter Θ is given by

$$\Theta \leftarrow \Theta + \alpha G_t \frac{\nabla \pi(a_t|\, s_t;\, \Theta)}{\pi(a_t|\, s_t;\, \Theta)} \qquad \text{(w/o Baseline)}$$

$$\Theta \leftarrow \Theta + \alpha (G_t - V(s_t;\, \Phi)) \frac{\nabla \pi(a_t|\, s_t;\, \Theta)}{\pi(a_t|\, s_t;\, \Theta)} \qquad \text{(w/ Baseline)}$$

where $V(\cdot;\, \Phi)$ is the state value function updated by TD(0) method. It can be shown that by introducing a baseline (as long as it does not vary with a)

$$\nabla J(\Theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s,\, a) - b(s)) \nabla \pi(a|\, s;\, \Theta)$$

the equation remains valid.

4.2 Experiments

We simulate a total of 2×2 experiments on the CartPole 2.1 and Acrobot 2.2 environments using both the update types (w/o Baseline, w/ Baseline). We repeat each experiment 5 times, with each run consisting of 1000 episodes (until termination). To ensure that the experiments are consistent, we re-initialize the seed to be 184 before the start of every experiment.

We iterate through a combination of hyperparameters for each experiment to find the optimal values through grid search. We fix the value of the discount factor $\gamma = 0.99$ throughout the experiments.

Parameter	Values
`hidden_size`	[32, 48, 64]

Table 4.1: Experiment Configurations

Optimality Measure

We use the following notion of optimality to filter out the "best" hyperparameter configuration for each experiment.

Asymptotic Behaviour At the end of 1000 episodes, we evaluate the policy and utilize the average reward over 10 runs as the metric for comparison. However, we acknowledge that several hyperparameter configurations lead to asymptotic convergence.

Implementation Details

We use independent feature extractors for the policy and the value functions. At the start of every run, the weights of the neural network are re-initialized according to Xavier Normal. The network is trained using AdAM optimizer with a learning rate of 10^{-4}.

Note: The code snippets are beautified for demonstration purposes. Actual code may vary.

```
    class PolicyNetwork(nn.Module):
        def __init__(self, state_size, action_size, hidden_size):
            super(PolicyNetwork, self).__init__()

            self.fc = nn.Sequential(
                nn.Linear(state_size, hidden_size),
                nn.ReLU(),
                nn.Linear(hidden_size, 4*hidden_size),
                nn.ReLU(),
                nn.Linear(4*hidden_size, action_size)
            )

        def forward(self, x):
            return self.fc(x)

        def select_action(self, state):
            with torch.no_grad():
                action = self.forward(state)
                m = Categorical(F.softmax(action, dim=-1))
            return m.sample(), m.log_prob(action)
```

Listing 4.1: Policy Network Definition

```
    class ValueNetwork(nn.Module):
        def __init__(self, state_size, hidden_size):
            super(ValueNetwork, self).__init__()

            self.fc = nn.Sequential(
                nn.Linear(state_size, hidden_size),
                nn.ReLU(),
                nn.Linear(hidden_size, 4*hidden_size),
                nn.ReLU(),
                nn.Linear(4*hidden_size, 1)
            )

        def forward(self, x):
            return self.fc(x)
```

Listing 4.2: Value Network Definition

```
    G = []
    total = 0
    with torch.no_grad():
        for r in batch_reward[::-1]:
            total = r + gamma * total
            G.insert(0, total)
        G = (G - G.mean()) / G.std()

    _, log_prob = self.policy.select_action(state)
    if rule == 'wbaseline':
        loss_value = F.mse_loss(self.value(state), G)
        optimizer_value.zero_grad()
        loss_value.backward()
        optimizer_value.step()

        deltas = G - value.detach()
        loss_policy = (-log_prob * deltas).sum()
    elif rule == 'wobaseline':
        deltas = G
        loss_policy = (-log_prob * deltas).sum()

    optimizer_policy.zero_grad()
    loss_policy.backward()
    optimizer_policy.step()
```

Listing 4.3: Learning

4.2.1 on CartPole

4.2.2 on Acrobot