CSE584 - Homework 2

Krishnasai Paleti - kxp5619

October 2024

1 Abstract

The inverted pendulum swingup problem is based on the classic problem in control theory. The system consists of a pendulum attached at one end to a fixed point, and the other end being free. The pendulum starts in a random position and the goal is to apply torque on the free end to swing it into an upright position, with its center of gravity right above the fixed point.

The code implements the Soft Actor-Critic (SAC) algorithm to solve a continuous control problem, specifically the Pendulum-v1 environment from the OpenAI Gym. SAC is a state-of-the-art, off-policy, model-free reinforcement learning algorithm that maximizes a trade-off between expected reward and entropy. The inclusion of an entropy term in the objective function encourages exploration by preventing premature convergence to suboptimal policies.

The implementation involves three main components: (1) a policy network (actor), which generates stochastic actions by modeling a Gaussian distribution over possible actions, (2) two Q-networks (critics), which estimate the expected cumulative rewards for state-action pairs, and (3) target Q-networks, which are soft-updated versions of the Q-networks to provide stable targets for training. Additionally, it uses an adaptive temperature parameter, referred to as alpha, which is dynamically adjusted to control the importance of the entropy term and maintain a desired level of exploration.

The learning process is driven by a replay buffer, which stores past experiences in the form of state, action, reward, next state, and done flag tuples. The SAC algorithm iteratively samples mini-batches from this buffer to update the actor and critic networks. The Q-network updates minimize the temporal difference (TD) error using a smoothed loss function, while the policy network updates maximize the minimum Q-value augmented with an entropy regularization term to encourage diverse action selection. The target Q-values are computed using the minimum of the two Q-values from the target Q-networks, which improves learning stability and reduces overestimation bias.

Overall, this SAC implementation leverages the reparameterization trick for efficient backpropagation through the stochastic policy, soft updates for target networks to prevent drastic changes, and an automatic adjustment of the entropy term to achieve a balance between exploration and exploitation. The code iterates through multiple episodes and performs action sampling, experience storage, and network training, ultimately seeking to achieve optimal control of the pendulum by learning a stable and efficient policy.

2 Commented Code File - (only Algo)

```
import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Normal
import numpy as np
import collections, random

# Hyperparameters for the SAC algorithm
lr_pi = 0.0005          # Learning rate for the policy network (actor)
lr_q = 0.001            # Learning rate for Q-networks (critics)
init_alpha = 0.01       # Initial value of the entropy coefficient (alpha)
gamma = 0.98            # Discount factor for future rewards
batch_size = 32         # Mini-batch size for training
buffer_limit = 50000    # Maximum size of the replay buffer
```

```
tau = 0.01              # Soft update factor for target networks
target_entropy = -1.0   # Desired target entropy level
lr_alpha = 0.001        # Learning rate for alpha update (entropy coefficient)

# Replay Buffer for storing and sampling experiences
class ReplayBuffer():
    def __init__(self):
        # Create a deque buffer with a maximum length
        self.buffer = collections.deque(maxlen=buffer_limit)

    def put(self, transition):
        # Add a new experience (transition) to the buffer
        self.buffer.append(transition)

    def sample(self, n):
        # Randomly sample a mini-batch of size n from the buffer
        mini_batch = random.sample(self.buffer, n)

        # Separate batch elements into individual lists for states, actions, rewards, etc.
        s_lst, a_lst, r_lst, s_prime_lst, done_mask_lst = [], [], [], [], []

        for transition in mini_batch:
            s, a, r, s_prime, done = transition
            s_lst.append(s)             # Store state
            a_lst.append([a])           # Store action
            r_lst.append([r])           # Store reward
            s_prime_lst.append(s_prime) # Store next state
            # Mask terminal states (0 if done, 1 otherwise)
            done_mask = 0.0 if done else 1.0
            done_mask_lst.append([done_mask])

        # Convert lists to PyTorch tensors for efficient computation
        return torch.tensor(s_lst, dtype=torch.float),
               torch.tensor(a_lst, dtype=torch.float), \
               torch.tensor(r_lst, dtype=torch.float),
               torch.tensor(s_prime_lst, dtype=torch.float), \
               torch.tensor(done_mask_lst, dtype=torch.float)

    def size(self):
        # Return the current size of the buffer
        return len(self.buffer)

# Policy Network (Actor) for generating actions
class PolicyNet(nn.Module):
    def __init__(self, learning_rate):
        super(PolicyNet, self).__init__()
        # Define neural network layers
        self.fc1 = nn.Linear(3, 128)     # Fully connected layer for state input
         # Output layer for mean of the action distribution
        self.fc_mu = nn.Linear(128, 1)
        # Output layer for std dev of the action distribution
        self.fc_std = nn.Linear(128, 1)

        # Optimizer for policy network parameters
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)

        # Initialize log of alpha for entropy regularization
        self.log_alpha = torch.tensor(np.log(init_alpha))
        self.log_alpha.requires_grad = True  # Enable gradients for alpha
        # Optimizer for updating alpha (entropy coefficient)
```

```
        self.log_alpha_optimizer = optim.Adam([self.log_alpha], lr=lr_alpha)

    def forward(self, x):
        # Forward pass through the network
        # Apply ReLU activation on input layer
        x = F.relu(self.fc1(x))
        # Calculate mean of the action distribution
        mu = self.fc_mu(x)
        # Calculate std dev using softplus to ensure positivity
        std = F.softplus(self.fc_std(x))

        # Create a Gaussian distribution for the action
        dist = Normal(mu, std)
        # Sample action using reparameterization trick
        action = dist.rsample()
        # Calculate log probability of the action
        log_prob = dist.log_prob(action)

        # Scale action using tanh for bounded actions (between -1 and 1)
        real_action = torch.tanh(action)
        # Adjust log probability for tanh transformation
        real_log_prob = log_prob - torch.log(1 - torch.tanh(action).pow(2) + 1e-7)
        return real_action, real_log_prob

    def train_net(self, q1, q2, mini_batch):
        # Extract states from mini-batch
        s, _, _, _, _ = mini_batch

        # Generate actions and log probabilities from policy
        a, log_prob = self.forward(s)

        # Calculate entropy regularization term for exploration
        entropy = -self.log_alpha.exp() * log_prob

        # Get Q-values for actions from both Q-networks
        q1_val, q2_val = q1(s, a), q2(s, a)
        q1_q2 = torch.cat([q1_val, q2_val], dim=1)  # Concatenate Q-values
        min_q = torch.min(q1_q2, 1, keepdim=True)[0]  # Find minimum Q-value

        # Policy loss: maximize Q-value and entropy (via gradient ascent)
        loss = -min_q - entropy
        self.optimizer.zero_grad()
        loss.mean().backward()  # Backpropagation to compute gradients
        self.optimizer.step()   # Update policy network weights

        # Update alpha (entropy coefficient) to balance exploration
        self.log_alpha_optimizer.zero_grad()
        alpha_loss = -(self.log_alpha.exp() * (log_prob + target_entropy).detach()).mean()
        alpha_loss.backward()
        self.log_alpha_optimizer.step()

# Q Network (Critic) for estimating state-action values
class QNet(nn.Module):
    def __init__(self, learning_rate):
        super(QNet, self).__init__()
        # Define separate layers for state and action inputs
        self.fc_s = nn.Linear(3, 64)  # State input layer
        self.fc_a = nn.Linear(1, 64)  # Action input layer
        self.fc_cat = nn.Linear(128, 32)  # Concatenated layer for state-action pair
        self.fc_out = nn.Linear(32, 1)  # Output layer for Q-value
```

```
        # Optimizer for Q-network parameters
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)

    def forward(self, x, a):
        # Forward pass for Q-value estimation
        h1 = F.relu(self.fc_s(x))  # State representation with ReLU activation
        h2 = F.relu(self.fc_a(a))  # Action representation with ReLU activation
        cat = torch.cat([h1, h2], dim=1)  # Concatenate state and action representations
        q = F.relu(self.fc_cat(cat))  # Pass through concatenated layer with ReLU
        q = self.fc_out(q)  # Output Q-value
        return q

    def train_net(self, target, mini_batch):
        # Extract mini-batch elements
        s, a, r, s_prime, done = mini_batch
        # Calculate Smooth L1 loss between predicted Q and target Q
        loss = F.smooth_l1_loss(self.forward(s, a), target)
        self.optimizer.zero_grad()
        loss.mean().backward()  # Backpropagation
        self.optimizer.step()   # Update Q-network weights

    def soft_update(self, net_target):
        # Soft update of target network weights using tau factor
        for param_target, param in zip(net_target.parameters(), self.parameters()):
            param_target.data.copy_(param_target.data * (1.0 - tau) + param.data * tau)

# Calculate target Q-value using the target Q-network
def calc_target(pi, q1, q2, mini_batch):
    s, a, r, s_prime, done = mini_batch

    with torch.no_grad():
        # Get next action and log probability from policy for next state
        a_prime, log_prob = pi(s_prime)
        # Calculate entropy for exploration
        entropy = -pi.log_alpha.exp() * log_prob
        # Get Q-values from target Q-networks for the next state
        q1_val, q2_val = q1(s_prime, a_prime), q2(s_prime, a_prime)
        q1_q2 = torch.cat([q1_val, q2_val], dim=1)  # Concatenate Q-values
        min_q = torch.min(q1_q2, 1, keepdim=True)[0]  # Minimum Q-value

        # Calculate target Q-value with entropy term
        target = r + gamma * done * (min_q + entropy)

    return target

# Main training loop for SAC
def main():
    env = gym.make('Pendulum-v1')  # Create the environment
    memory = ReplayBuffer()  # Initialize replay buffer

    # Initialize Q-networks and policy network
    q1, q2, q1_target, q2_target = QNet(lr_q), QNet(lr_q), QNet(lr_q), QNet(lr_q)
    pi = PolicyNet(lr_pi)

    # Load initial weights into target Q-networks
    q1_target.load_state_dict(q1.state_dict())
    q2_target.load_state_dict(q2.state_dict())

    score = 0.0  # For tracking average episode reward
```

```
    print_interval = 20  # Interval for printing results

    for n_epi in range(10000):  # Iterate through episodes
        s, _ = env.reset()  # Reset environment for a new episode
        done = False
        count = 0  # Step counter for episode

        # Interact with the environment
        while count < 200 and not done:
            # Get action from policy network
            # Take action in env
            a, log_prob = pi(torch.from_numpy(s).float())
            s_prime, r, done, truncated, info = env.step([2.0 * a.item()])
            # Store transition in replay buffer
            memory.put((s, a.item(), r/10.0, s_prime, done))
            score += r  # Accumulate reward
            s = s_prime  # Move to next state
            count += 1  # Increment step counter

        # Train after the buffer has enough samples
        if memory.size() > 1000:
            for i in range(20):  # Perform 20 updates per episode
                mini_batch = memory.sample(batch_size)  # Sample mini-batch
                # Calculate target Q-value
                td_target = calc_target(pi, q1_target, q2_target, mini_batch)
                q1.train_net(td_target, mini_batch)  # Update Q-network 1
                q2.train_net(td_target, mini_batch)  # Update Q-network 2
                pi.train_net(q1, q2, mini_batch)  # Update policy network
                q1.soft_update(q1_target)  # Soft update Q-target 1
                q2.soft_update(q2_target)  # Soft update Q-target 2

        # Print results every 20 episodes
        if n_epi % print_interval == 0 and n_epi != 0:
            print("# of episode :{}, avg score : {:.1f} alpha:{:.4f}".format(
                n_epi, score / print_interval, pi.log_alpha.exp()))
            score = 0.0  # Reset score tracker

    env.close()  # Close the environment

if __name__ == '__main__':
    main()  # Run the main function
```

3 References

- https://github.com/seungeunrho/minimalRL/blob/master/sac.py
- https://spinningup.openai.com/en/latest/algorithms/sac.html