Analysis of SAC-Based Autonomous Driving Implementation

Abstract

The code implements a Soft Actor-Critic (SAC) reinforcement learning algorithm for autonomous vehicle control. The implementation addresses the intricate challenge of autonomous driving by processing high-dimensional sensor data, such as LiDAR, and vehicle states to output continuous control actions, including steering and acceleration. The system employs twin critics to mitigate Q-value overestimation and incorporates automatic entropy adjustment to balance exploration and exploitation. The architecture is comprised of three principal components: a state processor that handles sensor data, an actor network that generates driving actions, and critic networks that evaluate these actions. The implementation stands out for its practical considerations, including adherence to safety constraints, real-time processing capabilities, and integration with vehicle dynamics.

Action Selection Function (select_action): Implements the core reinforcement learning concept of policy-based action selection, generating continuous control actions (e.g., steering and acceleration) based on the current system state.

Policy and Value Update Function (update): Implements policy evaluation and improvement by updating actor and critic networks through sampling from the replay buffer and calculating target Q-values, which are essential for learning optimal behaviors.

State Processing Function (process_state): Converts raw sensor data (e.g., LiDAR readings) into a normalized state representation, providing the agent with a stable input for learning and decision-making.

Soft Update Function (_soft_update): Implements the core reinforcement learning concept of stability through soft target updates, ensuring gradual changes in the target network for better training convergence.

Replay Buffer Class (ReplayBuffer): Facilitates experience replay, which is crucial for breaking temporal correlations and improving the stability and efficiency of the learning process.

Core Implementation Analysis

Core Function 1: Action Selection

The `select_action` function is responsible for generating driving actions based on the current system state. It takes as input a processed state, which includes LiDAR and vehicle data, and outputs a continuous action for controlling the vehicle (e.g., steering and acceleration). During evaluation, deterministic actions are used, whereas during training, actions are sampled from a Gaussian distribution via the reparameterization trick, allowing for backpropagation through the stochastic sampling process. The tanh function is applied to the selected action to ensure it remains within safety bounds.

```
def select_action(self, state, evaluate=False):
        """
        Select driving action based on current state.
        Args:
            state: Processed state including LiDAR and vehicle data.
            evaluate: Boolean flag for evaluation mode.
        Returns:
            numpy array of continuous actions (steering, acceleration).
        """
        # Convert state to PyTorch tensor and move to GPU if available.
        state = torch.FloatTensor(state).to(device)

        # Pass state through actor network to get action distribution parameters.
        mean, log_std = self.actor(state)
        # Convert log standard deviation to standard deviation.
        std = log_std.exp()

        # Create normal distribution for action sampling.
        normal = Normal(mean, std)

        if evaluate:
            # During evaluation (e.g., actual driving), use deterministic action.
            action = mean
```

```
        else:
            # During training, sample action using reparameterization trick,
            # which allows backpropagation through the random sampling process.
            x_t = normal.rsample()
            # Apply tanh to bound actions between -1 and 1 (safety constraint).
            action = torch.tanh(x_t)

        # Convert action tensor to numpy array for interaction with the environment.
        return action.cpu().detach().numpy()
```

Core Function 2: Policy and Value Update

The `update` function is central to the SAC algorithm, managing updates to the actor (policy) and critic (value) networks. This function involves sampling transitions from the replay buffer, calculating target Q-values using the Bellman equation, updating both critic networks to minimize the Temporal Difference (TD) error, and optimizing the actor network to maximize expected returns. The critic networks are updated by minimizing the mean squared error between predicted Q-values and target Q-values, while the actor is updated using the Q-value gradient. Additionally, a soft update is performed for the target networks to promote stability.

```
def update(self, replay_buffer, batch_size=256):
        """
        Update policy and value networks using the SAC algorithm.
        Args:
            replay_buffer: Buffer containing experience tuples.
            batch_size: Number of samples for update.
        """
        # Sample a random batch of transitions from the replay buffer.
        state, action, reward, next_state, done = replay_buffer.sample(batch_size)

        # Convert all samples to PyTorch tensors.
        state = torch.FloatTensor(state).to(device)          # Current states.
        action = torch.FloatTensor(action).to(device)        # Actions taken.
        reward = torch.FloatTensor(reward).to(device)        # Rewards received.
        next_state = torch.FloatTensor(next_state).to(device) # Resulting states.
        done = torch.FloatTensor(done).to(device)            # Episode termination flags.

        # Compute target Q-values (no gradient needed).
        with torch.no_grad():
            # Sample next actions from the current policy.
```

```
            next_mean, next_log_std = self.actor(next_state)
            next_std = next_log_std.exp()
            next_normal = Normal(next_mean, next_std)
            next_action = torch.tanh(next_normal.rsample())

            # Compute Q-values from both target critics.
            target_Q1 = self.critic1_target(next_state, next_action)
            target_Q2 = self.critic2_target(next_state, next_action)
            # Use the minimum Q-value to mitigate overestimation.
            target_Q = torch.min(target_Q1, target_Q2)

            # Compute the target using the Bellman equation with an entropy term.
            target_Q = reward + (1 - done) * self.gamma * target_Q

        # Update the first critic network.
        current_Q1 = self.critic1(state, action)
        critic1_loss = F.mse_loss(current_Q1, target_Q)
        self.critic1_optimizer.zero_grad()
        critic1_loss.backward()
        self.critic1_optimizer.step()

        # Update the second critic network.
        current_Q2 = self.critic2(state, action)
        critic2_loss = F.mse_loss(current_Q2, target_Q)
        self.critic2_optimizer.zero_grad()
        critic2_loss.backward()
        self.critic2_optimizer.step()

        # Update the actor (policy) network.
        # Sample actions from the current policy.
        mean, log_std = self.actor(state)
        std = log_std.exp()
        normal = Normal(mean, std)
        x_t = normal.rsample()
        action = torch.tanh(x_t)

        # Compute Q-values for policy actions.
        Q1 = self.critic1(state, action)
        Q2 = self.critic2(state, action)
        Q = torch.min(Q1, Q2)

        # Compute the policy loss (negative of Q-value).
        actor_loss = -(Q).mean()
```

```
        # Update the actor network.
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        # Soft update target networks.
        self._soft_update(self.critic1, self.critic1_target)
        self._soft_update(self.critic2, self.critic2_target)

        # Logging for monitoring training progress.
        self.logger.add_scalar('loss/critic1', critic1_loss.item())
        self.logger.add_scalar('loss/critic2', critic2_loss.item())
        self.logger.add_scalar('loss/actor', actor_loss.item())
        self.logger.add_scalar('q_value/target', target_Q.mean().item())
```

Core Function 3: State Processing

The `process_state` function is responsible for transforming raw sensor data, including LiDAR readings and vehicle dynamics, into a fixed-length state representation that the agent can utilize. The processed state combines both LiDAR and vehicle information and is normalized to ensure stable learning dynamics.

```
def process_state(self, raw_state):
        """
        Convert raw sensor data into agent-ready state representation.
        Args:
            raw_state: Dictionary of raw sensor and vehicle data.
        Returns:
            Processed state vector.
        """
        # Initialize a zero array for the processed state.
        processed_state = np.zeros(self.state_dim)

        # Process LiDAR point cloud data.
        # Convert to fixed-length representation and normalize.
        lidar = self._process_lidar(raw_state['lidar'])

        # Extract vehicle dynamics information.
        speed = raw_state['speed']           # Current vehicle speed.
        steering = raw_state['steering']     # Current steering angle.
```

```
        acceleration = raw_state['acceleration']  # Current acceleration.

        # Combine all state components into a single vector.
        # First part: LiDAR data.
        processed_state[:self.lidar_points] = lidar
        # Second part: Vehicle dynamics.
        processed_state[self.lidar_points:] = [speed, steering, acceleration]

        # Normalize the processed state.
        processed_state = (processed_state - self.state_mean) / (self.state_std + 1e-8)

        return processed_state
```

Core Function 4: Soft Update of Target Networks

The `_soft_update` function manages the stable updating of target networks. This function interpolates between the weights of the target network and those of the current network using a small value for `tau`, which ensures that target networks are updated gradually. Such gradual updates are critical for maintaining training stability.

```
def _soft_update(self, source_net, target_net, tau=0.005):
        """
        Soft update model parameters for target networks.
        Args:
            source_net: Source network (critic or actor).
            target_net: Target network to be updated.
            tau: Interpolation factor.
        """

        for target_param, param in zip(target_net.parameters(), source_net.parameters()):
            target_param.data.copy_(tau * param.data + (1.0 - tau) * target_param.data)
```

Learning Process Flow

1. **State Processing**:
    - Raw sensor data → Normalized state vector.
    - LiDAR data processing and vehicle state combination.
2. **Action Selection**:
    - State → Actor network → Action distribution.
    - Sample actions within safety constraints.
3. **Environment Interaction**:
    - Execute action → Observe reward and next state.
    - Store experience in the replay buffer.
4. **Learning Update**:
    - Sample a batch from the replay buffer.
    - Update critics to minimize the TD error.
    - Update the actor to maximize the expected return.
    - Soft update target networks for stability.

In summary, this report provides a comprehensive analysis of the Soft Actor-Critic (SAC) reinforcement learning algorithm as applied to autonomous driving. The SAC implementation encompasses multiple core functions, including action selection, policy and value updates, state processing, and experience replay. These components work together to address the complexities of continuous control, efficient learning from high-dimensional sensory data, and maintaining stability throughout training. By leveraging techniques like twin critic networks, entropy tuning, and soft target updates, the SAC algorithm demonstrates its robustness and efficacy in dynamic, real-world environments, making it a suitable solution for autonomous vehicle control.