CSE584 Homework2

Xinzhang Xiong

Abstract

This project implements a reinforcement learning agent in a simulated grid environment, a policy gradient method. The environment is structured as a 5x5 grid where the agent interacts with obstacles, a goal, and dynamic rewards. The agent learns a policy to navigate the grid and reach the goal while avoiding obstacles by maximizing the cumulative reward. The **environment.py** file defines the graphical environment and reward system, and **reinforcement_agent.py** contains the reinforcement algorithm implementation, including the neural network model, policy optimization, and discounted reward calculations. Over the episodes, the agent's policy network learns optimal actions through trial and error, with the goal of improving its navigation strategy based on the rewards.

Code Core Sections:

```
# create error function and training function to update policy network
def optimizer(self):
    # Placeholder tensor for actions chosen during an episode, with shape [batch_size, action_size]
    action = K.placeholder(shape=[None, 5])
    # Placeholder tensor for discounted rewards associated with each action, with shape [batch_size,]
    discounted_rewards = K.placeholder(shape=[None, ])

    # Calculate cross entropy error function
     # Computes the probability of the selected actions by summing over the output probabilities
     # based on actions taken (1 for chosen action, 0 for others) in 'action'
    action_prob = K.sum(action * self.model.output, axis=1)
     # Calculates the policy's cross-entropy loss weighted by the discounted rewards
     # This will encourage actions that lead to higher rewards while penalizing those with lower rewards
    cross_entropy = K.log(action_prob) * discounted_rewards
     # Takes the negative sum of cross-entropy to define the loss (policy gradient maximizes this)
    loss = -K.sum(cross_entropy)

    # create training function
    optimizer = Adam(lr=self.learning_rate)
    updates = optimizer.get_updates(self.model.trainable_weights, [],
                                    loss)
    train = K.function([self.model.input, action, discounted_rewards], [],
                       updates=updates)

    return train
```

```python
    # get action from policy network
    def get_action(self, state):
        # Predicts the probability distribution for each action in the current state using the policy network.
        policy = self.model.predict(state)[0]

        # Randomly selects an action based on the predicted policy distribution to introduce exploration.
        return np.random.choice(self.action_size, 1, p=policy)[0]

    # calculate discounted rewards
    def discount_rewards(self, rewards):
        # Initialize an array to store discounted rewards, matching the shape of the rewards array.
        discounted_rewards = np.zeros_like(rewards)
        # Initialize the running total to accumulate discounted rewards.
        running_add = 0

         # Iterate over rewards in reverse order (from the end of the episode to the beginning).
        for t in reversed(range(0, len(rewards))):
            running_add = running_add * self.discount_factor + rewards[t]
            discounted_rewards[t] = running_add
        return discounted_rewards

    # save states, actions and rewards for an episode
    def append_sample(self, state, action, reward):
        # Stores the current state by appending it to the agent's list of states.
        self.states.append(state[0])
        # Adds the current reward to the agent's list of rewards.
        self.rewards.append(reward)

        # Creates a one-hot encoded vector representing the chosen action.
        act = np.zeros(self.action_size)
        act[action] = 1
        self.actions.append(act)

    # update policy neural network
    def train_model(self):
        # Computes the discounted rewards for the episode and normalizes them.
        discounted_rewards = np.float32(self.discount_rewards(self.rewards))
        discounted_rewards -= np.mean(discounted_rewards)
        discounted_rewards /= np.std(discounted_rewards)

        # Trains the policy network with states, actions, and normalized discounted rewards.
        self.optimizer([self.states, self.actions, discounted_rewards])
        # Resets states, actions, and rewards for the next episode.
        self.states, self.actions, self.rewards = [], [], []
```