Connect 4 Reinforcement Learning Model

github: https://github.com/neoyung/connect-4

Summary

This repository implements Reinforcement Learning in the context of a machine learning model that learns how to play the game Connect 4. Reinforcement Learning has shown excellent performance in game-based models, as it performs very well at dealing with the trial-and-error nature of playing any type of game where you make a move and receive rewards for actions. The repository itself contains the following sections: the game setup, an experience replay module, a Deep-Q Network (DQN), an optimizer function, and, of course, the training loop.

The game is set up in Python utilizing an array for the game board. The module is trained in the following way. Game boards are passed into the DQN and return the expected (Q) values for placing a piece at any currently available positions in the game board. These can be considered the rewards in reinforcement learning. The DQN is optimized using the Bellman optimality equation which updates the Q-values. The experience replay module helps to improve efficiency by storing each state transition (move on the board or game state) which helps to generalize the DQN.

Two agents are trained and every 1000 epochs, the agents play against each other. The agent who wins 55% of the games is now considered the best model, and subsequent agents are trained and then compared to each other. This refines the best model until the best model is known.

Core Modules

I will not cover the training loop or the game setup as that is mostly irrelevant to the reinforcement learning aspect of this module.

Experience Replay Module

Essentially this module is storing transition tuples of form (state, action, reward, next_state). What these are used for is simple training data of what happens for each game state and finding out what moves correspond to higher values. These are passed into the DQN for improved training data. Most of this code is very self-explanatory

```
import random

# memory block for deep q learning
class replayMemory:
    def __init__(self):
        self.memory = []
```

```
    def dump(self, transition_tuple):
        self.memory.append(transition_tuple)

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

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

memory = replayMemory()
```

Deep-Q Network

DQNs are models designed to predict reward maps by generating Q-values. These Q-values determine how good future actions are and allow the models to make the best decision at each step. DQNs are constructed via CNNs as seen below here.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class DQN(nn.Module):

    def __init__(self, outputs):
        super(DQN, self).__init__()
        # 6 by 7, 10 by 11

        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=5, padding=2)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, padding=2)
        self.conv4 = nn.Conv2d(32, 32, kernel_size=5, padding=2)
        self.conv5 = nn.Conv2d(32, 32, kernel_size=5, padding=2)
        self.conv6 = nn.Conv2d(32, 32, kernel_size=5, padding=2)
        self.conv7 = nn.Conv2d(32, 32, kernel_size=5, padding=2)

        linear_input_size = 6 * 7 * 32
        self.MLP1 = nn.Linear(linear_input_size, 50)
        self.MLP2 = nn.Linear(50, 50)
        self.MLP3 = nn.Linear(50, 50)
        self.MLP4 = nn.Linear(50, outputs)

    def forward(self, x):
        x = F.leaky_relu(self.conv1(x))
        x = F.leaky_relu(self.conv2(x))
        x = F.leaky_relu(self.conv3(x))
        x = F.leaky_relu(self.conv4(x))
```

```
        x = F.leaky_relu(self.conv5(x))
        x = F.leaky_relu(self.conv6(x))
        x = F.leaky_relu(self.conv7(x))
        # flatten the feature vector except batch dimension
        x = x.view(x.size(0), -1)
        x = F.leaky_relu(self.MLP1(x))
        x = F.leaky_relu(self.MLP2(x))
        x = F.leaky_relu(self.MLP3(x))
        return self.MLP4(x)
```

This code works by using a variety of convolutional layers to extract relevant features and passing them through fully connected layers to determine our final Q-map. This is a very simple network.

Agent

This code implements the two agents that will be playing the game based on the models. These models will play the game based on an epsilon-greedy strategy. The epsilon-greedy policy is a strategy used in reinforcement learning where an agent chooses a random action with a probability of ε (exploration) and the best-known action with a probability of (1 - ε) (exploitation). This approach allows the use of both learned knowledge as well as the ability for the model to try new moves it wouldn't have tried before.

```
import torch.optim as optim
import math

BATCH_SIZE = 256
GAMMA = 0.999 //Discounts future rewards very slightly in order to maintain learned knowledge

//this just determines what actions are available from the loaded model and the current state of the game. It also loads the models and puts them into evaluation mode.
# get max no. of actions from action space
n_actions = env.board_width

height = env.board_height
width = env.board_width

policy_net = DQN(n_actions).to(device)
# target_net will be updated every n episodes to tell policy_net a better estimate of how far off from convergence
target_net = DQN(n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())
# set target_net in testing mode
target_net.eval()

optimizer = optim.Adam(policy_net.parameters())

//after loading the model, choose it's best action using epsilon optimality
```

```
def select_action(state, available_actions, steps_done=None, training=True):
    # batch and color channel
    state = torch.tensor(state, dtype=torch.float, device=device).unsqueeze(dim=0).unsqueeze(dim=0)
    epsilon = random.random()
    if training:
        eps_threshold = EPS_END + (EPS_START - EPS_END) * math.exp(-1 * steps_done /
EPS_DECAY)
    else:
        eps_threshold = 0

//based on the greedy policy, the following code will choose the best action possible if it's above a certain epsilon threshold which would determine it being a good move. If no move is above that threshold it will simply choose a random move.
    # follow epsilon-greedy policy
    if epsilon > eps_threshold:
        with torch.no_grad():
            # action recommendations from policy net
            r_actions = policy_net(state)[0, :]
            state_action_values = [r_actions[action] for action in available_actions]
            argmax_action = np.argmax(state_action_values)
            greedy_action = available_actions[argmax_action]
            return greedy_action
    else:
        return random.choice(available_actions)
```

Optimization Policy

This optimize function predicts the Q-values for the current state using a policy network. This policy network is one of the agents we've been training. This function then takes those predicted values and computes the actual expected Q-values using the Bellman equation. The Huber loss is then calculated to find the difference and loss between these values. Weights are then updated and the model continues.

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    //Sample a batch of transitions from experience replay
    transitions = memory.sample(BATCH_SIZE)
    state_batch, action_batch, reward_batch, next_state_batch = zip(*[
        (np.expand_dims(m[0], axis=0), [m[1]], m[2], np.expand_dims(m[3], axis=0)) for m in transitions
    ])

    //Converts all the lists to tensors
    state_batch = torch.tensor(state_batch, dtype=torch.float, device=device)
    action_batch = torch.tensor(action_batch, dtype=torch.long, device=device)
    reward_batch = torch.tensor(reward_batch, dtype=torch.float, device=device)
```

```
    non_final_mask = torch.tensor(tuple(map(lambda s_: s_[0] is not None, next_state_batch)),
device=device)
    non_final_next_state = torch.cat([
        torch.tensor(s_, dtype=torch.float, device=device).unsqueeze(0) for s_ in next_state_batch if s_[0] is
not None
    ])

//uses the policy network to evaluate the next state q values
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_state).max(1)[0].detach()

    //compute the expected Q values using the Bellman equation
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    //Calculate the Huber loss between predicted and expected Q values
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze(1))  #
Unsqueeze to match dimensions

    //perform backpropagation to update the model weights
    optimizer.zero_grad()  # Reset gradients
    loss.backward()  # Compute gradients
    optimizer.step()  # Update weights based on gradients
```