CSE 584 – Fall 2024 – Homework 2

Sandhya Somasundaram
939157171
sms9723@psu.edu

October 27 2024, Sunday

1 Abstract

In this project, an agent would be developed that can play the game Breakout developed by Atari; this is a classic task in the domain of RL research whereby an agent learns to control a paddle to keep a ball in play and break bricks. The problem is modelled as a Markov Decision Process (MDP) with the environment providing the agent with an observable state in the form of game frames. The agent has to pick actions so as to maximize the cumulative reward with respect to score. Therefore, the task is to train a model capable of processing visual inputs to understand the dynamics of the game and learn an effective policy to maximize the score over multiple episodes.

We implement the Deep Q-Network (DQN) algorithm, a popular RL method for choosing actions in games. It combines value-based Q-learning with deep learning using CNNs to process raw pixel data from game frames. The convolution layers are used to capture the spatial features, and the fully connected layers set up for predicting Q-values over the expected future rewards for every action given the state. An epsilon-greedy policy helps with balancing exploration versus exploitation in a sense that the agent must start exploring new actions and then slowly shifts toward exploiting learned knowledge to maximize the score. Also, some further steps, such as frame stacking and preprocessing, are applied to transform raw game frames into a form that can serve as input for the CNN model in the capture of temporal dependencies within game play.

The model will learn to predict the Q-values and optimally select actions based on the cumulative rewards it receives over episodes. We use experience replay and the model's pre-trained weights for training to be stable, hence the generalization of the agent in various game scenarios. Thereby, we are having a solid setup of decision making by the agent, equipping with progressive improvement through trials and errors within the Breakout environment.

The code has 2 parts - breakout and play. The breakout section implements the code while the play loads the model saved in breakout and runs the file.

2 Commented Code

Breakout: (breakout_dqn.py)

```python
# Import necessary libraries for the environment, math operations, neural network, and image processing
import gym  # Toolkit for reinforcement learning environments
import random
import numpy as np
import tensorflow as tf
from collections import deque
from skimage.color import rgb2gray
from skimage.transform import resize
from keras.models import Sequential
from keras.optimizers import RMSprop
from keras.layers import Dense, Flatten
from keras.layers.convolutional import Conv2D
from keras import backend as K


# Define the total number of episodes for training
EPISODES = 50000  # Number of episodes for training the agent

# Define a class DQNAgent, which represents the reinforcement learning agent
class DQNAgent:

    # Initialize the agent with necessary parameters and settings
    def __init__(self, action_size):

        # If True, the environment will be rendered visually
        self.render = False
        # If True, load pre-trained model weights
        self.load_model = False
        # Define input state shape (width, height, channels)
        self.state_size = (84, 84, 4)
        # Number of possible actions in the environment
        self.action_size = action_size

        # Set up epsilon-greedy exploration settings
        # Initial epsilon for exploration (1 = full exploration)
```

```python
        self.epsilon = 1.0
        # Start and minimum epsilon values
        self.epsilon_start, self.epsilon_end = 1.0, 0.1
        # Total steps over which epsilon decays
        self.exploration_steps = 1000000.
        # Epsilon decay per step
        self.epsilon_decay_step = (self.epsilon_start - self.epsilon_end) / self.exploration_steps



        # Training-related parameters
        # Number of samples per training batch
        self.batch_size = 32
        # Steps before starting training
        self.train_start = 50000
        # Frequency (in steps) to update target network
        self.update_target_rate = 10000
        # Discount rate for future rewards
        self.discount_factor = 0.99
        # Memory buffer for past experiences
        self.memory = deque(maxlen=400000)
        # Steps to perform at the start of episodes
        self.no_op_steps = 30



        # Build main and target Q-networks
        # Initialize the primary network
        self.model = self.build_model()
        # Initialize the target network for stability
        self.target_model = self.build_model()
        # Synchronize weights between primary and target network
        self.update_target_model()

        # Set up the optimizer
        # Define the custom optimizer
        self.optimizer = self.optimizer()

        # Initialize the TensorFlow session for this model
        # Begin a TF session
        self.sess = tf.InteractiveSession()
        # Set Keras backend to this session
        K.set_session(self.sess)
```

```python

        # Initialize tracking variables
        # Variables to hold average max Q-value and loss
        self.avg_q_max, self.avg_loss = 0, 0

        # Set up TensorFlow summary for tracking performance in TensorBoard
        self.summary_placeholders, self.update_ops, self.summary_op = self.setup_summary()
        # Directory for summary logs
        self.summary_writer = tf.summary.FileWriter("summary/breakout_dqn", self.sess.graph)
        # Initialize all variables in the session
        self.sess.run(tf.global_variables_initializer())


        # Load model weights if loading a pre-trained model
        if self.load_model:
            # Load saved weights
            self.model.load_weights("./save_model/breakout_dqn.h5")


    # Define a custom optimizer using Huber loss
    def optimizer(self):
        # Placeholder for action input
        a = K.placeholder(shape=(None,), dtype='int32')
        # Placeholder for expected Q-values
        y = K.placeholder(shape=(None,), dtype='float32')

        # Output predictions from the model
        py_x = self.model.output

        # Create one-hot encoding for actions
        a_one_hot = K.one_hot(a, self.action_size)  # One-hot encode actions
        # Select the Q-value for the action taken
        q_value = K.sum(py_x * a_one_hot, axis=1)


        # Calculate Huber loss
        # Difference between target and predicted Q-value
        error = K.abs(y - q_value)
        # Errors <= 1.0 are squared
        quadratic_part = K.clip(error, 0.0, 1.0)
```

```
        # Errors > 1.0 use absolute error
        linear_part = error - quadratic_part
        # Combine quadratic and linear parts
        loss = K.mean(0.5 * K.square(quadratic_part) + linear_part)

        # Define RMSprop optimizer
        optimizer = RMSprop(lr=0.00025, epsilon=0.01)
        # Get the training updates
        updates = optimizer.get_updates(self.model.trainable_weights, [], loss)
        # Training function
        train = K.function([self.model.input, a, y], [loss], updates=updates)

        return train  # Return training function



    # Define the convolutional neural network model
    def build_model(self):
        # Initialize sequential model
        model = Sequential()
        # First conv layer
        model.add(Conv2D(32, (8, 8), strides=(4, 4), activation='relu', input_shape=self.state_size))
        # Second conv layer
        model.add(Conv2D(64, (4, 4), strides=(2, 2), activation='relu'))
        # Third conv layer
        model.add(Conv2D(64, (3, 3), strides=(1, 1), activation='relu'))
        # Flatten conv output to feed into fully connected layers
        model.add(Flatten())
        # Fully connected layer with 512 units
        model.add(Dense(512, activation='relu'))
        # Output layer for action Q-values
        model.add(Dense(self.action_size))
        # Print model summary
        model.summary()

        return model  # Return the model
```

```python

    # Update the target model to match weights of the main model
    def update_target_model(self):
        # Copy weights to target model
        self.target_model.set_weights(self.model.get_weights())



    # Select an action using epsilon-greedy policy

    def get_action(self, history):
        # Normalize the history input
        history = np.float32(history / 255.0)
        # Explore with probability epsilon
        if np.random.rand() <= self.epsilon:
            # Choose a random action
            return random.randrange(self.action_size)

        else:  # Exploit the learned policy
            # Predict Q-values for actions
            q_value = self.model.predict(history)
            # Choose action with highest Q-value
            return np.argmax(q_value[0])



    # Save experience in memory

    def replay_memory(self, history, action, reward, next_history, dead):
        # Append experience tuple to memory
        self.memory.append((history, action, reward, next_history, dead))



    # Train the model with a batch of experiences

    def train_replay(self):
        # Skip if not enough samples
        if len(self.memory) < self.train_start:
            return

```

```
        if self.epsilon > self.epsilon_end:  # Decrease epsilon over time
            self.epsilon -= self.epsilon_decay_step


        # Sample a mini-batch from memory
        mini_batch = random.sample(self.memory, self.batch_size)

        # Prepare arrays for batch processing
        # Batch of current states
        history = np.zeros((self.batch_size, *self.state_size))
        # Batch of next states
        next_history = np.zeros((self.batch_size, *self.state_size))
        # Array for target Q-values
        target = np.zeros((self.batch_size,))
        # Lists for actions, rewards, and terminal states
        action, reward, dead = [], [], []


        # Process each sample in the mini-batch

        for i in range(self.batch_size):
            # Normalize and store current state
            history[i] = np.float32(mini_batch[i][0] / 255.)
            # Normalize and store next state
            next_history[i] = np.float32(mini_batch[i][3] / 255.)
            # Store action taken
            action.append(mini_batch[i][1])
            # Store reward received
            reward.append(mini_batch[i][2])
            # Store whether episode ended
            dead.append(mini_batch[i][4])



        # Predict Q-values for the next states
        target_value = self.target_model.predict(next_history)

        # Calculate target Q-values for training
        for i in range(self.batch_size):
```

```
            if dead[i]:  # If episode ended, set target to reward only
                target[i] = reward[i]

            else:  # Otherwise, add discounted max future Q-value
                target[i] = reward[i] + self.discount_factor * np.amax(target_value[i])

        # Perform a training step
        # Compute loss and apply gradient updates
        loss = self.optimizer([history, action, target])
        # Accumulate loss for tracking
        self.avg_loss += loss[0]

    # Preprocess the observation to grayscale and resize it
    def preprocess_observation(self, observe):
        # Grayscale and resize
        processed_observe = np.uint8(resize(rgb2gray(observe), (84, 84), mode='constant') * 255)
        # Return processed observation
        return processed_observe
```

Play: (play_dqn_model.py)

```



# Import necessary libraries for environment, random operations, numerical operations, and deep learning

import gym  # Toolkit for creating and interacting with RL environments
import random
import numpy as np
import tensorflow as tf
from skimage.color import rgb2gray
from skimage.transform import resize  # Resize images to desired shape
from keras.models import Sequential  # Keras sequential model for linear stack of layers
from keras.layers import Dense, Flatten  # Fully connected (
```

```
    Dense) and Flatten layers
from keras.layers.convolutional import Conv2D  # 2D convolution layer for image processing
from keras import backend as K  # Keras backend, allows low-level control over TensorFlow


# Define the number of episodes for training or testing the agent
EPISODES = 50000

# Define a class TestAgent to represent the reinforcement learning agent
class TestAgent:

    # Initialize the agent with required parameters and model setup

    def __init__(self, action_size):
        # Input state shape (width, height, frames)
        self.state_size = (84, 84, 4)
        # Number of possible actions for the agent
        self.action_size = action_size
        # Number of steps at the beginning of episodes
        self.no_op_steps = 20



        # Build the model to predict Q-values for actions
        self.model = self.build_model()  # Initialize the model

        # Start a TensorFlow session for the model
        self.sess = tf.InteractiveSession()  # Create an interactive TensorFlow session
        # Set this session for Keras backend
        K.set_session(self.sess)


        # Initialize tracking variables for performance monitoring
        # Average max Q-value and average loss
        self.avg_q_max, self.avg_loss = 0, 0
        # Initialize variables in session
        self.sess.run(tf.global_variables_initializer())
```

```python


    # Define a method to build the neural network model for the agent
    def build_model(self):
        # Initialize a sequential model
        model = Sequential()
        # First convolutional layer with 32 filters
        model.add(Conv2D(32, (8, 8), strides=(4, 4), activation='relu',
                         input_shape=self.state_size))
        # Second convolutional layer with 64 filters
        model.add(Conv2D(64, (4, 4), strides=(2, 2), activation='relu'))
        # Third convolutional layer with 64 filters
        model.add(Conv2D(64, (3, 3), strides=(1, 1), activation='relu'))
        # Flatten the convolutional layer output
        model.add(Flatten())
        # Fully connected layer with 512 units
        model.add(Dense(512, activation='relu'))
        # Output layer to predict Q-values for each action
        model.add(Dense(self.action_size))
        # Print a summary of the model architecture
        model.summary()

        return model  # Return the built model



    # Define a method to choose an action based on the epsilon-greedy policy

    def get_action(self, history):
        # Choose random action with probability 0.01 (exploration)
        if np.random.random() < 0.01:
            # Return a random action
            return random.randrange(3)
        # Normalize history for input
        history = np.float32(history / 255.0)
        # Predict Q-values for each action
        q_value = self.model.predict(history)
        # Return the action with the highest Q-value
        return np.argmax(q_value[0])
```

```python

    # Define a method to load pre-trained weights for the model

    def load_model(self, filename):
        # Load weights from the specified file
        self.model.load_weights(filename)



# Define a function to preprocess the observation (frame) from the environment
def pre_processing(observe):
    # Convert to grayscale, resize, and scale
    processed_observe = np.uint8(
        resize(rgb2gray(observe), (84, 84), mode='constant') * 255)

    return processed_observe  # Return processed observation



# Main function to run the reinforcement learning loop
if __name__ == "__main__":
    # Create the Breakout game environment
    env = gym.make('BreakoutDeterministic-v4')
    # Initialize agent with 3 actions
    agent = TestAgent(action_size=3)
    # Load pre-trained model weights
    agent.load_model("./save_model/breakout_dqn_5.h5")

    # Loop through the defined number of episodes
    for e in range(EPISODES):
        # Flag to track episode completion
        done = False
        # Flag to track if agent lost a life
        dead = False
        # Initialize tracking variables for the episode
        # Set step, score, and starting life count
        step, score, start_life = 0, 0, 5
        # Reset the environment at the start of each episode
        observe = env.reset()

        # Execute 'do nothing' steps at the start of the episode
```

```python
        for _ in range(random.randint(1, agent.no_op_steps)):
            # Execute a 'do nothing' action
            observe, _, _, _ = env.step(1)

        # Preprocess and initialize state history for the
            episode
        # Preprocess initial observation
        state = pre_processing(observe)
        # Stack initial frames
        history = np.stack((state, state, state, state), axis
            =2)
        # Reshape to add batch dimension
        history = np.reshape([history], (1, 84, 84, 4))

        # Loop until the episode is done
        while not done:
            # Render the environment visually
            env.render()
            # Increment the step counter
            step += 1
            # Get action from the agent
            action = agent.get_action(history)

            # Map action from agent to environment's action
                space
            if action == 0:
                # Move paddle left
                real_action = 1
            elif action == 1:
                # Move paddle right
                real_action = 2
            else:
                # Fire the ball
                real_action = 3

            # Reset real action if agent just lost a life
            if dead:
                # Default action after losing a life
                real_action = 1
                # Reset dead flag
                dead = False

            # Take the chosen action and observe the next
                state and reward
            observe, reward, done, info = env.step(
```

```
                real_action)  # Execute action in environment

            # Preprocess the observed frame and update history
            # Preprocess next observation
            next_state = pre_processing(observe)
            # Reshape next state for input
            next_state = np.reshape([next_state], (1, 84, 84, 1))
            # Update history by shifting frames
            next_history = np.append(next_state, history[:, :, :, :3], axis=3)

            # Check if agent lost a life to update state and flags
            if start_life > info['ale.lives']:
                # Set dead flag if life lost
                dead = True
                # Update start life count
                start_life = info['ale.lives']

            # Update the total score
            score += reward
            # Update history for the next step
            history = next_history

            # Print the episode result once the episode ends
            if done:
                # Print episode number and score
                print("episode:", e, "  score:", score)
```

3 References

- https://github.com/rlcode/reinforcement-learning/blob/master/3-atari/1-breakout/breakout_dqn.py
- https://github.com/rlcode/reinforcement-learning/blob/master/3-atari/1-breakout/play_dqn_model.py