

CMPSC 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)

```
1 # Import necessary libraries for the environment, math
   operations, neural network, and image processing
2 import gym # Toolkit for reinforcement learning environments
3 import random
4 import numpy as np
5 import tensorflow as tf
6 from collections import deque
7 from skimage.color import rgb2gray
8 from skimage.transform import resize
9 from keras.models import Sequential
10 from keras.optimizers import RMSprop
11 from keras.layers import Dense, Flatten
12 from keras.layers.convolutional import Conv2D
13 from keras import backend as K
14
15
16 # Define the total number of episodes for training
17 EPISODES = 50000 # Number of episodes for training the agent
18
19 # Define a class DQNAgent, which represents the reinforcement
   learning agent
20 class DQNAgent:
21
22     # Initialize the agent with necessary parameters and
       settings
23     def __init__(self, action_size):
24
25         # If True, the environment will be rendered visually
26         self.render = False
27         # If True, load pre-trained model weights
28         self.load_model = False
29         # Define input state shape (width, height, channels)
30         self.state_size = (84, 84, 4)
31         # Number of possible actions in the environment
32         self.action_size = action_size
33
34         # Set up epsilon-greedy exploration settings
35         # Initial epsilon for exploration (1 = full
           exploration)
```

```

36     self.epsilon = 1.0
37     # Start and minimum epsilon values
38     self.epsilon_start, self.epsilon_end = 1.0, 0.1
39     # Total steps over which epsilon decays
40     self.exploration_steps = 1000000.
41     # Epsilon decay per step
42     self.epsilon_decay_step = (self.epsilon_start - self.
        epsilon_end) / self.exploration_steps
43
44
45
46     # Training-related parameters
47     # Number of samples per training batch
48     self.batch_size = 32
49     # Steps before starting training
50     self.train_start = 50000
51     # Frequency (in steps) to update target network
52     self.update_target_rate = 10000
53     # Discount rate for future rewards
54     self.discount_factor = 0.99
55     # Memory buffer for past experiences
56     self.memory = deque(maxlen=400000)
57     # Steps to perform at the start of episodes
58     self.no_op_steps = 30
59
60
61
62     # Build main and target Q-networks
63     # Initialize the primary network
64     self.model = self.build_model()
65     # Initialize the target network for stability
66     self.target_model = self.build_model()
67     # Synchronize weights between primary and target
        network
68     self.update_target_model()
69
70     # Set up the optimizer
71     # Define the custom optimizer
72     self.optimizer = self.optimizer()
73
74     # Initialize the TensorFlow session for this model
75     # Begin a TF session
76     self.sess = tf.InteractiveSession()
77     # Set Keras backend to this session
78     K.set_session(self.sess)

```

```

79
80     # Initialize tracking variables
81     # Variables to hold average max Q-value and loss
82     self.avg_q_max, self.avg_loss = 0, 0
83
84     # Set up TensorFlow summary for tracking performance
      in TensorBoard
85     self.summary_placeholders, self.update_ops, self.
      summary_op = self.setup_summary()
86     # Directory for summary logs
87     self.summary_writer = tf.summary.FileWriter("summary/
      breakout_dqn", self.sess.graph)
88     # Initialize all variables in the session
89     self.sess.run(tf.global_variables_initializer())
90
91
92     # Load model weights if loading a pre-trained model
93     if self.load_model:
94         # Load saved weights
95         self.model.load_weights("./save_model/
      breakout_dqn.h5")
96
97
98     # Define a custom optimizer using Huber loss
99     def optimizer(self):
100         # Placeholder for action input
101         a = K.placeholder(shape=(None,), dtype='int32')
102         # Placeholder for expected Q-values
103         y = K.placeholder(shape=(None,), dtype='float32')
104
105         # Output predictions from the model
106         py_x = self.model.output
107
108         # Create one-hot encoding for actions
109         a_one_hot = K.one_hot(a, self.action_size) # One-hot
      encode actions
110         # Select the Q-value for the action taken
111         q_value = K.sum(py_x * a_one_hot, axis=1)
112
113
114         # Calculate Huber loss
115         # Difference between target and predicted Q-value
116         error = K.abs(y - q_value)
117         # Errors <= 1.0 are squared
118         quadratic_part = K.clip(error, 0.0, 1.0)

```

```

119         # Errors > 1.0 use absolute error
120         linear_part = error - quadratic_part
121         # Combine quadratic and linear parts
122         loss = K.mean(0.5 * K.square(quadratic_part) +
123                        linear_part)
124
125         # Define RMSprop optimizer
126         optimizer = RMSprop(lr=0.00025, epsilon=0.01)
127         # Get the training updates
128         updates = optimizer.get_updates(self.model.
129                                         trainable_weights, [], loss)
130         # Training function
131         train = K.function([self.model.input, a, y], [loss],
132                           updates=updates)
133
134         return train # Return training function
135
136     # Define the convolutional neural network model
137     def build_model(self):
138         # Initialize sequential model
139         model = Sequential()
140         # First conv layer
141         model.add(Conv2D(32, (8, 8), strides=(4, 4),
142                          activation='relu', input_shape=self.state_size))
143         # Second conv layer
144         model.add(Conv2D(64, (4, 4), strides=(2, 2),
145                          activation='relu'))
146         # Third conv layer
147         model.add(Conv2D(64, (3, 3), strides=(1, 1),
148                          activation='relu'))
149         # Flatten conv output to feed into fully connected
150         # layers
151         model.add(Flatten())
152         # Fully connected layer with 512 units
153         model.add(Dense(512, activation='relu'))
154         # Output layer for action Q-values
155         model.add(Dense(self.action_size))
156         # Print model summary
157         model.summary()
158
159         return model # Return the model

```

```

157
158     # Update the target model to match weights of the main
        model
159     def update_target_model(self):
160         # Copy weights to target model
161         self.target_model.set_weights(self.model.get_weights
            ())
162
163
164
165     # Select an action using epsilon-greedy policy
166
167     def get_action(self, history):
168         # Normalize the history input
169         history = np.float32(history / 255.0)
170         # Explore with probability epsilon
171         if np.random.rand() <= self.epsilon:
172             # Choose a random action
173             return random.randrange(self.action_size)
174
175         else: # Exploit the learned policy
176             # Predict Q-values for actions
177             q_value = self.model.predict(history)
178             # Choose action with highest Q-value
179             return np.argmax(q_value[0])
180
181
182
183     # Save experience in memory
184
185     def replay_memory(self, history, action, reward,
        next_history, dead):
186         # Append experience tuple to memory
187         self.memory.append((history, action, reward,
            next_history, dead))
188
189
190
191     # Train the model with a batch of experiences
192
193     def train_replay(self):
194         # Skip if not enough samples
195         if len(self.memory) < self.train_start:
196             return
197

```

```

198         if self.epsilon > self.epsilon_end: # Decrease
199             epsilon over time
200             self.epsilon -= self.epsilon_decay_step
201
202         # Sample a mini-batch from memory
203         mini_batch = random.sample(self.memory, self.
204             batch_size)
205
206         # Prepare arrays for batch processing
207         # Batch of current states
208         history = np.zeros((self.batch_size, *self.state_size
209             ))
210         # Batch of next states
211         next_history = np.zeros((self.batch_size, *self.
212             state_size))
213         # Array for target Q-values
214         target = np.zeros((self.batch_size,))
215         # Lists for actions, rewards, and terminal states
216         action, reward, dead = [], [], []
217
218         # Process each sample in the mini-batch
219
220         for i in range(self.batch_size):
221             # Normalize and store current state
222             history[i] = np.float32(mini_batch[i][0] / 255.)
223             # Normalize and store next state
224             next_history[i] = np.float32(mini_batch[i][3] /
225                 255.)
226             # Store action taken
227             action.append(mini_batch[i][1])
228             # Store reward received
229             reward.append(mini_batch[i][2])
230             # Store whether episode ended
231             dead.append(mini_batch[i][4])
232
233         # Predict Q-values for the next states
234         target_value = self.target_model.predict(next_history
235             )
236
237         # Calculate target Q-values for training
238         for i in range(self.batch_size):

```



```

237
238         if dead[i]: # If episode ended, set target to
                reward only
239             target[i] = reward[i]
240
241         else: # Otherwise, add discounted max future Q-
                value
242             target[i] = reward[i] + self.discount_factor
                * np.amax(target_value[i])
243
244         # Perform a training step
245         # Compute loss and apply gradient updates
246         loss = self.optimizer([history, action, target])
247         # Accumulate loss for tracking
248         self.avg_loss += loss[0]
249
250     # Preprocess the observation to grayscale and resize it
251     def preprocess_observation(self, observe):
252         # Grayscale and resize
253         processed_observe = np.uint8(resize(rgb2gray(observe)
                , (84, 84), mode='constant') * 255)
254         # Return processed observation
255         return processed_observe

```

Play: (play_dqn_model.py)

```

1
2
3
4 # Import necessary libraries for environment, random
    operations, numerical operations, and deep learning
5
6 import gym # Toolkit for creating and interacting with RL
    environments
7 import random
8 import numpy as np
9 import tensorflow as tf
10 from skimage.color import rgb2gray
11 from skimage.transform import resize # Resize images to
    desired shape
12 from keras.models import Sequential # Keras sequential model
    for linear stack of layers
13 from keras.layers import Dense, Flatten # Fully connected (

```

```

    Dense) and Flatten layers
14 from keras.layers.convolutional import Conv2D # 2D
    convolution layer for image processing
15 from keras import backend as K # Keras backend, allows low-
    level control over TensorFlow
16
17
18 # Define the number of episodes for training or testing the
    agent
19 EPISODES = 50000
20
21 # Define a class TestAgent to represent the reinforcement
    learning agent
22 class TestAgent:
23
24     # Initialize the agent with required parameters and model
        setup
25
26     def __init__(self, action_size):
27         # Input state shape (width, height, frames)
28         self.state_size = (84, 84, 4)
29         # Number of possible actions for the agent
30         self.action_size = action_size
31         # Number of steps at the beginning of episodes
32         self.no_op_steps = 20
33
34
35
36         # Build the model to predict Q-values for actions
37         self.model = self.build_model() # Initialize the
            model
38
39         # Start a TensorFlow session for the model
40         self.sess = tf.InteractiveSession() # Create an
            interactive TensorFlow session
41         # Set this session for Keras backend
42         K.set_session(self.sess)
43
44
45         # Initialize tracking variables for performance
            monitoring
46         # Average max Q-value and average loss
47         self.avg_q_max, self.avg_loss = 0, 0
48         # Initialize variables in session
49         self.sess.run(tf.global_variables_initializer())

```

```

50
51
52     # Define a method to build the neural network model for
        the agent
53     def build_model(self):
54         # Initialize a sequential model
55         model = Sequential()
56         # First convolutional layer with 32 filters
57         model.add(Conv2D(32, (8, 8), strides=(4, 4),
            activation='relu',
58                        input_shape=self.state_size))
59         # Second convolutional layer with 64 filters
60         model.add(Conv2D(64, (4, 4), strides=(2, 2),
            activation='relu'))
61         # Third convolutional layer with 64 filters
62         model.add(Conv2D(64, (3, 3), strides=(1, 1),
            activation='relu'))
63         # Flatten the convolutional layer output
64         model.add(Flatten())
65         # Fully connected layer with 512 units
66         model.add(Dense(512, activation='relu'))
67         # Output layer to predict Q-values for each action
68         model.add(Dense(self.action_size))
69         # Print a summary of the model architecture
70         model.summary()
71
72         return model    # Return the built model
73
74
75
76     # Define a method to choose an action based on the
        epsilon-greedy policy
77
78     def get_action(self, history):
79         # Choose random action with probability 0.01 (
            exploration)
80         if np.random.random() < 0.01:
81             # Return a random action
82             return random.randrange(3)
83         # Normalize history for input
84         history = np.float32(history / 255.0)
85         # Predict Q-values for each action
86         q_value = self.model.predict(history)
87         # Return the action with the highest Q-value
88         return np.argmax(q_value[0])

```

```

89
90     # Define a method to load pre-trained weights for the
      model
91
92     def load_model(self, filename):
93         # Load weights from the specified file
94         self.model.load_weights(filename)
95
96
97
98     # Define a function to preprocess the observation (frame)
      from the environment
99     def pre_processing(observe):
100         # Convert to grayscale, resize, and scale
101         processed_observe = np.uint8(
102             resize(rgb2gray(observe), (84, 84), mode='constant')
              * 255)
103
104         return processed_observe # Return processed observation
105
106
107
108     # Main function to run the reinforcement learning loop
109     if __name__ == "__main__":
110         # Create the Breakout game environment
111         env = gym.make('BreakoutDeterministic-v4')
112         # Initialize agent with 3 actions
113         agent = TestAgent(action_size=3)
114         # Load pre-trained model weights
115         agent.load_model("./save_model/breakout_dqn_5.h5")
116
117         # Loop through the defined number of episodes
118         for e in range(EPISODES):
119             # Flag to track episode completion
120             done = False
121             # Flag to track if agent lost a life
122             dead = False
123             # Initialize tracking variables for the episode
124             # Set step, score, and starting life count
125             step, score, start_life = 0, 0, 5
126             # Reset the environment at the start of each episode
127             observe = env.reset()
128
129             # Execute 'do nothing' steps at the start of the
              episode

```

```

130     for _ in range(random.randint(1, agent.no_op_steps)):
131         # Execute a 'do nothing' action
132         observe, _, _, _ = env.step(1)
133
134     # Preprocess and initialize state history for the
135         episode
136     # Preprocess initial observation
137     state = pre_processing(observe)
138     # Stack initial frames
139     history = np.stack((state, state, state, state), axis
140         =2)
141
142     # Reshape to add batch dimension
143     history = np.reshape([history], (1, 84, 84, 4))
144
145     # Loop until the episode is done
146     while not done:
147         # Render the environment visually
148         env.render()
149         # Increment the step counter
150         step += 1
151         # Get action from the agent
152         action = agent.get_action(history)
153
154         # Map action from agent to environment's action
155         space
156         if action == 0:
157             # Move paddle left
158             real_action = 1
159         elif action == 1:
160             # Move paddle right
161             real_action = 2
162         else:
163             # Fire the ball
164             real_action = 3
165
166         # Reset real action if agent just lost a life
167         if dead:
168             # Default action after losing a life
169             real_action = 1
170             # Reset dead flag
171             dead = False
172
173         # Take the chosen action and observe the next
174         state and reward
175         observe, reward, done, info = env.step(

```

```

    real_action) # Execute action in environment
171
172 # Preprocess the observed frame and update
    history
173 # Preprocess next observation
174 next_state = pre_processing(observe)
175 # Reshape next state for input
176 next_state = np.reshape([next_state], (1, 84, 84,
    1))
177 # Update history by shifting frames
178 next_history = np.append(next_state, history[:,
    :, :, :3], axis=3)
179
180 # Check if agent lost a life to update state and
    flags
181 if start_life > info['ale.lives']:
182     # Set dead flag if life lost
183     dead = True
184     # Update start life count
185     start_life = info['ale.lives']
186
187 # Update the total score
188 score += reward
189 # Update history for the next step
190 history = next_history
191
192 # Print the episode result once the episode ends
193 if done:
194     # Print episode number and score
195     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