CSE 584 – MACHINE LEARNING: TOOLS AND ALGORITHMS

HOMEWORK-2

Divya Navuluri
PSU ID: 903661629

ABSTRACT THAT PROVIDES A HIGH-LEVEL OVERVIEW OF THE REINFORCEMENT LEARNING CODE'S PURPOSE AND THE OVERALL PROCESS:

I have taken the below code from the online GitHub. The Reinforcement learning code covers the implementation of a Deep Q-Network with TensorFlow and Keras that solves reinforcement learning tasks by letting an agent learn how to select optimal policies while interacting with an environment. The DQN is built upon off-policy Q-learning, where it uses a deep neural network in place of tabular Q-value approximation for the selection of actions. It contains two major components in the network: a primary model for training and making action decisions, and a target model to give stable Q-value estimates during training.

The DQN class is initiated with several parameters, state dimensions, action space, discount factor often known as gamma, exploration rate known as epsilon, and learning rate. There are two hidden layers in the model architecture, both with ReLU as their activation functions. This enables the network to learn complex patterns in the input state space. Action selection is based on an epsilon-greedy policy, which forms a good trade-off between exploration and exploitation for optimal learning.

The training process consists of replaying experiences from a memory buffer, where mini-batches are sampled to update the weights of the main model, based on predicted Q-values and corresponding rewards. Then, it comes to a soft update mechanism for synchronizing the target model with the primary one in order to ensure stability during training. The exploration rate grows more conservatory over time in order to gradually shift the agent's behavior from exploration toward exploitation.

Besides, it includes functions to save model architecture and weights that would allow the saving and reusability of the trained DQN model. Overall, this implementation serves as a foundational framework for applying deep reinforcement learning techniques to various decision-making problems.

CORE SECTION OF THE REINFORCEMENT LEARNING IMPLEMENTATION CODE:

```python
import numpy as np
from keras.models import Sequential
from keras.models import model_from_json
from keras.layers import Dense, Activation
from keras import optimizers
from keras import backend as K
import tensorflow as tf
from random import random, randrange

# 1.DQN Class Initialization and Model Creation
# Deep Q Network (DQN) class implementing off-policy Q-learning with deep neural networks
class DQN:

    def __init__(self,
                 input_dim,              # Number of inputs for the DQN network
                 action_space,           # Size of action space
                 gamma=0.99,             # Discount factor for future rewards, balancing immediate and future rewards
                 epsilon=1,              # Initial epsilon
                 epsilon_min=0.01,       # Minimum epsilon value to maintain some exploration in the long term
                 epsilon_decay=0.999,    # Decay rate of epsilon after each action taken to reduce exploration over time
                 learning_rate=0.00025,  # Learning rate for training the neural network
                 tau=0.125,              # Soft update factor for the target network, controls how fast it tracks the main model
                 model=None,             # Placeholder for the main DQN model
                 target_model=None,      # Placeholder for the target DQN model used for stable Q-learning targets
                 sess=None):             # TensorFlow session

        # Initialize parameters for the DQN
        self.input_dim = input_dim        # Set the input dimension for the DQN
        self.action_space = action_space  # Set the size of the action space
        self.gamma = gamma                # Set the discount factor
        self.epsilon = epsilon            # Initialize epsilon for exploration
        self.epsilon_min = epsilon_min    # Set the minimum epsilon for exploration
        self.epsilon_decay = epsilon_decay # Set the decay rate for epsilon
        self.learning_rate = learning_rate # Set the learning rate for model training
        self.tau = tau                     # Set tau for soft updates of the target network

        # Create the main and target DQN models
        self.model = self.create_model()  # Main DQN model for training and action selection
        self.target_model = self.create_model()  # Target DQN model to stabilize learning

        # TensorFlow configuration for GPU memory optimization
        config = tf.compat.v1.ConfigProto()  # Create a TensorFlow configuration object
        config.gpu_options.allow_growth = True  # Allow the GPU memory to grow as needed to prevent allocation issues
        self.sess = tf.compat.v1.Session(config=config)  # Create a TensorFlow session with the specified configuration
        K.set_session(self.sess)  # Set the created session as the backend session for Keras
        self.sess.run(tf.compat.v1.global_variables_initializer())  # Initialize global variables in TensorFlow

    def create_model(self):
        model = Sequential()  # Initialize a sequential model for easy stacking of layers
        model.add(Dense(300, input_dim=self.input_dim))  # First hidden layer with 300 neurons, taking the input dimension
        model.add(Activation('relu'))  # ReLU activation function introduces non-linearity

        model.add(Dense(300))  # Second hidden layer with 300 neurons
        model.add(Activation('relu'))  # ReLU activation function for the second hidden layer

        model.add(Dense(self.action_space))  # Output layer with a number of neurons equal to the action space size
        model.add(Activation('linear'))  # Linear activation function for producing Q-values

        # Define the optimizer and compile the model
        sgd = optimizers.SGD(lr=self.learning_rate, decay=1e-6, momentum=0.95)  # Stochastic Gradient Descent optimizer
        model.compile(optimizer=sgd, loss='mse')  # Compile the model with Mean Squared Error loss function for Q-learning

        return model  # Return the compiled model for use in training and action selection
```

```python

    # 2. Action Selection
    def act(self, state):
        # Selects an action based on an epsilon-greedy policy
        a_max = np.argmax(self.model.predict(state.reshape(1, len(state))))  # Get action with maximum Q-value
        if random() < self.epsilon:  # epsilon-greedy decision with probability epsilon, choose random action
            a_chosen = randrange(self.action_space)  # Randomly select an action
        else:
            a_chosen = a_max  # Choose action with the highest Q-value otherwise
        return a_chosen  # Return the selected action

    # 3. replay Method and training the model
    def replay(self, samples, batch_size):
        # Trains the DQN model on a mini-batch of experiences
        inputs = np.zeros((batch_size, self.input_dim))  # Initialize batch input array
        targets = np.zeros((batch_size, self.action_space))  # Initialize batch target Q-values array

        for i in range(batch_size):  # Loop through each sample in the batch
            state = samples[0][i, :]  # Get current state from samples
            action = samples[1][i]  # Get action taken from samples
            reward = samples[2][i]  # Get reward received for action
            new_state = samples[3][i, :]  # Get resulting state from samples
            done = samples[4][i]  # Check if this state is terminal

            inputs[i, :] = state  # Set input as current state for batch
            targets[i, :] = self.target_model.predict(state.reshape(1, len(state)))  # Predict current Q-values for state

            if done:  # If terminal state, set target as immediate reward
                targets[i, action] = reward  # Assign the reward to the Q-value of the action taken
            else:
                Q_future = np.max(self.target_model.predict(new_state.reshape(1, len(new_state))))  # Predict future max Q-value
                targets[i, action] = reward + Q_future * self.gamma  # Set target Q-value using discounted future reward

        # Train the model on the current batch of inputs and target Q-values
        loss = self.model.train_on_batch(inputs, targets)

    # 4. target_train Method
    def target_train(self):
        # Updates the target model with weights from the main model using a soft update
        weights = self.model.get_weights()  # Get weights from main DQN model
        target_weights = self.target_model.get_weights()  # Get weights from target DQN model
        for i in range(len(target_weights)):  # Update each layer's weights
            target_weights[i] = weights[i] * self.tau + target_weights[i] * (1 - self.tau)  # Soft update rule

        self.target_model.set_weights(target_weights)  # Set updated weights in target model

    # 5. Epsilon Update
    def update_epsilon(self):
        # Reduces the epsilon to encourage exploitation over time
        self.epsilon = self.epsilon * self.epsilon_decay  # Decay epsilon
        self.epsilon = max(self.epsilon_min, self.epsilon)  # Ensure epsilon does not fall below minimum value

    # 6. Model Saving
    def save_model(self, path, model_name):
        # Saves the model structure and weights to disk for later usage
        model_json = self.model.to_json()  # Serialize model to JSON format
        with open(path + model_name + ".json", "w") as json_file:  # Open a file to save the model structure
            json_file.write(model_json)  # Save JSON structure
        self.model.save_weights(path + model_name + ".h5")  # Save model weights in HDF5 format
        print("Saved model to disk")  # Print confirmation of save
```

1. DQN Class Initialization and Model Creation: The DQN class implements a Deep Q-Network, which uses off-policy Q-learning with neural networks. It initializes parameters like input dimensions, action space, discount factor, and exploration settings, and creates both a primary model for choosing actions and a target model for stable training. This is a two-hidden-layer sequence model, compiled with Stochastic Gradient Descent and mean squared error loss to have effective learning.

2. Action Selection: Act method adopts an epsilon-greedy strategy in picking an action, which balances exploration and exploitation. It predicts the Q-values for all the actions, picks a random action if the generated number is less than epsilon, or otherwise picks the action with the highest Q-value. This way, while making effective decisions, the agent can still explore new actions.

3. replay method and training the model: The replay method trains the model using mini-batches from experience replay memory. It prepares the input states and target Q-values, retrieves the necessary information for each sample, and calculates target values based on the immediate rewards and future maximum Q-values. The model is then updated with efficient mini-batch training, returning the loss for this iteration.

4. target_train Method: The target_train method performs a soft update of the target model weights. It pulls the weights from the main and target models then sets the target model weights to be a mix of its current weights and those of the main model, controlled by a specified tau. parameter.

5. Epsilon Update: The update_epsilon method decreases the rate of exploration epsilon in such a way that, increasingly, the more the agent is trained, the exploitation of learned Q-values is favored. It decays epsilon by a factor, while never letting it drop below a minimum value, trying to encourage more confident action selections over time.

6. Model Saving: The save_model method allows for saving the architecture along with the weights of DQN. This serializes the model architecture in JSON format and saves it, along with the model weights in HDF5 format, confirming that saving has been successful for later retrieval and use.