VENKATA SESH TEJ MATTA : vmm5481@psu.edu **CSE584: HOMEWORK-2 REPORT**

Title: Deep Q-Learning Solution for Atari Breakout

---

1. Abstract:

The goal of this notebook is to implement a Deep Q-Learning (DQN) algorithm to train an agent to play the Atari game "Breakout" using OpenAI Gym. DQN, a model-free reinforcement learning approach, uses a Convolutional Neural Network (CNN) as a function approximator to predict Q-values for each possible action in a given state. Key elements in the implementation include:

- **State Processing**: The raw game frames are converted to grayscale, cropped, and resized to an 84x84 image for input into the network. This reduces computational requirements and enhances learning efficiency.
- **Q-Value Estimator (Q-Network)**: This CNN-based model predicts Q-values for each action, helping the agent choose actions based on expected rewards.
- **Target Network**: A secondary network, identical to the Q-network, is updated periodically. This stabilizes training by providing a consistent set of Q-values for temporal difference updates.
- **Experience Replay**: A buffer stores gameplay experiences, which are later sampled in mini-batches for training. This approach breaks correlations between experiences and improves sample efficiency.
- **Training Loop**: The agent plays multiple episodes, following an epsilon-greedy policy that balances exploration and exploitation. The Q-network is updated based on rewards received from actions, gradually improving the agent's performance.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For

- **End For**

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

$$L = \frac{1}{2}[\underbrace{r + max_{a'} Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

2. Commented out code sections (core code)

a) State Processor Class

The StateProcessor class preprocesses game frames to reduce computational complexity and facilitate learning. It converts RGB images to grayscale, crops, and resizes them.

```python
class StateProcessor():
    """
    This processes a raw Atari image to gray scale and resizes it to an 84x84 frame.

    """
    def __init__(self):
        # TensorFlow variable scope for encapsulating processing operations
        with tf.variable_scope("state_processor"):
            # Placeholder for the input state, which is an RGB image (210x160x3)
            self.input_state = tf.placeholder(shape=[210, 160, 3], dtype=tf.uint8)
            # Converting RGB to grayscale for simpler processing
            self.output = tf.image.rgb_to_grayscale(self.input_state)
            # Crop the image to focus on the main play area
            self.output = tf.image.crop_to_bounding_box(self.output, 34, 0, 160, 160)
            # Resize the cropped grayscale image to 84x84 for CNN input
            self.output = tf.image.resize_images(self.output, [84, 84], method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
            # Removing the single color channel dimension
            self.output = tf.squeeze(self.output)

    def process(self, sess, state):
        # Process the input state through the defined operations
        return sess.run(self.output, { self.input_state: state })
```

b) Q-Value Estimator Class: The Estimator class defines a CNN for estimating Q-values. It serves as both the main Q-network and the target network.

```python
class Estimator():

    """
    A CNN-based Q-value estimator used for both Q-network and target network.
    """
    def __init__(self, scope="estimator", summaries_dir=None):
        self.scope = scope
        # Initializing a summary writer for logging if summaries_dir is specified
        if summaries_dir:
            summary_dir = os.path.join(summaries_dir, "summaries_{}".format(scope))
            self.summary_writer = tf.summary.FileWriter(summary_dir)

        # Building the model with CNN architecture
        self._build_model()

    def _build_model(self):
        # Placeholder for state input (84x84 grayscale frames, stacked)
        self.X_pl = tf.placeholder(shape=[None, 84, 84, 4], dtype=tf.uint8, name="X")
        # Placeholder for TD target values for loss calculation
        self.y_pl = tf.placeholder(shape=[None], dtype=tf.float32, name="y")
        # Placeholder for action indices for chosen actions
        self.actions_pl = tf.placeholder(shape=[None], dtype=tf.int32, name="actions")

        # Normalize the input images
        X = tf.to_float(self.X_pl) / 255.0

        # Convolutional layers extract features from the input images
        conv1 = tf.contrib.layers.conv2d(X, 32, 8, 4, activation_fn=tf.nn.relu)
        conv2 = tf.contrib.layers.conv2d(conv1, 64, 4, 2, activation_fn=tf.nn.relu)
        conv3 = tf.contrib.layers.conv2d(conv2, 64, 3, 1, activation_fn=tf.nn.relu)

        # Flatten the final conv layer output for fully connected layer
        flattened = tf.contrib.layers.flatten(conv3)
        fc1 = tf.contrib.layers.fully_connected(flattened, 512)
        # Output layer: Q-values for each action
        self.predictions = tf.contrib.layers.fully_connected(fc1, len(VALID_ACTIONS))

        # Gather the predictions for the specific actions chosen
        gather_indices = tf.range(tf.shape(self.X_pl)[0]) * tf.shape(self.predictions)[1] + self.actions_pl
        self.action_predictions = tf.gather(tf.reshape(self.predictions, [-1]), gather_indices)

        # Calculating the loss using squared difference (TD error)
        self.loss = tf.reduce_mean(tf.squared_difference(self.y_pl, self.action_predictions))

        # RMSProp optimizer as recommended in the DQN paper
        self.optimizer = tf.train.RMSPropOptimizer(0.00025, 0.99, 0.0, 1e-6)
        # Training operation to minimize loss
        self.train_op = self.optimizer.minimize(self.loss, global_step=tf.contrib.framework.get_global_step())
```

c) Deep Q-Learning Function : The main training function, deep_q_learning, runs the episodes, updates the Q-values, and manages experience replay.

```
def deep_q_learning(sess, env, q_estimator, target_estimator, state_processor, num_episodes, experiment_dir):
    """
    Main training function implementing the Deep Q-Learning algorithm.

    Args:
        sess: TensorFlow session
        env: OpenAI Gym environment
        q_estimator: Q-network for estimating action values
        target_estimator: Target Q-network, updated periodically for stability
        state_processor: Object for processing and resizing environment states
        num_episodes: Number of training episodes
        experiment_dir: Directory path for saving logs and model checkpoints
    """
    # Initialize the replay memory, which will store past experiences for experience replay
    replay_memory = []

    # Retrieve the current global step to track progress in training
    total_t = sess.run(tf.contrib.framework.get_global_step())

    # Create a policy function that selects actions using an epsilon-greedy strategy
    policy = make_epsilon_greedy_policy(q_estimator, len(VALID_ACTIONS))

    # Populate the replay memory with initial experiences
    # Reset the environment to get the initial state
    state = env.reset()

    # Process the initial state (resize and grayscale conversion) for consistency in the network input
    state = state_processor.process(sess, state)

    # Stack the processed state 4 times to provide temporal context (e.g., movement over frames)
    state = np.stack([state] * 4, axis=2)

    # Start the training loop, iterating through each episode
    for i_episode in range(num_episodes):

        # Reset environment at the beginning of each episode to get a new initial state
        state = env.reset()

        # Process the new state for input into the Q-network
        state = state_processor.process(sess, state)

        # Stack the processed state 4 times to retain temporal information over frames
        state = np.stack([state] * 4, axis=2)

        # Select an action using epsilon-greedy policy, balancing exploration and exploitation
        action_probs = policy(sess, state, epsilon)

        # Choosing an action based on the calculated probabilities
        action = np.random.choice(np.arange(len(action_probs)), p=action_probs)

        # Checking if the replay memory is large enough to begin training
        if len(replay_memory) > batch_size:
        # Randomly sample a mini-batch from replay memory for training the Q-network
            samples = random.sample(replay_memory, batch_size)

        # Unpack the sample into batches for states, actions, rewards, next states, and done flags
            states_batch, action_batch, reward_batch, next_states_batch, done_batch = map(np.array, zip(*samples))

        # Perform a training update on the Q-network with the current mini-batch

            loss = q_estimator.update(sess, states_batch, action_batch, targets_batch)
```

REFERENCES:

1. https://github.com/dennybritz/reinforcement-learning/blob/master/DQN/Deep%20Q%20Learning%20Solution.ipynb

2. https://github.com/SinanGncgl/Deep-Q-Network-AtariBreakoutGame

3. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). "Human-level control through deep reinforcement learning." *Nature*, 518(7540), 529-533.

4. Lapan, M. (2018). *Deep Reinforcement Learning Hands-On*. Packt Publishing.