Deep Q Network (DQN) Implementation with Atari Breakout

I. Abstract

1.1 Overview

This project implements a Deep Q-Network (DQN) for playing Atari games, specifically focusing on the game "Breakout." The implementation includes enhancements to the basic DQN structure, such as Dueling Network Architectures and Double Q-Learning, which make the agent's learning process more efficient and robust. The experiment setup is designed to run game environments on multiple processes for efficient sampling. Each component of the DQN in this project is well modularized to facilitate understanding and application of DQN in practical reinforcement learning scenarios.

1.2 Model Architecture

The DQN model employs a convolutional neural network (CNN) that processes the state inputs from the environment. These states are typically frames or images that describe the current situation in the environment. The network then outputs Q-values for each possible action, which represent the expected future rewards that can be obtained by taking each action in the current state.

1.3 Dueling Network Architecture

A key feature of the model is the dueling network architecture, which consists of two separate estimators:

1. Value Function $V(s)$: Estimates how good it is to be in a given state s
2. Advantage Function $A(s, a)$: Estimates the advantage of taking a specific action a over others in state s

The final Q-value for each action is computed using the equation:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

Where $|\mathcal{A}|$ is the number of possible actions. This equation adjusts the advantage values so that they are zero-centered, which stabilizes the training process by reducing the variance of the advantage estimates.

1.4 Training Process

1. **Sampling**: Actions are sampled using an ϵ-greedy policy, where ϵ is gradually reduced. This means the model mostly exploits the best-known actions while occasionally exploring other actions to discover potentially better strategies.
2. **Experience Replay**: Interactions with the environment (state transitions) are stored in a replay buffer. Training samples are drawn from this buffer, which helps to break the correlation between consecutive training samples and stabilizes learning.
3. **Double Q-Learning**: To mitigate the overestimation of Q-values, two networks are maintained: the primary network and a target network. The target network's weights are periodically updated with the weights from the primary network, providing stable targets for training.
4. **Loss Function**: The loss is computed as the difference between the current predicted Q-values and the target Q-values, which are adjusted by the reward received and the discounted highest Q-value of the next state, as predicted by the target network.
5. **Prioritized Experience Replay**: The replay buffer is enhanced with a prioritization mechanism that more frequently samples transitions with high temporal difference (TD) errors, which are indicative of significant learning potential.

By continuously interacting with the environment, updating the network weights, and adjusting the exploration/exploitation balance, the model progressively learns to maximize rewards over time, ideally converging to an optimal policy that dictates the best actions to take from any given state.

II. Implementation Code Core Section Annotation

In this section, I will provide screenshots of the code, along with comments on the core sections of the project. At the beginning of each function, there is a short description (orange color).

2.1 Model.py

2.1.1 Import Section

```
import torch
from torch import nn
from labml_helpers.module import Module
```

2.1.2 Class Model

model.py > Model > __init__

```python
class Model(Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Sequential(
            # First convolutional layer: takes a 84x84 frame with 4 channels as input
            # and outputs a 20x20 frame with 32 channels
            nn.Conv2d(in_channels=4, out_channels=32, kernel_size=8, stride=4),
            nn.ReLU(),

            # Second convolutional layer: reduces the dimension from 20x20 to 9x9 with 64 output channels
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),
            nn.ReLU(),

            # Third convolutional layer: processes the 9x9 frame to output a 7x7 frame with 64 channels
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1),
            nn.ReLU(),
        )

        # A fully connected layer takes the flattened frame from third convolution layer,
        # and outputs 512 features
        self.lin = nn.Linear(in_features=7 * 7 * 64, out_features=512)
        self.activation = nn.ReLU()

        # This head gives the state value $V$
        self.state_value = nn.Sequential(
            nn.Linear(in_features=512, out_features=256),
            nn.ReLU(),
            nn.Linear(in_features=256, out_features=1),
        )
        # This head gives the action value $A$
        self.action_value = nn.Sequential(
            nn.Linear(in_features=512, out_features=256),
            nn.ReLU(),
            nn.Linear(in_features=256, out_features=4),
        )
```

```python
    def forward(self, obs: torch.Tensor):
        # Convolution
        h = self.conv(obs)

        # Reshape for linear layers
        h = h.reshape((-1, 7 * 7 * 64))

        # Pass the reshaped tensor 'h' through a linear layer followed by an activation function.
        h = self.activation(self.lin(h))

        # Compute the advantage of each action using the 'action_value' network.
        action_value = self.action_value(h)

        # Compute the value of being in the given state using the 'state_value' network.
        state_value = self.state_value(h)

        # Center the action values by subtracting the mean action value.
        action_score_centered = action_value - action_value.mean(dim=-1, keepdim=True)

        # Calculate the Q-value for each action by combining the state value and the centered action values.
        q = state_value + action_score_centered

        return q
```

2.2 Experiment.py

2.2.1 Import Section

```python
import numpy as np
import torch
from labml import tracker, experiment, logger, monit
from labml.internal.configs.dynamic_hyperparam import FloatDynamicHyperParam
from labml_helpers.schedule import Piecewise
from labml_nn.rl.dqn import QFuncLoss
from labml_nn.rl.dqn.model import Model
from labml_nn.rl.dqn.replay_buffer import ReplayBuffer
from labml_nn.rl.game import Worker

# Select the computing device
if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = torch.device("cpu")


def obs_to_torch(obs: np.ndarray) -> torch.Tensor:
    """Scale observations from `[0, 255]` to `[0, 1]`"""
    return torch.tensor(obs, dtype=torch.float32, device=device) / 255.
```

2.2.2 Class Trainer

2.2.2.1 Init Function

```python
class Trainer:
    def __init__(self, *, updates: int, epochs: int, n_workers: int, worker_steps: int, mini_batch_size: int,
                update_target_model: int, learning_rate: FloatDynamicHyperParam):

        self.n_workers = n_workers  # Number of parallel worker processes
        self.worker_steps = worker_steps  # Number of steps each worker takes before updating the model
        self.train_epochs = epochs  # Number of epochs to train with each batch of data
        self.updates = updates  # Total number of model updates to perform
        self.mini_batch_size = mini_batch_size  # Size of each training batch
        self.update_target_model = update_target_model  # Interval at which to update the target model
        self.learning_rate = learning_rate  # Dynamic adjustment of learning rate


        # exploration as a function of updates
        self.exploration_coefficient = Piecewise(
            [
                (0, 1.0),
                (25_000, 0.1),
                (self.updates / 2, 0.01)
            ], outside_value=0.01)

        # beta for replay buffer as a function of updates
        self.prioritized_replay_beta = Piecewise(
            [
                (0, 0.4),
                (self.updates, 1)
            ], outside_value=1)

       # Initialize replay buffer with specific capacity (power of 2) and alpha (0.6)
        self.replay_buffer = ReplayBuffer(2 ** 14, 0.6)

        # Model for sampling and training
        self.model = Model().to(device)

        # Target model used in Double Q-learning
        self.target_model = Model().to(device)
```

```python

        # Initialize workers for parallel environment interaction
        self.workers = [Worker(47 + i) for i in range(self.n_workers)]

        # Initial observations from the environment
        self.obs = np.zeros((self.n_workers, 4, 84, 84), dtype=np.uint8)

        # Initialize the environment in each worker
        for worker in self.workers:
            worker.child.send(("reset", None))

        # Get the initial observations from each worker
        for i, worker in enumerate(self.workers):
            self.obs[i] = worker.child.recv()

        # Initialize the loss function with a discount factor
        self.loss_func = QFuncLoss(0.99)

        # Initialize the optimizer with a learning rate
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=2.5e-4)
```

2.2.2.2 _Sample_action Function

```python
    def _sample_action(self, q_value: torch.Tensor, exploration_coefficient: float):
        """
        Sample an action using an epsilon-greedy strategy where epsilon is the exploration coefficient.
        """

        # Sampling doesn't need gradients
        # Disable gradient calculations for efficiency
        with torch.no_grad():
            # Sample the action with highest Q-value. This is the greedy action.
            greedy_action = torch.argmax(q_value, dim=-1)
            # Uniformly sample and action
            random_action = torch.randint (variable) greedy_action: Tensor shape, device=q_value.device)
            # Whether to chose greedy ac
            is_choose_rand = torch.rand(greedy_action.shape, device=q_value.device) < exploration_coefficient
            # Pick the action based on epsilon-greedy strategy `is_choose_rand`
            return torch.where(is_choose_rand, random_action, greedy_action).cpu().numpy()
```

2.2.2.3 Sample Function

```python
    def sample(self, exploration_coefficient: float):
        """
        Collect data samples from the environment using the current policy and update the replay buffer.
        """
        # Disable gradient calculations for efficiency
        with torch.no_grad():
            # Perform sampling for defined number of worker steps
            for t in range(self.worker_steps):
                # Get Q_values for the current observation
                q_value = self.model(obs_to_torch(self.obs))
                # Sample actions
                actions = self._sample_action(q_value, exploration_coefficient)

                # Run sampled actions in each worker and update observations
                for w, worker in enumerate(self.workers):
                    worker.child.send(("step", actions[w]))

                # Collect information from each worker
                for w, worker in enumerate(self.workers):
                    # Get results after executing the actions
                    next_obs, reward, done, info = worker.child.recv()

                    # Add transition to replay buffer
                    self.replay_buffer.add(self.obs[w], actions[w], reward, next_obs, done)

                    # update episode information.
                    # collect episode info, which is available if an episode finished;
                    #  this includes total reward and length of the episode -
                    #  look at `Game` to see how it works.
                    if info:
                        tracker.add('reward', info['reward'])
                        tracker.add('length', info['length'])

                    # update current observation
                    self.obs[w] = next_obs
```

2.2.2.4 Train Function

```python
    def train(self, beta: float):
        """
        Train the model using samples from the replay buffer and update the model and target networks.
        """
        # Perform training for defined number of epochs
        for _ in range(self.train_epochs):
            # Sample from priority replay buffer
            samples = self.replay_buffer.sample(self.mini_batch_size, beta)
            # Get the predicted Q-value
            q_value = self.model(obs_to_torch(samples['obs']))

            # Get the Q-values of the next state for [Double Q-learning](index.html).
            # Gradients shouldn't propagate for these
            with torch.no_grad():
                # Double Q-learning Q(s;theta_i)
                double_q_value = self.model(obs_to_torch(samples['next_obs']))
                # Target Q-value for Double Q-learning Q(s;theta_i^-)
                target_q_value = self.target_model(obs_to_torch(samples['next_obs']))
            # Calculate Temporal Difference (TD) errors (delta) and loss L(theta)
            td_errors, loss = self.loss_func(q_value,
                                             q_value.new_tensor(samples['action']),
                                             double_q_value, target_q_value,
                                             q_value.new_tensor(samples['done']),
                                             q_value.new_tensor(samples['reward']),
                                             q_value.new_tensor(samples['weights']))

            # Calculate priorities for replay buffer, p_i = |delta_i| + epsilon
            new_priorities = np.abs(td_errors.cpu().numpy()) + 1e-6
            # Update replay buffer priorities
            self.replay_buffer.update_priorities(samples['indexes'], new_priorities)

            # Set learning rate
            for pg in self.optimizer.param_groups:
                pg['lr'] = self.learning_rate()
```

```python
            # Zero out the previously calculated gradients
            self.optimizer.zero_grad()
            # Calculate gradients
            loss.backward()
            # Clip gradients
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=0.5)
            # Update parameters based on gradients
            self.optimizer.step()
```

2.2.2.5 Run_training_loop Function

```python
    def run_training_loop(self):
        """
        Execute the main training loop, managing exploration, training, and updates to the target network.
        """

        # Last 100 episode information
        tracker.set_queue('reward', 100, True)
        tracker.set_queue('length', 100, True)

        # Copy to target network initially
        self.target_model.load_state_dict(self.model.state_dict())
        # Main update loop
        for update in monit.loop(self.updates):
            # Calculate current exploration rate epsilon
            exploration = self.exploration_coefficient(update)
            tracker.add('exploration', exploration)
            # Calculate current beta for the replay buffer beta
            beta = self.prioritized_replay_beta(update)
            tracker.add('beta', beta)

            # Sample with current policy
            self.sample(exploration)

            # Start training after the buffer is full
            if self.replay_buffer.is_full():
                # Train the model
                self.train(beta)

                # Update the target network periodically
                if update % self.update_target_model == 0:
                    self.target_model.load_state_dict(self.model.state_dict())

```

```python
            # Save tracked indicators.
            tracker.save()
            # Add a new line to the screen periodically
            if (update + 1) % 1_000 == 0:
                logger.log()
```

2.2.2.6 Destroy Function

```python
    def destroy(self):
        """
        Clean up resources and stop worker processes at the end of training.
        """
        # Send close signal to each worker
        for worker in self.workers:
            worker.child.send(("close", None))
```

2.2.3 Main Function

```python
def main():
    # Create the experiment
    experiment.create(name='dqn')

    # Configurations
    configs = {
        # Number of updates
        'updates': 1_000_000,
        # Number of epochs to train the model with sampled data.
        'epochs': 8,
        # Number of worker processes
        'n_workers': 8,
        # Number of steps to run on each process for a single update
        'worker_steps': 4,
        # Mini batch size
        'mini_batch_size': 32,
        # Target model updating interval
        'update_target_model': 250,
        # Learning rate.
        'learning_rate': FloatDynamicHyperParam(1e-4, (0, 1e-3)),
    }

    # Apply configurations to the experiment
    experiment.configs(configs)

    # Initialize the trainer
    m = Trainer(**configs)
    # Run and monitor the experiment
    with experiment.start():
        m.run_training_loop()
    # Stop the workers
    m.destroy()


# Run the main function
if __name__ == "__main__":
    main()
```

Reference

Labmlai. (n.d.). *annotated_deep_learning_paper_implementations/labml_nn/rl/dqn at master · labmlai/annotated_deep_learning_paper_implementations*. GitHub. https://github.com/labmlai/annotated_deep_learning_paper_implementations/tree/master/labml_nn/rl/dqn