EEE 448/548 - Project Report

Playing Atari with Deep Reinforcement Learning

Abdallah Zaid Alkilani & Mustafa Hakan Kara

Abstract—In this project, we examined the Deep Q-Network (DQN) that pioneered use of CNNs for reinforcement learning, directly learning from high-dimensional sensory inputs like raw pixel data. Employing a modified Q-learning algorithm, this model was applied to seven Atari 2600 games, demonstrating its versatility without needing architecture-specific adjustments. Notably, it demonstrated superior performance over existing methods in most games and even outperformed human experts in several instances. Our project focused on replicating the original experiments from the DQN paper and comparing our results with the initial findings, while critically evaluating the paper, mainly the quality and communication of the findings and methods, as this paper is purely on the experimental side with little theortical involvement. The code is available on our GitHub Repository.

Introduction

The integration of deep learning into reinforcement learning (RL) has been a significant advancement, particularly in learning control policies from high-dimensional sensory inputs. This development was notably advanced by a 2013 study that introduced a deep learning model for RL, capable of processing raw pixel data. This model, a convolutional neural network (CNN) trained with a Q-learning variant, was distinctive in its ability to directly interpret raw sensory data without relying on pre-processed features or game-specific information.

The original paper demonstrated the model's performance across seven Atari 2600 games within the Arcade Learning Environment (ALE), highlighting its robustness without requiring adjustments to its architecture or learning algorithm. Its performance exceeded previous methods in most games and, in some cases, surpassed human experts. This accomplishment addressed several challenges inherent in RL, such as learning from delayed and sparse rewards, managing correlated state sequences, and adapting to changing data distributions.

In our term project, we have replicated these experiments to validate the findings. We meticulously followed the methodology described in the original paper to reproduce the experimental results and conducted a comparison with the reported outcomes. This exercise provided practical insights into the application of deep learning techniques in RL and the replication of a computational experiment.

Related Work

Revisiting the development of RL, the seminal TD-gammon project [9] stands out as an early success story, mastering backgammon through a model-free RL algorithm akin to Q-learning. However, subsequent attempts to apply similar methods to other strategic games like chess or Go yielded limited success [6], suggesting that the specific characteristics of backgammon, such as the stochastic nature of dice rolls, played a crucial role in TD-gammon's success.

The integration of model-free RL algorithms with non-linear function approximators, such as in Q-learning, initially faced challenges due to divergence issues [10]. This led to a focus on linear function approximators, offering more stability in convergence. However, the advent of deep learning rekindled interest in combining deep neural networks with RL. This interest is evident in works that employed deep neural networks for environment estimation [8] and addressed the divergence problems in Q-learning [3].

A significant precursor to the Deep Q-Network (DQN) paper [4] is Neural Fitted Q-Learning (NFQ) [7]. NFQ optimized loss functions using RPROP, contrasting DQN's use of stochastic gradient descent updates. The utilization of the Atari 2600 emulator as an RL platform [1] further laid the groundwork for the development and evaluation of RL algorithms in complex environments.

Background

In RL, a central concept is the estimation of the action-value function, which is typically approached using the Bellman equation in an iterative manner. The iterative update can be formulated as:

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a')|s,a\right].$$

These value iteration algorithms aim to converge towards the optimal action-value function Q_i approaching Q^* as $i \to \infty$. However, estimating the action-value function for each sequence separately is impractical due to the lack of generalization.

To tackle this, function approximators, such as neural networks, are utilized. These neural networks, or Q-networks, parameterized by weights θ, are trained by minimizing a loss function, which at each iteration i is given by:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right], \tag{2}$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$ is the target for iteration i, with $\rho(s,a)$ being the behavior distribution over sequences and actions.

The paper's approach contrasts with supervised learning, where targets evolve dynamically with the network weights. This is reflected in the gradient of the loss function for network weight updates:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]. \tag{3}$$

The Stochastic Gradient Descent (SGD) is employed to optimize the loss function. In this approach, weights are updated after every timestep, replacing the expectations with single samples from ρ and $\mathcal{E}$, leading to the Q-learning algorithm. This model-free algorithm learns directly from samples of $\mathcal{E}$ without constructing a model of the environment and is off-policy, learning about the optimal policy while following a behavior distribution that facilitates exploration, typically through an ϵ-greedy strategy.

Description of the Algorithm

The approach discussed in the paper builds upon recent advancements in training deep neural networks for computer vision and speech recognition, where learning directly from raw inputs has been notably successful. This concept is applied to RL, with the aim of connecting an RL algorithm to a deep neural network that processes RGB images and utilizes stochastic gradient updates for efficient training.

Drawing inspiration from Tesauro's TD-Gammon, which updated a network's parameters from on-policy experiences, the paper explores the potential of leveraging modern deep neural networks and scalable RL algorithms. However, a distinct technique called "experience replay" is employed. In this method, the agent's experiences at each timestep, denoted as $e_t = (s_t, a_t, r_t, s_{t+1})$, are stored in a replay memory $\mathcal{D}$. The algorithm then samples randomly from this pool to apply Q-learning updates.

This deep Q-learning approach offers several advantages. It enhances sample efficiency by reusing experiences for multiple weight updates. Random sampling from the replay memory alleviates the correlation issue in consecutive samples, reducing update variance. Furthermore, it smooths out the learning process and prevents oscillations or divergence in parameter values.

The pseudocode for the deep Q-learning algorithm is outlined as follows:

Algorithm 1 Deep Q-learning Algorithm

```
for each episode do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for each timestep $t$ do
        With probability $\epsilon$ select a random action $a_t$
        Otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        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 replay memory $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = r_j$ if episode terminates at step $j + 1$
        Otherwise set $y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to $\theta$
    end for
end for
```

Results

Akin to the original paper [4], we tested seven Atari 2600 games (Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, Space Invaders) using the same network architecture, learning algorithm, and hyperparameters to demonstrate the approach's versatility without game-specific adjustments. Figure 1 provides sample screenshots from these Atari games. We used the same discount factor ($\gamma = 0.99$) and identical modified reward structure for training only, normalizing all positive and negative rewards to 1 and -1, respectively, to maintain consistent learning across games. This might impact the agent's performance, as it cannot distinguish between different reward sizes.

For the optimization part, we also employed the RMSProp algorithm with 32-sized mini-batches and an ϵ-greedy policy, reducing ϵ from 1 to 0.1 over the first million frames, then fixing it at 0.1 for a total of 10 million frames, with a replay memory of the last million frames. We used frame-skipping (kth frame action selection) for efficiency, with k=4 for all games. While k=3 was reportedly used for Space Invaders in order to make the lasers visible, this was not the case in later studies [5] so we chose k=4 for consistency and ease of generalization. The number of "no-op" (i.e., "do nothing") actions was also set to be 30 at maximum.

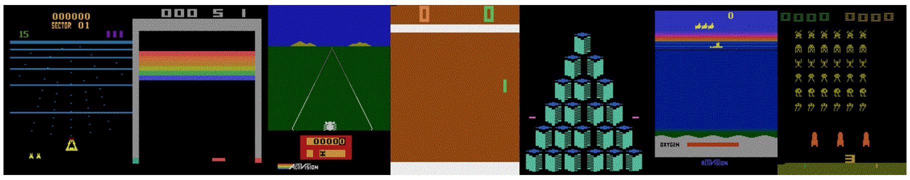
Figure 1: Screen shots from the seven Atari 2600 games we employed. *From left to right*: Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, and Space Invaders

For all seven Atari games, the DQN architecture takes an input of an $84 \times 84 \times 4$ image generated by ϕ [4]. The architecture consists of three hidden layers: the first applies 16 8×8 filters with stride 4 and a rectifier nonlinearity, the second applies 32 4×4 filters with stride 2 and a rectifier nonlinearity, and the final hidden layer is fully connected with 256 rectifier units. The output layer is fully connected with one output for each valid action, and the number of valid actions varies from 4 to 18 in the games. RMSProp was initialized with learning rate of 0.00025, gradient momentum and squared gradient momentum of 0.95, and minimum squared gradient of 0.01 [5].

While some methods use both history and action as inputs to parameterize the action-value function using a neural network, this can be computationally expensive because it requires a separate pass for each action. In contrast, the DQN approach uses a single forward pass with only the state representation as input, with separate output units for each action, allowing for efficient computation of Q-values for all possible actions in that state.

To reduce computational demands when working with raw Atari frames (210×160 pixels with a 128-color palette), the frames are first converted to grayscale, down-sampled to 110×84, and cropped to 84×84 for input. This preprocessing is performed on the last 4 frames of a history and stacked for DQN input in GPU implementation. Cropping also intends to remove the score area from the game display, where the score might confuse the network.

Cropping was additionally investigated by using DQN with direct down-sampling to 84×84, dubbed "Uncropped". Moreover, in attempts to improve learning, we investigated learning rate scheduling via the exponential scheduler, "Exp LR", as well as cosine annealing with warm restarts, "Cos LR + Warm Restarts". An exponential learning rate scheduler decays the learning rate of each parameter group by a hyperparameter γ every epoch i: $\eta_{i+1} = \gamma\eta$ ($\gamma = 0.99999$ in our case). Cosine annealing with warm restarts sets the learning rate of each parameter group using a cosine annealing schedule [2]: $\eta_t = \eta_{min} + 0.5(\eta_{max} - \eta_{min})(1 + \cos(\pi T_{current}/T_i)$, where η_{max} is set to the initial learning rate (0.01 in our case), η_{min} is a hyperparameter (0.00025 in our case), T_{curent} is the number of epochs since the last restart, and

T_i is the number of epochs between two warm restarts (25,000 in our case). Gradient clipping to within the range of $[-1, 1]$ was also considered under "Exp LR + Clip Grad", which is hypothesized to stabilize training by preventing excessively large updates to network weights

Reproduced Experimental Results

Table 1 displays the average rewards obtained by the cropped and uncropped methods under a random policy to set a baseline. The evaluations were conducted for 10,000 steps, as per the original paper [4].

Method	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Uncropped	195.6	0.1706	-1.0	-21.0	67.8	20.0	65.5
Cropped	195.6	0.1431	-1.0	-21.0	70.2	22.3	67.6

Table 1: Average Rewards under Random Policy ($\epsilon = 1$) across Games

An illustration of the rewards and Q-values during training is supplied in Figure 2. The rewards climb up as a general trend, and Q-values initially explode then stabilize and increase steadily. The plots were smoothed with a Savitzky-Golay filter with window length 5 and fitting polynomial order 3, in order to facilitate proper visualization of the plots.

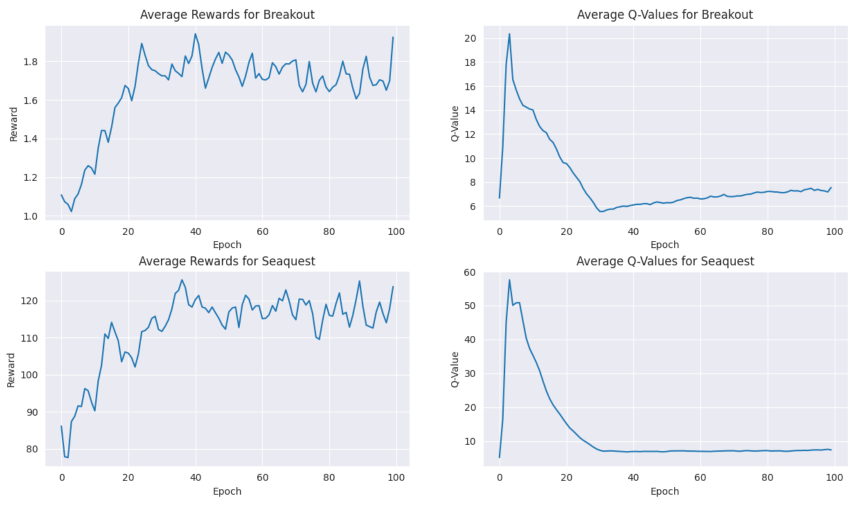

Figure 2: Average reward (*left column*) and average maximum predicted action-value (Q; *right column*) per episode on Breakout (*top row*) and Seaquest (*bottom row*) during training. The statistics were computed by running an ϵ-greedy policy with $\epsilon = 0.05$ for 10,000 steps. The two plots on the right show the. One epoch corresponds to 50,000 minibatch weight updates

The results from the evaluation phase of DQN are tabulated below in Table 2. They represent inference results over 10,000 steps, displaying the average rewards for different methods across various games. Each row in the table corresponds to a specific method, while each column represents a different game. The bottom row re-displays the results from the original paper [4] for convenience. The apparent best approach, "Exp LR + Clip Grad", supersedes the random baseline results from Table 1 in terms of all games except Enduro and Pong. The "Uncropped" approach performs poorly, except in Q*ber and Space Invaders, when compared to the corresponding random baseline.

Method	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Uncropped	38.5	0.0029	-1.0	-21.0	141.8	0.0	86.2
Exp LR	12.9	0.3429	25.0	-21.0	121.4	20.9	92.5
Exp LR + Clip Grad	255.2	0.5446	-1.0	-20.0	119.6	64.2	103.2
Cos LR + Warm Restarts	44.0	0.4825	17.0	-21.0	120.0	47.1	91.1
DQN [4]	4092	168	470	20	1952	1705	581

Table 2: Average Rewards Across Games with an ϵ-greedy Strategy ($\epsilon = 0.05$). The last row reiterates the results from the original DQN paper [4] for ease of comparison. "B. Rider" for Beam Rider, "S. Invaders" for Space Invaders, "Exp LR" for Exponential Learning Rate, "Clipped Grad" for Gradient Clipping, and "Cos LR" for Cosine Learning Rate. "Warm Restarts" refer to periodic learning rate resets. All methods, except "Uncropped", involve cropping the input RGB images.

Comparison with and Critique of the Original DQN Paper's Results

In a comparative analysis of our various attempts to re-implement DQN as detailed in the original paper [4], a stark contrast in performance is evident across all seven Atari games. The DQN method consistently outperforms all four of our method variants. For instance, in Beam Rider, the "DQN" reward of 4092 is significantly higher than our best method, "Exp LR + Clip Grad", which scored only 255.2. This trend is consistent across all games, where DQN not only surpasses but substantially elevates the performance benchmark. Notably, in Breakout and Seaquest, "DQN" scores 168 and 1705, respectively, and dwarfs our attempts' rewards.

This seems to indicate something missing in the original paper [4], which is further corroborated by the subsequent paper [5] that attempts to clarify certain occluded explanations. While the second paper [5] is of significantly higher quality, it still misses explanations and contains logical holes that are not addressed. Instead, the authors aim to include more games in their evaluations. While no doubt the peer review process improved the quality of the work, more transparency and higher clarity is needed.

Discussion & Conclusion

In our replication of the DQN experiments, we observed that while our methods showed competence in certain games, they generally fell short of the original DQN results. This disparity underscores the nuanced complexity inherent in deep RL. Specifically, minor variations in implementation or parameter tuning can lead to significant performance differences. Our work highlights the challenges in replicating deep learning models, emphasizing the need for detailed documentation and transparency in research publications.

The replication process also brought to light the importance of feature preprocessing and network architecture choices. The cropping and learning rate strategies, for example, had noticeable impacts on game performance, suggesting that these are crucial factors in the model's success. Our analysis indicates that while deep learning in RL is a powerful tool, its effective application requires careful consideration of these elements.

In conclusion, this project served as a valuable learning experience in the field of deep RL. It provided practical insights into the challenges of replicating complex models and the critical role of meticulous methodology.

References

[1] Marc G. Bellemare et al. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2013.

[2] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2017.

[3] Hamid Maei et al. Toward off-policy learning control with function approximation. *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, 2010.

[4] Volodymyr Mnih et al. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 02 2015.

[6] Jordan B. Pollack and Alan D. Blair. Why did td-gammon work. *Advances in Neural Information Processing Systems*, 1996.

[7] Martin Riedmiller. Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method. *Machine Learning: ECML 2005*, 2005.

[8] Brian Sallans and Geoffrey E. Hinton. Reinforcement learning with factored states and actions. *Journal of Machine Learning Research*, 2004.

[9] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[10] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 1997.

Appendix

In this section, we present our core implementation of the Deep Q-Network, excluding certain utility functions used for evaluation and presentation purposes. For a comprehensive view of the full implementation, please visit our GitHub Repository.[1]

Implementation of DQN from Scratch

```python
import os
import argparse
import gym
import torch
import random
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

import matplotlib
matplotlib.use('Agg')

from torch.nn import Module, Sequential, Conv2d, ReLU, Flatten, Linear, init
from torch.nn.functional import huber_loss
from torch.nn.utils import clip_grad_norm_

from stable_baselines3.common.atari_wrappers import MaxAndSkipEnv
from stable_baselines3.common.buffers import ReplayBuffer


class DQN(Module):
    """ DQN - Deep Q-Network Definition """
    def __init__(self, n_actions):
        super().__init__()
        self.network = Sequential(
            Conv2d(4, 16, 8, stride=4),
            ReLU(),

            Conv2d(16, 32, 4, stride=2),
            ReLU(),

            Flatten(),
            Linear(2592, 256),
            ReLU(),

            Linear(256, n_actions)
            )

        # Initialize the layers with Xavier Initialization
        for layer in self.network:
            if isinstance(layer, (Conv2d, Linear)):
                init.xavier_uniform_(layer.weight)

    def forward(self, x):
        return self.network(x / 255.)


def DQL(env, env_name,
        N=1_000_000, n_epochs=5_000_000,
        update_f=4, bs=32,
        gamma=0.99, replay_start_size=50_000,
        initial_explore=1., final_explore=0.1,
        explore_steps=1_000_000, device='cuda',
        seed=0, savepath='', lr_scheduler=False, clip_grads=False) -> None:
    D = ReplayBuffer(N, env.observation_space, env.action_space, device,
                     optimize_memory_usage=True, handle_timeout_termination=False)
```

[1] https://github.com/mhakankara/EEE548_Term_Project

```
    q_net = DQN(env.action_space.n).to(device)


    if lr_scheduler:
        optim = torch.optim.RMSprop(
            q_net.parameters(),
            lr=0.01, alpha=0.95, eps=0.01, momentum=0.95
        )

        scheduler = torch.optim.lr_scheduler.ExponentialLR(optim, gamma=0.99999)
        # scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optim, T_0=25_000,
            eta_min=0.00025)
    else:
        optim = torch.optim.RMSprop(
            q_net.parameters(),
            lr=0.00025, alpha=0.95, eps=0.01, momentum=0.95
        )

    # Gradient clipping
    if clip_grads:
        max_grad_norm = 1.0

    epoch = 0

    smoothed_rewards = []
    rewards_list = []

    smoothed_q_values = []
    average_q_values = []

    progress_bar = tqdm(total=n_epochs)

    scheduler_str = '_lr_scheduler' if lr_scheduler else ''

    while (epoch <= n_epochs):

        died = False

        total_rewards = 0
        max_q_values = []

        # Initialise sequence s_1 = \{x_1\} and preprocessed sequenced \phi_1 = \phi(s_1)
        seq = env.reset()

        # Fire after reset / "do nothing"
        for _ in range(random.randint(1, 30)):
            seq, _, _, info = env.step(1)

        # Play until death
        while not died:
            current_life = info['lives']

            # Probability \epsilon, decays with each epoch
            epsilon = max(
                (final_explore - initial_explore) / explore_steps * epoch + initial_explore,
                final_explore
                )

            # With probability \epsilon select a random action a.
            if (random.random() < epsilon):
                action = np.array(env.action_space.sample())
            # Otherwise select a = max_a Q^{*}(\phi(s_t), a; \theta)
            else:
                q_values = q_net(torch.Tensor(seq).unsqueeze(0).to(device))
                action = torch.argmax(q_values, dim=1).item()

            # Execute action a in emulator and observe reward r_t and image x_{t+1}
            next_seq, r, died, info = env.step(action)

            died = (info['lives'] < current_life) or died

            real_next_seq = next_seq.copy()
            total_rewards += r

            # Reward clipping
            r = np.sign(r)

            # Store transition (\phi_t, a_t, r_t, \phi_{t+1}) in D
            D.add(seq, real_next_seq, action, r, died, info)

            seq = next_seq

            # Minibatch updates
            if (epoch >= replay_start_size) and (epoch % update_f == 0):
                minibatch = D.sample(bs)
                with torch.no_grad():
                    max_q_value, _ = q_net(minibatch.next_observations).max(dim=1)
                    y = minibatch.rewards.flatten() + gamma * max_q_value * (1 - minibatch.
                        dones.flatten())
                current_q_value = q_net(minibatch.observations).gather(1, minibatch.actions).
                    squeeze()
                loss = huber_loss(y, current_q_value)

                max_q_values.append(current_q_value.max().item())

                optim.zero_grad()
                loss.backward()
                if clip_grads:
                    # Clip gradients to the specified range [-max_grad_norm, max_grad_norm]
                    clip_grad_norm_(q_net.parameters(), max_grad_norm)
                optim.step()

                # Update scheduler
                if lr_scheduler and scheduler.get_last_lr()[-1] > 0.00025:
                    scheduler.step()

            epoch += 1
            plot_period = n_epochs // 100
            if (epoch % plot_period == 0) and (epoch > 0):
                smoothed_rewards.append(np.mean(rewards_list))
                rewards_list = []

                plt.figure()
                plt.plot(smoothed_rewards)
                plt.title(f'Average Reward on {env_name.title()}')
                plt.xlabel('Training Epochs')
                plt.ylabel('Average Reward per Episode')
                plt.savefig(fr'{savepath}/{env_name.title()}_reward_{seed}{scheduler_str}.png'
                    )
                plt.close()

                smoothed_q_values.append(np.mean(average_q_values))
                average_q_values = []

                plt.figure()
                plt.plot(smoothed_q_values)
                plt.title(f'Average Q on {env_name.title()}')
                plt.xlabel('Training Epochs')
                plt.ylabel('Average Action Value (Q)')
                plt.savefig(fr'{savepath}/{env_name.title()}_Q_{seed}{scheduler_str}.png')
                plt.close()

                with open(fr'{savepath}/{env_name.title()}_reward_{seed}{scheduler_str}.npy',
                    'wb') as f:
                    np.save(f, np.array(smoothed_rewards))

                with open(fr'{savepath}/{env_name.title()}_Q_{seed}{scheduler_str}.npy', 'wb')
                     as f:
                    np.save(f, np.array(smoothed_q_values))

                torch.save(q_net.state_dict(), fr'{savepath}/{env_name.title()}_{seed {
                    scheduler_str}.pth')

            progress_bar.update()
```

```
        # Append results if not NaN/None/[]
        if not total_rewards:
            rewards_list.append(0.)
        else:
            rewards_list.append(total_rewards)

        if not max_q_values:
            average_q_values.append(0.)
        else:
            average_q_values.append(np.mean(max_q_values))

class AtariCropWrapper(gym.Wrapper):
    def __init__(self, env, game_name):
        super(AtariCropWrapper, self).__init__(env)
        self.game_name = game_name
        self.crop_values = self.get_crop_values(game_name)

        # Determine the dimensions after cropping
        self.crop_height = self.crop_values[2] - self.crop_values[0]
        self.crop_width = self.crop_values[3] - self.crop_values[1]

        # Set the observation space to the cropped dimensions
        self.observation_space = gym.spaces.Box(low=0, high=255, shape=(self.crop_height, self
            .crop_width, 3), dtype=np.uint8)

    def get_crop_values(self, game_name):
        # Define crop values for different Atari games
        crop_values = {
            'BeamRider': (30, 5, 195, 160),
            'Breakout': (30, 5, 195, 160),
            'Enduro': (30, 10, 195, 150),
            'Pong': (35, 5, 190, 160),
            'Qbert': (30, 5, 190, 160),
            'Seaquest': (30, 10, 190, 160),
            'SpaceInvaders': (30, 5, 190, 160),
        }
        return crop_values.get(game_name, (0, 0, 210, 160))  # Default to (0, 0, 210, 160) if
            game_name not found

    def preprocess_observation(self, observation):
        # Crop observation without converting to grayscale
        cropped = observation[self.crop_values[0]:self.crop_values[2], self.crop_values[1]:
            self.crop_values[3], :]
        return cropped

    def step(self, action):
        observation, reward, done, info = self.env.step(action)
        return self.preprocess_observation(observation), reward, done, info

    def reset(self, seed=None, options=None):
        # We need the following line to seed self.np_random
        super().reset(seed=seed)

        observation = self.env.reset()
        return self.preprocess_observation(observation)


def create_env(env_name, k=4, seed=0, crop=False) -> gym.Env:
    # env = gym.make(f'ALE/{env_name}-v5')
    env = gym.make(f'{env_name}NoFrameskip-v4') # Better behavior
    # env = gym.make(f'{env_name}NoFrameskip-v4', render_mode='human') # Better behavior

    # Replicate paper's conditions (grayscale, frameskip, etc.)
    if crop:
        env = AtariCropWrapper(env, env_name)
    env = gym.wrappers.RecordEpisodeStatistics(env)
    env = gym.wrappers.ResizeObservation(env, (84, 84))
    env = gym.wrappers.GrayScaleObservation(env)
    env = gym.wrappers.FrameStack(env, k)
    env = MaxAndSkipEnv(env, skip=k)

    env.seed(seed)
    np.random.seed(seed)
    random.seed(seed)
    torch.manual_seed(seed)

    return env

def clean_gpu_cache():
    if torch.cuda.is_available():
        torch.cuda.empty_cache()

if __name__ == '__main__':
    # Environments considered in the paper
    env_list = ['BeamRider', 'Breakout', 'Enduro', 'Pong', 'Qbert', 'Seaquest', 'SpaceInvaders
        ']

    parser = argparse.ArgumentParser()
    parser.add_argument('--seed', default=0, type=int)
    parser.add_argument('--env', default=env_list[0], choices=env_list)
    parser.add_argument('--savepath', default='Env_Plots')
    parser.add_argument('--device', default='cuda')
    parser.add_argument('--epochs', default=1_000_000, type=int)
    parser.add_argument('--N', default=750_00, type=int)
    parser.add_argument('--lr_scheduler', action='store_true')
    parser.add_argument('--clipgrads', action='store_true')
    parser.add_argument('--cropenv', action='store_true')
    args = parser.parse_args()

    savepath = os.path.join(os.getcwd(), args.savepath)

    if not os.path.isdir(savepath):
        os.makedirs(savepath)

    # Check if CUDA is available
    if args.device == 'cuda' and not torch.cuda.is_available():
        print('CUDA not available. Switching to CPU ...')
        device = 'cpu'
    else:
        device = torch.device(args.device)
        clean_gpu_cache()

    # k = 3 if args.env == 'SpaceInvaders' else 4 # Lasers disappear, apparently
    k = 4
    env = create_env(args.env, k=k, seed=args.seed, crop=args.cropenv)

    print('Environment:', args.env)
    print('Device:', device)

    # Maximum N allowed due to mem limits === 750_000 (<1_000_000)
    DQL(env, args.env, n_epochs=args.epochs, N=args.N, update_f=k, device=device, seed=args.
        seed, savepath=savepath, lr_scheduler=args.lr_scheduler, clip_grads=args.clipgrads)

    env.close()
```