Neural net project

Reviving DeepMind's DQN: A Replication Study

Briceno Benjamin, Cerutti Paolo, Delisle Thibault, Genin Victor,
Department of Computer Science,
Østfold University College

1. INTRODUCTION

The group consisting of Briceno Benjamin, Cerutti Paolo, Delisle Thibault, and Genin Victor, are Computer Science students at Østfold University College.

People often use pre-made algorithms from libraries without deeply understanding their inner workings. In this paper, we aim to shed light on this issue and provide an accessible explanation that facilitates a deeper comprehension and enables readers to recreate the algorithm with ease. In the pursuit of recreating the DQN Algorithm for Deep Reinforcement Learning using a minimalist approach, this research endeavors to uncover the underlying challenges, trade-offs, and intricacies involved in the reimplementation.

To faithfully recreate the DQN algorithm from DeepMind using minimal external libraries, and to comprehensively explore the nuances of its reimplementation. This research aims to gain insights into the DQN algorithm's core principles, evaluate its performance in a simplified game environment, and potentially enhance it with recent improvements.

This report will focus on our objective of reproducing and making functional a DQN (Deep Q-Network) algorithm used in reinforcement learning. To achieve this, a great deal of research and reading had to be done to learn these technologies and methods and then implement them in an environment. For simplicity's sake, the neural network will be implemented in a simple game environment where the number of inputs is limited. This will make it easier to evaluate performance of our DQN algorithm.

1.1. BACKGROUND

1.1.1. Reinforcement Learning

Reinforcement learning is a branch of machine learning based on the concept of an agent's interaction with an environment. The main aim of reinforcement learning is to teach an agent how to make sequential decisions to maximize a cumulative reward over time. The process works as follows: the agent interacts with the environment by performing actions, and in return, the environment provides a reward or punishment depending on the actions taken. The agent then learns to optimize its actions to maximize rewards over the long term by exploring different strategies. This exploration is balanced with the use of the knowledge acquired to make informed decisions. Reinforcement learning algorithms and deep neural networks are commonly used to solve complex problems such as game control, robotics and many others, where the long-term consequences of actions are crucial.

Here's a brief overview of the key components in reinforcement learning:

- **Agent:** The entity that makes decisions and takes actions in the environment.
- **Environment:** The external system or surroundings with which the agent interacts.
- **State:** A representation of the current situation or configuration of the environment.
- **Action:** The set of possible moves or decisions that the agent can take in a given state.
- **Reward:** Feedback from the environment that quantifies the immediate benefit or cost of an action taken by the agent.
- **Policy:** The strategy or mapping from states to actions that the agent learns to maximize its cumulative reward.

In summary, reinforcement learning involves the agent exploring different actions, receiving feedback, and adjusting its strategy over time through a learning process. The agent aims to discover the optimal policy that leads to the most favorable outcomes in terms of cumulative reward.

1.1.2. DQN

The Deep Q-Network (DQN) is a reinforcement learning algorithm that exploits the advantages of deep learning in combination with Q-learning, a classic reinforcement learning approach. DQN works by using a deep neural network to estimate the Q-function, which associates with each state-action pair a value representing the expectation of future reward. These Q values help the agent to make decisions by selecting the most promising actions in a given state. DQN learning is iterative, where the agent collects data by interacting with the environment, storing these experiences in a buffer. Periodically, a sample of experiments is taken to train the neural network to minimize the mean square error between Q estimates and actual observed rewards. This network update improves the accuracy of Q estimates, helping the agent to make more intelligent decisions.
In the next paragraphs, we'll outline features from DeepMind's DQN algorithm that we recreated in our project.

Experience Replay: In the context of reinforcement learning, Experience Replay involves the systematic storage of agent-environment transitions, comprising state-action pairs, associated rewards, and the subsequent states, in a replay memory. During training, batches of these experiences are randomly sampled from the replay memory rather than using sequential experiences. This approach serves to break the temporal correlation in the data, preventing the learning algorithm from being overly influenced by recent experiences. By introducing this element of randomness, Experience Replay contributes to improved stability in training, facilitating a more robust and efficient learning process.

Target Q-Network: To enhance the stability of training in DQN and similar reinforcement learning algorithms, a Target Q-Network is introduced. This network serves as a separate set of parameters, periodically updated to mimic the current Q-network's weights. This delayed and less frequent update of the target network helps to mitigate the potential issues of using rapidly changing targets during the learning process. By providing a more stable set of target Q-values, the target Q-network contributes to a smoother convergence of the Q-learning algorithm, ultimately improving the effectiveness of the learned policy.

Exploration-Exploitation: During the training phase of reinforcement learning, Exploration-Exploitation is addressed using an ε-greedy strategy for action selection. This strategy involves a balance between exploration (randomly selecting actions) and exploitation (choosing the action with the highest Q-value). With a probability of ε, the agent explores by taking a random action, while with a probability of 1-ε, it exploits its current knowledge by selecting the action with the highest estimated Q-value. This trade-off ensures that the agent both explores the environment to discover potentially better actions and exploits its existing knowledge to maximize short-term rewards, contributing to a well-balanced and effective learning process.

Reward Clipping: In the training of reinforcement learning models, Reward Clipping involves constraining the received rewards to a specific range, often [-1, 1]. This clipping process is applied to limit the impact of extreme rewards that could lead to instability during training. By confining the rewards within a manageable range, the learning algorithm becomes less sensitive to large variations, preventing potential issues such as exploding gradients and promoting a more controlled and stable learning process. Reward Clipping is a practical technique to ensure that the learning algorithm converges effectively without being disproportionately influenced by extreme reward values.

2. ANALYSIS

The analysis section serves as the core of our research. It encompasses our approach, the tools we use, and a review of related works. In this section, we lay the foundation for our study, detailing how we plan to replicate and analyze DeepMind's DQN model. By exploring relevant works, outlining our approach, and introducing our tools, we establish the framework for our research methodology and design.

2.1. RELATED WORKS

Our project is based on DeepMind Technologies' report "Playing Atari with Deep Reinforcement Learning"[1], in which the editors implement the DQN method on Atari game environments to beat the records of other algorithms, and even those of a human. The results show that this method does indeed outperform other algorithms and even manages to beat a human on a few games, without needing to change hyperparameters from one game to another, showing that this method is robust in all cases and can be further improved on a case-by-case basis.

Our work also relied on learning the DQN model from the explanations and documentation of youtuber "sentdex", who also shared his work on the "pythonprogramming.net" website[4]. In his reports, he explains in 6 chapters how the DQN model works and is coded, starting with the simplest basics through to implementation and full training of the algorithm.

We based our work on the paper "Human-level control through deep reinforcement learning" too [2], which is the natural successor of the previous paper [1]. This report explains its approach to the DQN model by training it on 49 atari games. The authors then compare the results obtained with other algorithms and with a professional gamer, and seek to improve the algorithm by modifying parameters one by one in order to maximize results. This document will therefore be useful to us in the future, when we also make modifications to our model to improve our results and try to obtain ones as good as the documents we rely on.

We can see that a lot of work has already been done on the DQN model implemented in Atari games. So, with the help of these reports, we're going to create and implement our own algorithm to understand how it works throughout the code and how to optimize its parameters according to the environments used.

2.2. METHOD AND DESIGN

Learning Foundation:
Starting from scratch, we delved into our exploration of reinforcement learning by studying Mutual Information's Reinforcement Learning video series [3]. Building upon this, we practiced implementing basic Reinforcement Learning algorithms with the assistance of a reliable online resource [4, 5]. The insights and skills gained from this exercise were then applied to comprehend the contents of papers [1,2] and the algorithms presented within them.

Practical Implementation:
Translating theoretical knowledge into practical skills, we quickly applied our learning through hands-on experience. Using reputable online resources [4, 5], we implemented basic Reinforcement Learning algorithms. This practical phase served as a crucial bridge, facilitating a smooth transition from abstract concepts to actual code.

Replicating DeepMind's DQN Algorithm:
Algorithmic Framework:
Our strategy for replicating the DQN algorithm involved a phased plan. Starting with a library-dependent approach, we utilized existing frameworks to create a functional reinforcement learning program. This initial phase helped us understand the broader landscape and functionality of the DQN algorithm.

Iterative Refinement:
To deepen our understanding and gain more control over the algorithm, we took on the task of reconstructing the algorithm without external libraries. This phase included a detailed exploration of DeepMind's original papers [1, 2] to grasp the intricacies of the DQN model. The next step was to take on the challenge of re-implementing the entirety of the project without utilizing any libraries.

2.3. TOOLS

In this section, we introduce the key tools that will play a central role in our research, Gymnasium and PyTorch. These tools empower us to recreate and analyze DeepMind's DQN model efficiently, facilitating our study of reinforcement learning in controlled environments.

2.3.1. Gymnasium

"Gymnasium" is an open-source reinforcement learning environment developed by DeepMind. It provides a flexible platform for researchers and developers to experiment with reinforcement learning algorithms, making it an invaluable tool for our study. Gymnasium offers a wide range of pre-built environments, making it easier to test and evaluate reinforcement learning algorithms in diverse scenarios. We can then concentrate on the fundamental principles of reinforcement learning without the complications of real-world scenarios.

2.3.2. Pytorch

PyTorch is an open-source machine learning framework that provides valuable support for our research. It simplifies the development and implementation of neural network models, making it a powerful tool in our study of replicating DeepMind's DQN model.
One noteworthy advantage of PyTorch is its flexibility in model design and training. It allows us to easily construct and fine-tune neural networks, including the DQN architecture, with a high level of control. This flexibility enables us to experiment with different components and hyperparameters to better understand how they impact the model's performance.

3. IMPLEMENTATION

The code relies on two classes, namely DQN and DQNAgent. These classes are subsequently employed in the main training loop to efficiently construct and train the neural network. In the upcoming sections, following a comprehensive introduction to these classes, we will dissect the most important parts of the code step by step, facilitating a clear understanding for easy re-implementation. Note that the full code is presented at the end of the paper.

DQN Class: Encapsulates a simple neural network. This network is composed of fully connected layers and serves both as the policy and target networks.
DQNAgent: Is responsible for managing the agent's behavior, memory, exploration strategy, and training process. Experience replay is employed to store and sample transitions, while ε-greedy action selection enhances exploration-exploitation balance. The optimization of the neural network is carried out using Huber loss.
Main: Sets up the CartPole environment and initializes the DQNAgent. It then executes the main training loop for a specified number of episodes and utilizes the plotting function to present the training progress or final results.
Training loop: is a critical component, iterating through episodes and employing an epsilon-greedy strategy for action selection. It stores transitions in the agent's memory and performs model optimization. Additionally, target network updates are implemented using a soft update strategy, contributing to more stable training. The training loop tracks episode durations and scores, providing valuable insights into the agent's performance.

3.1. DQN Class

```
class DQN(nn.Module):
    def __init__(self, state_size, action_size):
        super(DQN, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(state_size, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, action_size)
        )
```

Inside the constructor, a neural network is defined using the `nn.Sequential` container. This container allows us to sequentially apply a list of layers to the input data. In this case, the neural network consists of three linear layers (fully connected layers) with ReLU activation functions in between them.

- The first `nn.Linear(state_size, 128)` layer takes an input of size state_size and produces an output of size 128.
- The first `nn.ReLU()` layer applies the rectified linear unit (ReLU) activation function element-wise.

- The second `nn.Linear(128, 128)` layer takes the output from the previous layer (of size 128) and produces another output of size 128.
- The second `nn.ReLU()` layer applies the ReLU activation function again.
- The third `nn.Linear(128, action_size)` layer takes the output from the previous layer and produces the final output of size `action_size`. This represents the Q-values for each possible action in the given environment.

So, in summary, this code defines a simple feedforward neural network with two hidden layers and ReLU activation functions for a DQN. The input size is `state_size`, and the output size is `action_size`. The neural network is designed to map states to Q-values for each possible action in a reinforcement learning setting.

2.3. DQNAgent Class

The agent is initialized with parameters specifying the size of the state space and the number of possible actions. A replay memory is created using a deque with a maximum size determined by `MEMORY_SIZE`. Essential hyperparameters, including the discount factor (`gamma`) and epsilon-greedy exploration parameters (`eps_start`, `eps_end`, `eps_decay`), are set. Two instances of the DQN class, `self.policy_net` and `self.target_net`, are created to serve as the policy network and target network, respectively. The initial weights of the target network are loaded to match those of the policy network, and the target network is set to evaluation mode. Furthermore, an AdamW optimizer is initialized for the policy network, completing the setup for the agent.

Code:

```
def __init__(self, state_size, action_size):
    self.state_size = state_size
    self.action_size = action_size
    self.memory = deque(maxlen=MEMORY_SIZE)
    self.gamma = GAMMA
    self.eps_start = EPS_START
    self.eps_end = EPS_END
    self.eps_decay = EPS_DECAY
    self.policy_net = DQN(state_size, action_size)
    self.target_net = DQN(state_size, action_size)
    self.target_net.load_state_dict(self.policy_net.state_dict())
    self.target_net.eval()
    self.optimizer = optim.AdamW(self.policy_net.parameters(), lr=LR, amsgrad=True)
    self.steps_done = 0
```

The following code incorporates epsilon-greedy action selection, where, based on a specified probability threshold (`eps_threshold`), the agent either selects the action with the highest Q-value according to the policy network or, alternatively, chooses a random action when the probability threshold is not met. This strategy balances exploration and exploitation, allowing the agent to explore the action space while also exploiting the current best-known actions.

```
def select_action(self, state):
    sample = random.random()
    eps_threshold = self.eps_end + (self.eps_start - self.eps_end) * np.exp(-1.0 * self.steps_done / self.eps_decay)
    self.steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            return self.policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(self.action_size)]], dtype=torch.long
```

Code:
Next we have the optimization step, which aims to train the neural network, referred to as the policy network (`self.policy_net`), to enhance its Q-value predictions. Here's a breakdown of the code:

Firstly, the code checks if there are a sufficient number of samples in the replay memory to create a batch for training. If the replay memory size is less than the specified batch size (`BATCH_SIZE`), the optimization process is skipped.

Next, a batch of transitions is randomly sampled from the replay memory using the random.sample function. These transitions consist of state-action pairs, rewards, and next states, and they are organized using the `Transition` class.

For these non-final states in the batch, additional processing is necessary; The code creates a mask to identify these non-final states. The mask is a boolean tensor that indicates whether a state in the batch is non-final or not.
The purpose of this mask is to filter out the non-final states from the batch when performing computations. This is important because the Q-value calculation for non-final states involves information from the subsequent states, and including terminal states in these calculations could lead to incorrect results. By creating and using the mask, the code ensures that the Q-value computations only consider the relevant non-final states in the batch.

The batch components, including states, actions, and rewards, are concatenated to form the input for the neural network.

Q-values for the state-action pairs in the batch are computed using the policy network (`self.policy_net`). Simultaneously, next state values are initialized for non-final states. For non-final states, the maximum Q-value from the target network (`self.target_net`) is computed and used as the next state value.

The expected Q-values for the state-action pairs are then calculated based on the Bellman equation. The smooth L1 loss is computed between the predicted Q-values and the expected Q-values, providing a measure of the disparity between the current predictions and the desired values. The smooth L1 loss is a variant of the mean absolute error (L1 loss) and is often used in Q-learning to provide a robust and less sensitive measure of the disparity between predictions and target values.

Finally, the code performs backpropagation to update the model's parameters. Gradients are zeroed, and the optimizer is utilized to adjust the parameters of the policy network. Additionally, gradient clipping is applied to prevent exploding gradients, enhancing the stability of the training process.

Code:

```
def optimize_model(self):
        if len(self.memory) < BATCH_SIZE:
            return
        transitions = random.sample(self.memory, BATCH_SIZE)
        batch = Transition(*zip(*transitions))

        non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.next_state)), dtype=torch.bool)
        non_final_next_states = torch.cat([s for s in batch.next_state
                                                    if s is not None])
        state_batch = torch.cat(batch.state)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)

        state_action_values = self.policy_net(state_batch).gather(1, action_batch)
        next_state_values = torch.zeros(BATCH_SIZE)
        next_state_values[non_final_mask] = self.target_net(non_final_next_states).max(1)[0].detach()
        expected_state_action_values =(next_state_values * self.gamma) + reward_batch

        loss = nn.functional.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze(1))

        self.optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_value_(self.policy_net.parameters(), 5)
        self.optimizer.step()
```

2.3. Main and Training Loop

The `Main` script creates a CartPole-v1 environment using OpenAI Gym. The dimensions of the state space and the number of possible actions are extracted from the environment, and a DQNAgent is initialized with these specifications.

The core of the script is the training loop, which iterates through a specified number of episodes (`EPISODES`). For each episode, the environment is reset, and the initial state is converted into a torch tensor. The agent selects actions based on its policy network, interacts with the environment, and stores the observed transitions in its replay memory. The agent's neural network is then optimized using the stored experiences. The target network is updated using a soft update strategy, and if the episode concludes, relevant statistics are recorded for later analysis.

Within the training loop, there is a conditional block that updates the target network at regular intervals defined by `TARGET_UPDATE`. The target network is updated by blending the weights of the policy network and the current target network using a parameter `TAU`. As already discussed this step is crucial for stabilizing and improving the learning process in reinforcement learning scenarios.

Code:

```python
if __name__ == "__main__":
    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    agent = DQNAgent(state_size, action_size)
    scores = []
    episodes = []
    counter = 0

    for e in range(EPISODES):
        state = env.reset()
        state = torch.tensor(state, dtype=torch.float32).view(1, -1)
        done = False
        score = 0
        while not done:
            counter += 1
            action = agent.select_action(state)
            next_state, reward, done, _ = env.step(action.item())
            score += reward
            reward = torch.tensor([reward], dtype=torch.float32)
            next_state = torch.tensor(next_state, dtype=torch.float32).view(1, -1)

            if done:
                next_state = None

            agent.store_transition(state, action, next_state, reward, done)
            state = next_state
            agent.optimize_model()

            target_net_state_dict = agent.target_net.state_dict()
            policy_net_state_dict = agent.policy_net.state_dict()

            for key in policy_net_state_dict:
                target_net_state_dict[key] = policy_net_state_dict[key] * TAU + target_net_state_dict[key] * (1 - TAU)
            agent.target_net.load_state_dict(target_net_state_dict)

            if done:
                episode_durations.append(counter + 1)
                scores.append(score)
                episodes.append(e)
                plot_durations()

        counter = 0
        if e % TARGET_UPDATE == 0:
            agent.target_net.load_state_dict(agent.policy_net.state_dict())
```

4. CHALLENGES

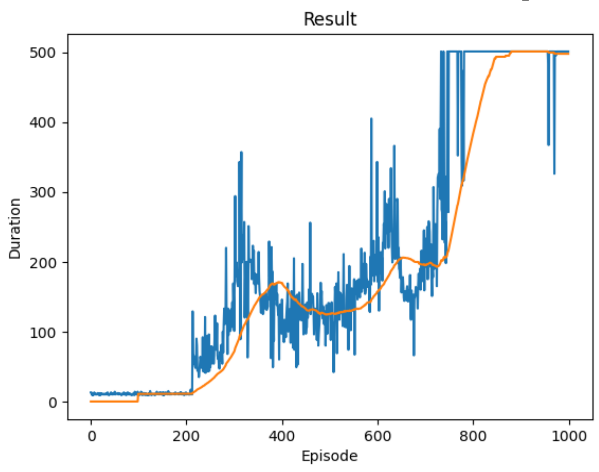
Effectively rebuilding the DQN model from the ground up requires overcoming various challenges, each demanding particular expertise. The first obstacle involves laying a robust groundwork of comprehension, covering aspects such as reinforcement learning, the logic embedded in the code, the utilized libraries, and the fundamental principles of the DQN algorithm. This foundational understanding is a necessary prerequisite for the later phases of the project.

Once the background understanding is in place, the next challenge arises in comprehending the intricacies of the DQN algorithm itself. This involves delving into the nuances of the algorithm's logic, reinforcing the theoretical understanding gained in the background phase. Understanding the algorithm lays the groundwork for the subsequent challenges.

With a comprehensive understanding of the algorithm, the focus shifts to the challenge of finding optimal hyperparameters. Key parameters, including the learning rate, discount factor, exploration rate, and target network update frequency, significantly impact the performance and convergence of the algorithm. Addressing this challenge necessitates combining insights from various sources, such as official documentation, user experiences in blog posts, and instructional video tutorials.
The final challenge involves thorough testing and debugging. As the selected hyperparameters influence the algorithm's behavior, extensive testing is imperative. This iterative process includes experimenting with different parameter values, analyzing results, identifying errors, and refining the code until achieving satisfactory outcomes. Successfully overcoming these challenges demands a comprehensive understanding of deep reinforcement learning theory, practical programming skills, and the ability to navigate the nuanced landscape of recreating a DQN algorithm from scratch.

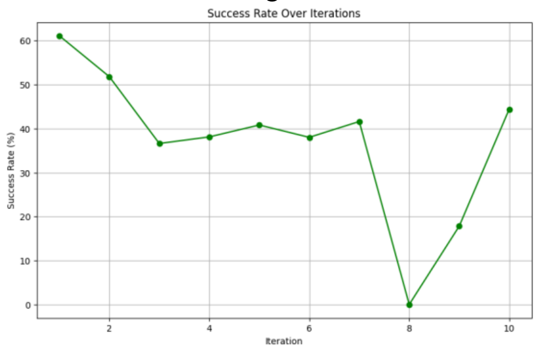
5. RESULTS AND FINDINGS

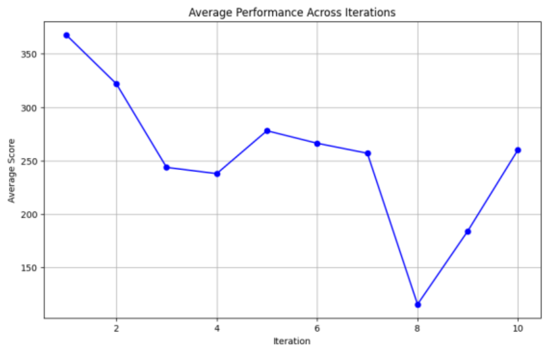
First initial Run:
After running our algorithm and adjusting the parameters, the results obtained are satisfactory, albeit relatively slow: thanks to the DQN method, the algorithm is able to play the game perfectly after around 800 episodes (see graph below). We can see, however, that the learning curve is not monotonic: it has two peaks of improvement before reaching the maximum score. This behavior is due to the rate of exploration and exploitation. In fact, the algorithm starts by exploring more before giving priority to exploitation, hence the non-constant evolution and the occasional dip in efficiency around episode 1000.



Test over 10 iterations:
In our thorough exploration across 10 rounds, we closely looked at the DQN algorithm to see how well it performs, if it's consistent, and how good it is overall. While a basic test above helps us know if the algorithm is functional, it doesn't tell us how successful it is in general. So, we did this 10 times, running 1000 episodes each time, to get a deeper understanding.
The graphs for each round encompass all 1000 episodes, indicating the influence of initial training on the representations. This factor should be kept in mind when interpreting the graphs.

Success rate and average score over each iteration:

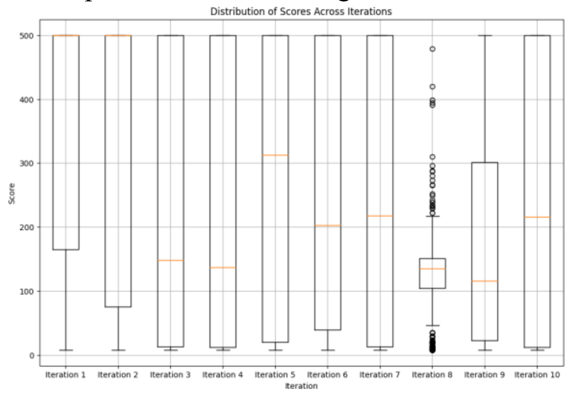


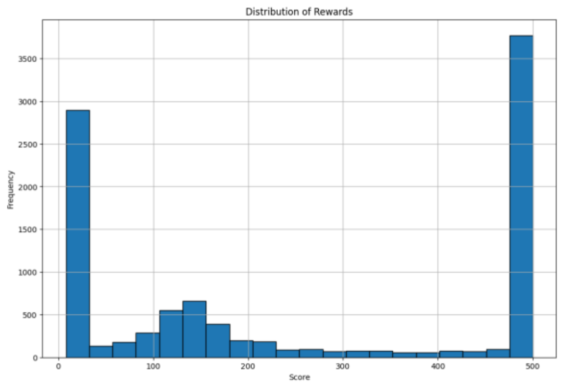


We made graphs for each round to see how often the algorithm hit 500 steps (success rate) and what the average score was. The graphs showed a clear trend: the algorithm either consistently reached 500 steps or fell just short after initial training.

An interesting thing we noticed was a recurring 40% success rate. This includes initial training episodes, so differences between rounds likely come from early successes rather than a big change in how well the algorithm works. But, rounds 8 and 9 were different. They struggled, especially round 8, which never managed to hit 500 steps.

Box plot over each generation and general distribution of scores over all iterations:





Looking at the box plot, we found that the algorithm's performance is all over the place, especially after training. While it generally gets good after training, the box plot shows big ups and downs. This matches what we saw before, highlighting that round 8 had a tough time getting a good score. This tells us that our algorithm isn't perfect, at least not within the 1000 episodes we used for training.

The general distribution supported this, indicating that initial training often kept performance within 100 and 200 steps, with subsequent breakthroughs coming quickly and reaching 500 steps. However, maintaining a consistent 500 steps proved challenging after the initial success.

In summary, our in-depth exploration of the DQN algorithm unveiled its nuanced performance, identifying areas of strength, challenges, and the impact of multiple training rounds on its overall effectiveness. Thoughts on the non-monotonic learning curve and recurring success rate nuances could potentially guide future refinements to enhance the algorithm's robustness.

6. CONCLUSION

This report traces the progress of our project to understand and reproduce the DQN algorithm. Our primary goal was to move beyond the use of pre-made algorithms without a deep comprehension of their internal mechanisms. After extensive research, documentation and testing, we were able to reproduce and use the algorithm based on the Deep Mind document. First, we had to familiarize ourselves with reinforcement learning and the DQN method itself, in order to understand their principle, how they work, and how to parameterize them. We then had to find out how to use them by researching more specialized documents and determining the libraries that would be useful in designing our algorithm. Gymnasium and Pytorch were chosen to create our environment and neural network respectively. After implementation and parameterization, such as experience replay, Target Q-Network, Exploration-Exploitation, and Reward Clipping, interspersed with difficulties and challenges, we obtained good results, demonstrating the success of our project

Results from our initial run showcased satisfactory performance, albeit with a non-monotonic learning curve attributed to the delicate balance of exploration and exploitation. In a more extensive examination over 10 iterations, our algorithm exhibited a recurring 40% success rate over the 1000 episodes, highlighting consistent speed of training and success. Notably, rounds 8 and 9 faced challenges, shedding light on areas for potential improvement.

The box plot analysis post-training revealed performance fluctuations, emphasizing the algorithm's imperfections within the 1000 episodes used for training. It proved challenging to sustain a consistent 500 steps with every run after training.

In summary, our effort increased our grasp of reinforcement learning concepts and their practical application while also reproducing DeepMind's DQN algorithm. In the future, the knowledge obtained from this replication research may be applied to make more adjustments and enhancements, which will support the continuous advancement of deep reinforcement learning algorithms.

7. REFERENCES

1. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). *Playing Atari with Deep Reinforcement Learning.* CoRR, abs/1312.5602. http://arxiv.org/abs/1312.5602

2. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). *Human-level control through deep reinforcement learning. Nature*, 518, 529–533. https://api.semanticscholar.org/CorpusID:205242740

3. Mutual Information. Reinforcement Learning by the Book. 2022-2023, www.youtube.com/playlist?list=PLzvYlJMoZ02Dxtwe-MmH4nOB5jYlMGBjr

4. sentdex. (2019). Reinforcement Learning. https://www.youtube.com/playlist?list=PLQVvvaa0QuDezJFIOU5wDdfy4e9vdnx-7

5. Reinforcement Learning (DQN) Tutorial. (n.d.). Pytorch.org. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

8. CODE

```
import gym
import random
import numpy as np
from collections import deque, namedtuple
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

EPISODES = 1000
MEMORY_SIZE = 1000
BATCH_SIZE = 128
GAMMA = 0.95
EPS_START = 0.9
EPS_END = 0.1
EPS_DECAY = 10
TARGET_UPDATE = 1000
TAU = 0.005
LR = 1e-4

# set up matplotlib
is_ipython = 'inline' in plt.get_backend()
if is_ipython:
    from IPython import display

plt.ion()

Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward', 'done'))

class DQN(nn.Module):
  def __init__(self, state_size, action_size):
      super(DQN, self).__init__()
```

```
        self.fc = nn.Sequential(
            nn.Linear(state_size, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, action_size)
        )

    def forward(self, x):
        return self.fc(x)

class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=MEMORY_SIZE)
        self.gamma = GAMMA
        self.eps_start = EPS_START
        self.eps_end = EPS_END
        self.eps_decay = EPS_DECAY
        self.policy_net = DQN(state_size, action_size)
        self.target_net = DQN(state_size, action_size)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()
        self.optimizer = optim.AdamW(self.policy_net.parameters(), lr=LR, amsgrad=True)
        self.steps_done = 0

    def select_action(self, state):
        sample = random.random()
        eps_threshold = self.eps_end + (self.eps_start - self.eps_end) * \
            np.exp(-1.0 * self.steps_done / self.eps_decay)
        self.steps_done += 1
        if sample > eps_threshold:
            with torch.no_grad():
                return self.policy_net(state).max(1)[1].view(1, 1)
        else:
            return torch.tensor([[random.randrange(self.action_size)]], dtype=torch.long)

    def store_transition(self, state, action, next_state, reward, done):
        self.memory.append((state, action, next_state, reward, done))

    def optimize_model(self):
        if len(self.memory) < BATCH_SIZE:
            return
        transitions = random.sample(self.memory, BATCH_SIZE)
        batch = Transition(*zip(*transitions))

        non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.next_state)),
dtype=torch.bool)
        non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
        state_batch = torch.cat(batch.state)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)

        state_action_values = self.policy_net(state_batch).gather(1, action_batch)

        next_state_values = torch.zeros(BATCH_SIZE)

        next_state_values[non_final_mask] = self.target_net(non_final_next_states).max(1)[0].detach()
        expected_state_action_values = (next_state_values * self.gamma) + reward_batch

        loss = nn.functional.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze(1))

        self.optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_value_(self.policy_net.parameters(), 5)
        self.optimizer.step()

episode_durations = []

def plot_durations(show_result=False):
    plt.figure(1)
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    if show_result:
        plt.title('Result')
```

```
  else:
      plt.clf()
      plt.title('Training...')
  plt.xlabel('Episode')
  plt.ylabel('Duration')
  plt.plot(durations_t.numpy())
  # Take 100 episode averages and plot them too
  if len(durations_t) >= 100:
      means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
      means = torch.cat((torch.zeros(99), means))
      plt.plot(means.numpy())

  plt.pause(0.001)  # pause a bit so that plots are updated
  if is_ipython:
      if not show_result:
          display.display(plt.gcf())
          display.clear_output(wait=True)
      else:
          display.display(plt.gcf())

if __name__ == "__main__":
  env = gym.make('CartPole-v1')
  state_size = env.observation_space.shape[0]
  action_size = env.action_space.n
  agent = DQNAgent(state_size, action_size)

  scores = []
  episodes = []
  counter = 0
  for e in range(EPISODES):
      state = env.reset()
      state = torch.tensor(state, dtype=torch.float32).view(1, -1)
      done = False
      score = 0
      while not done:
        counter+=1
        action = agent.select_action(state)
        next_state, reward, done, _ = env.step(action.item())
        score += reward
        reward = torch.tensor([reward], dtype=torch.float32)
        next_state = torch.tensor(next_state, dtype=torch.float32).view(1, -1)
        if done:
            next_state = None
        agent.store_transition(state, action, next_state, reward, done)
        state = next_state
        agent.optimize_model()

        target_net_state_dict = agent.target_net.state_dict()
        policy_net_state_dict = agent.policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key]*TAU + target_net_state_dict[key]*(1-
TAU)
        agent.target_net.load_state_dict(target_net_state_dict)

        if done:
            episode_durations.append(counter+1)
            scores.append(score)
            episodes.append(e)
            plot_durations()
      counter = 0
      if e % TARGET_UPDATE == 0:
          agent.target_net.load_state_dict(agent.policy_net.state_dict())

  print('Complete')
  plot_durations(show_result=True)
  plt.ioff()
  plt.show()
```