Reinforcement Learning Project

WordZapper

Reinforcement Learning Agents on Word Zapper:

Using DQN and PPO

Project Team

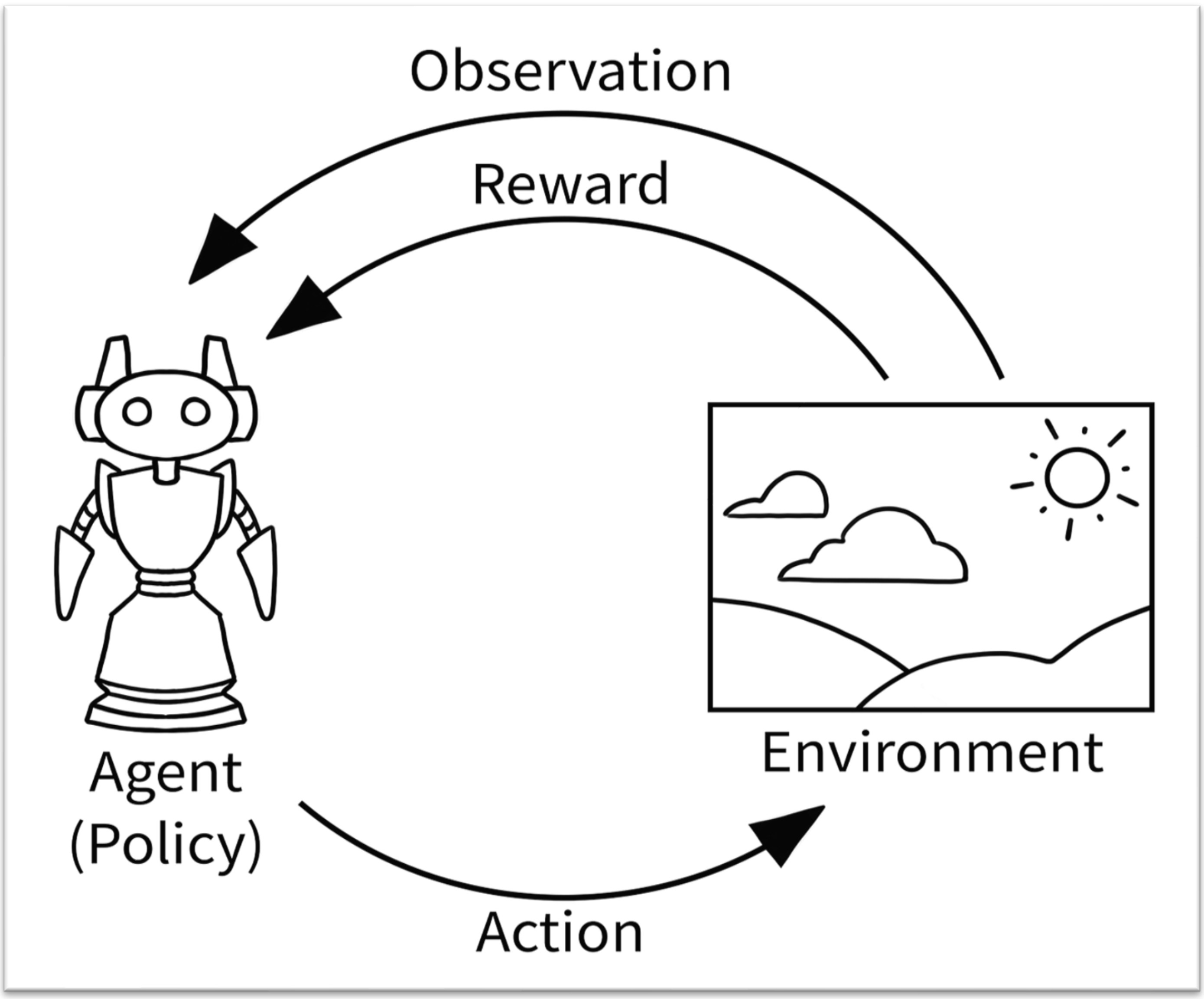
1-	Mohamed Anwar Hosny	3
2-	Mohamed Ibrahim Abdelfatah	3
3-	Ahmed Raffat feshar	1
4-	Thomas Basem	2
5-	Mohamed Ahmed Elnashar	3
6-	Abdelrahman Maged	2
7-	Abdallah Nasser Atta	3
8-	Fady Fayez	3

Objective

Our objective was to thoroughly explore the application of reinforcement learning (RL) algorithms in classic gaming environments. Specifically, we implemented and compared two popular RL algorithms—Deep Q-Network (DQN) and Proximal Policy Optimization (PPO)—on the Word Zapper Atari game.

The core aim of this project was to design intelligent agents capable of playing the game effectively by learning how to shoot letters that match the target word, using only image-based observations as input.

We investigated agent behaviour, training performance, and generalization while working within the constraints of the original game environment. Through this project, we aimed not only to evaluate the effectiveness of different RL strategies but also to highlight practical limitations and identify areas for improvement in the reinforcement learning workflow.

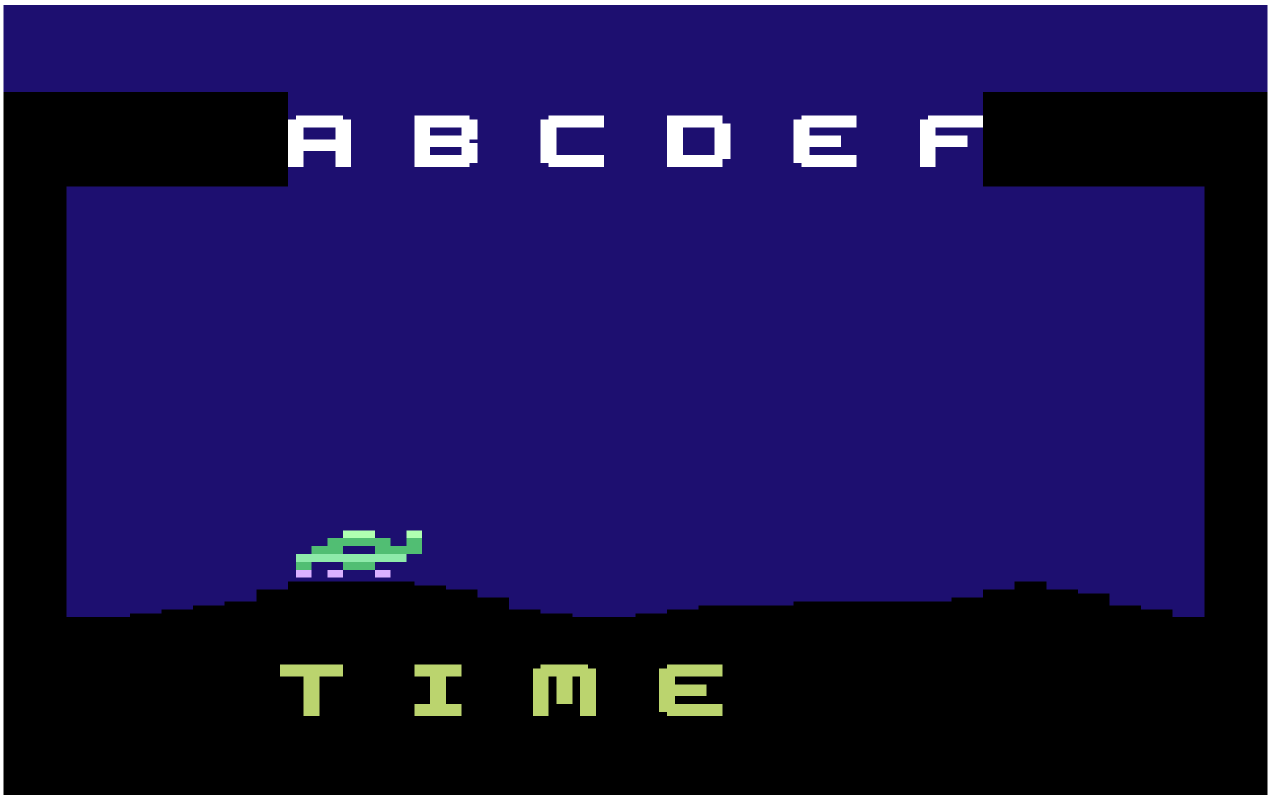


1. Introduction (game overview)

Word Zapper is a game in which the player controls a shooter and must hit letters that match a target word displayed at the top of the screen. The letters continuously pass across the screen in different directions, and the player must shoot them in time to spell the correct word. In this project, we assess how two well-established reinforcement learning methods—DQN and PPO—perform in this complex yet visually minimalistic setting. Unlike other approaches that involve modifying or simplifying the game, we utilized the game environment in its original form, incorporating reinforcement learning wrappers and leveraging open-source tools for integration. Our primary interest was to understand how effectively deep RL algorithms can learn to process visual information and perform strategic decision-making under time constraints and sparse feedback.

This environment is part of the Atari environments.

Action Space	Discrete(18)
Observation Space	Box(0, 255, (210, 160, 3), uint8)
Import	`gymnasium.make("ALE/WordZapper-v5")`



2. Problem Definition

- **Observation:** Agents receive raw image frames representing the current visual state of the game, which must be interpreted and processed to extract meaningful signals.

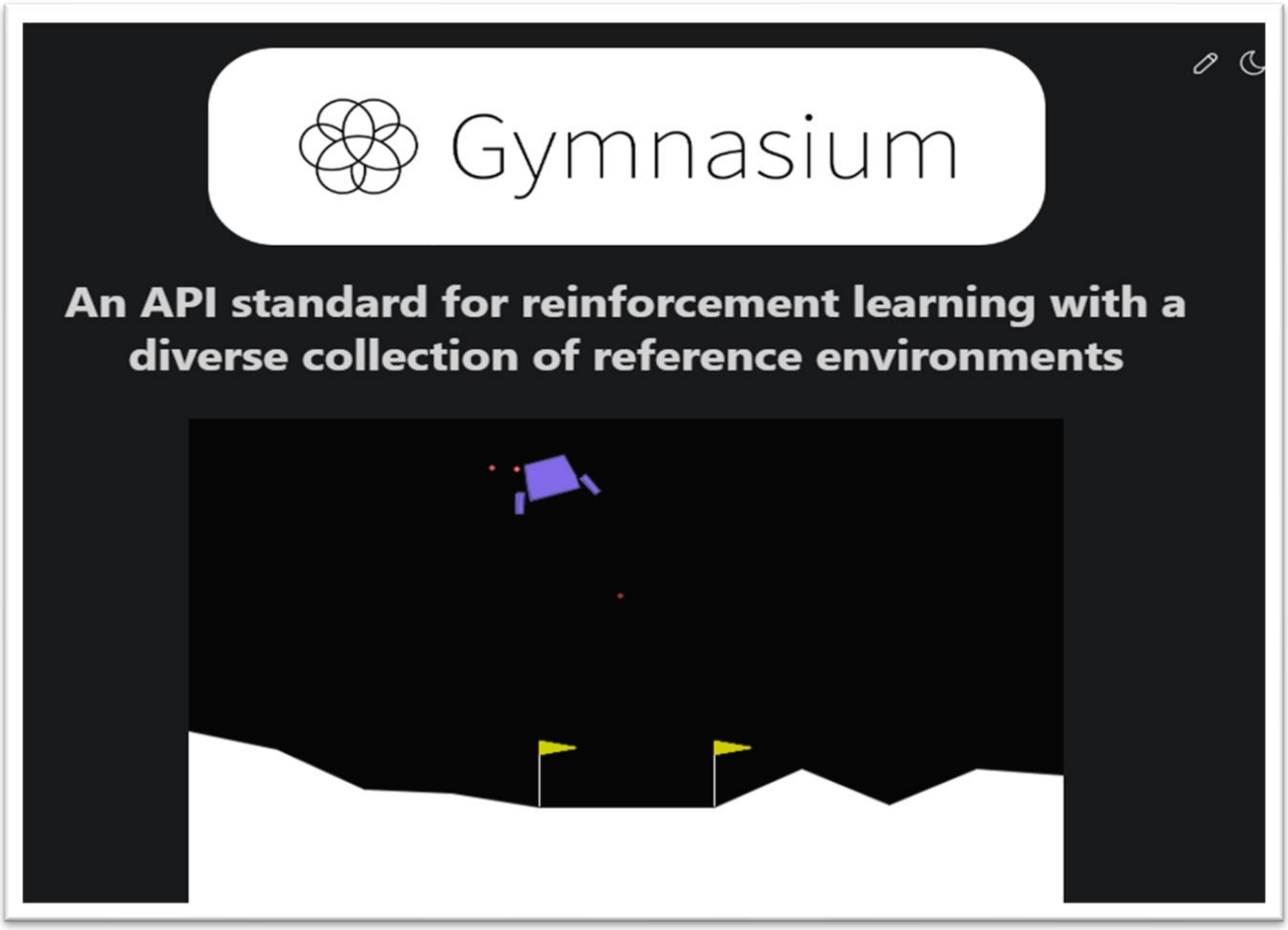
- **Action Space:** A set of discrete actions including shooting left, shooting right, or remaining idle. These correspond to control decisions the agent can make at each time step.

- **Reward Signal:** A positive reward is given for shooting correct letters that belong to the target word, while incorrect shots result in no reward or a penalty (negative reward). This sparse and delayed feedback structure makes the learning problem non-trivial.

- **Challenges:**
  - Visual input is high-dimensional and requires downscaling and preprocessing.
  - The agent must make decisions based on limited temporal context due to frame skipping.
  - Sparse and delayed reward signals make it difficult to determine effective actions.
  - Transitioning between words mid-game introduces discontinuity in the learning task, confusing the learning policy.

3. Tools and Technologies

Tool / Library	Purpose
Gymnasium	Provides a standardized interface for RL environments
ALE-py	Offers an interface to the Atari Learning Environment
Stable-Baselines3	Implementation of RL algorithms such as PPO and DQN
AutoROM	Automatically installs and manages Atari ROMs
Matplotlib	Used to visualize training curves and performance metrics
Python	Core scripting and orchestration language for building experiments
OpenCV	Used for optional image preprocessing and debugging visuals


Gymnasium
An API standard for reinforcement learning with a diverse collection of reference environments

4. Methodology

4.1. Preprocessing

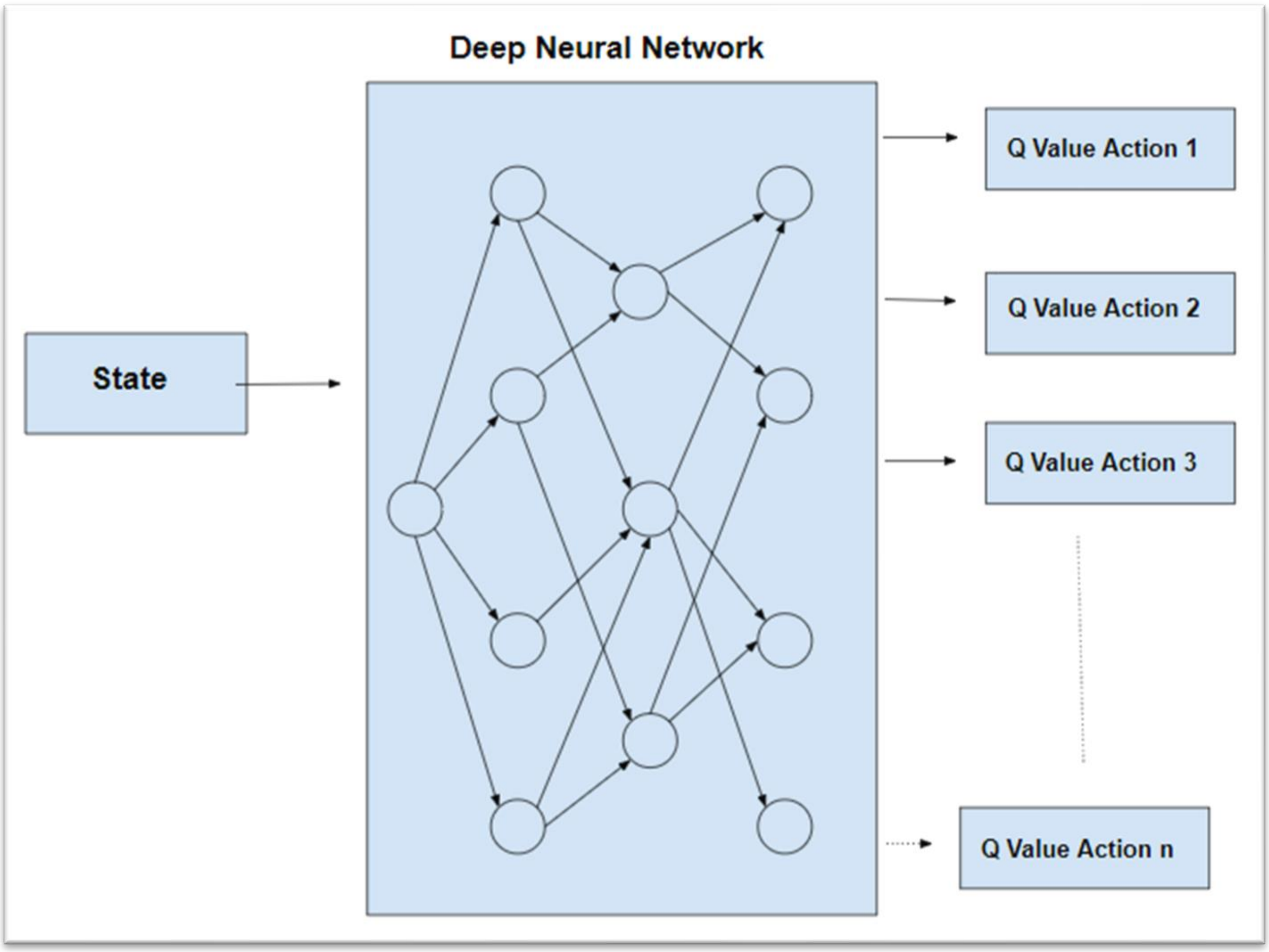
To handle the visual complexity of Atari games, we used standard wrappers provided by Stable-Baselines3 and ALE-py:

- **AtariWrapper**:
  - Downsamples frames and converts them to grayscale.
  - Normalizes pixel values to [0, 1].
  - Stacks four consecutive frames to preserve temporal dynamics and allow the agent to infer motion.

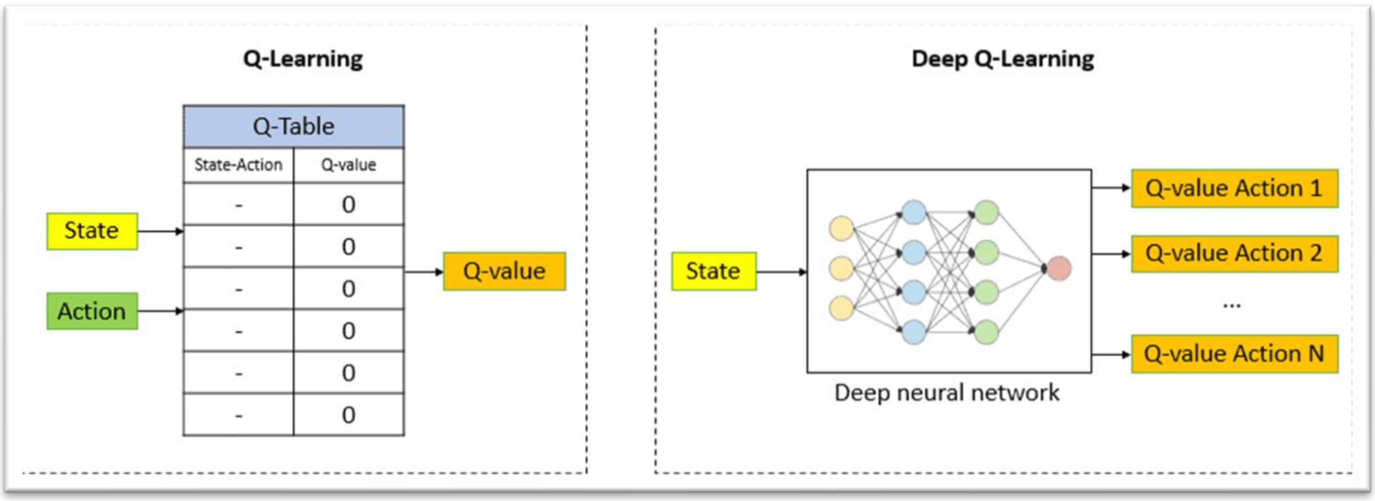
- The environment was wrapped using DummyVecEnv for compatibility with Stable-Baselines3's parallelized environments.

- Additional options for frame skipping and early episode termination were considered for optimization.

4.2. Algorithm 1: Deep Q-Network (DQN)



Compared to Q-learning

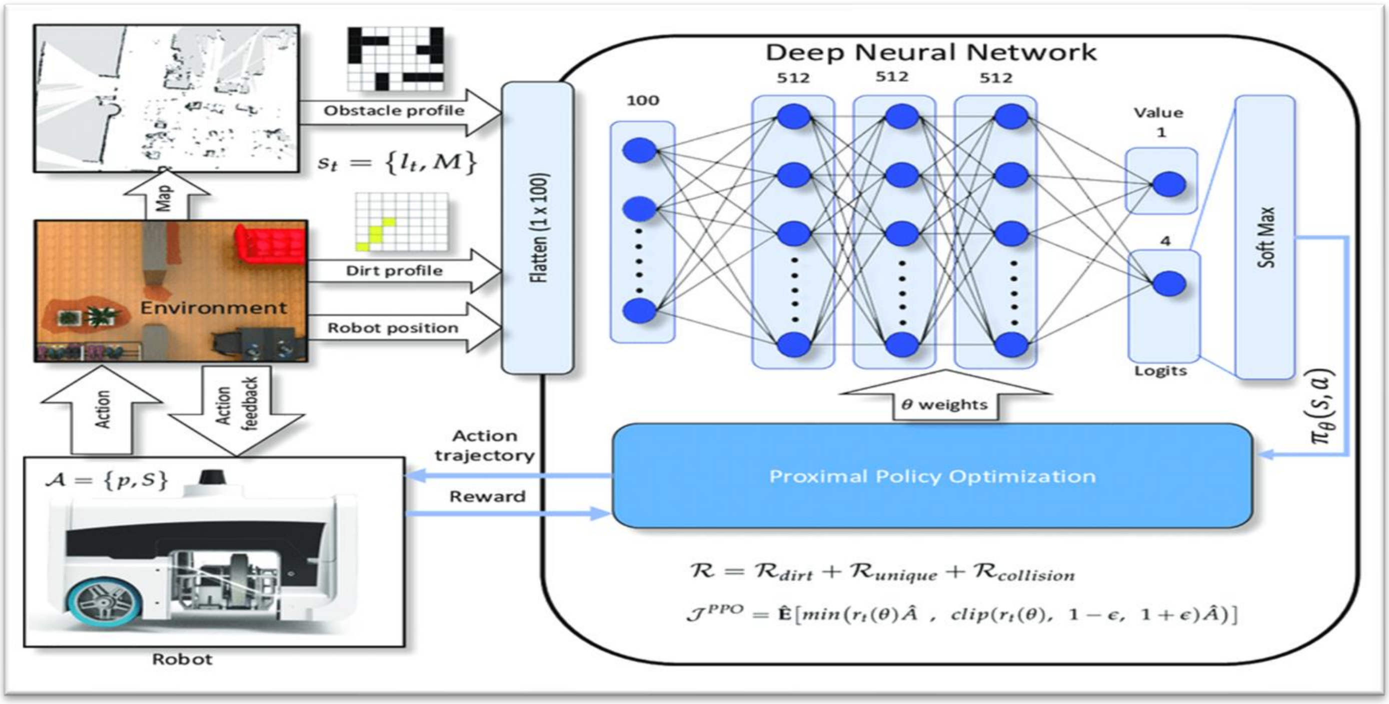


- **Architecture:** A convolutional neural network (CNN)-based Q-network is used to approximate the action-value (Q) function.

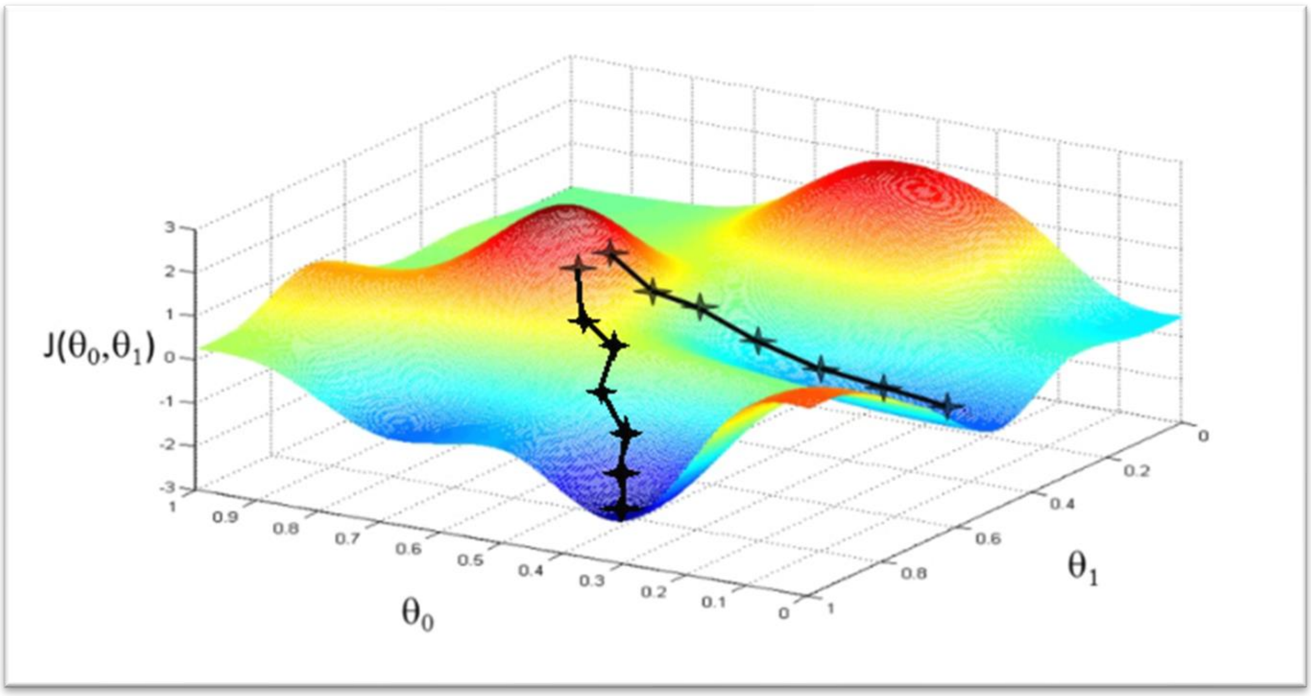
- **Core Techniques:**
  - Experience Replay to break correlations in sequential data.
  - Fixed Q-targets to stabilize training by decoupling target generation from the current Q-network.
  - Epsilon-greedy strategy to balance exploration and exploitation.

- **Training Procedure:**
  - The agent was trained over 2 million timesteps using the Stable-Baselines3 implementation.
  - We monitored the training using reward plots, episode length summaries, and video evaluations.

4.3. Algorithm 2: Proximal Policy Optimization (PPO)



- **Directly learns a policy using gradient ascent to maximize reward.**
  - Unlike value-based methods, PPO optimizes the policy directly without the intermediate step of learning Q-values.
  - It iteratively updates policy parameters in the direction that increases expected reward.



- **Uses a "surrogate" objective function to update the policy.**
    - This function compares the likelihood of actions under the new policy versus the old policy.
    - It's multiplied by an estimate of the advantage (how much better an action is compared to the average).

- **Employs a clipping mechanism to limit the size of policy updates.**
    - This is where the term "proximal" comes from - it restricts how much the policy can change in a single update.
    - This promotes stability and prevents drastic policy shifts that could lead to performance collapse.
- **Uses an Actor-Critic architecture, including a value function estimate (Critic) to reduce variance in policy gradients.**
    - The Actor (policy) determines action selection.
    - The Critic (value function) aids in estimating the quality of actions and states.

- **Works with probability distributions instead of Q-tables or deterministic values.**
    - Unlike Q-learning or DQN which output single values for actions, PPO produces a distribution over actions.
    - This allows for more nuanced policies, especially in complex or continuous action spaces.
    - The probabilistic approach enables better handling of uncertainty and exploration-exploitation trade-offs.

5. Evaluation

5.1. Quantitative Metrics

During PPO training, a variety of performance and training metrics were tracked to assess the agent’s learning progress. For example, the following snapshot from the training output illustrates the detailed training statistics at various stages:

```
----------------------------
| time/              |     |
|    fps             | 352 |
|    iterations      | 1   |
|    time_elapsed    | 1   |
|    total_timesteps | 512 |
----------------------------
-------------------------------------------
| time/                   |               |
|    fps                  | 239           |
|    iterations           | 2             |
|    time_elapsed         | 4             |
|    total_timesteps      | 1024          |
| train/                  |               |
|    approx_kl            | 4.2477623e-06 |
|    clip_fraction        | 0             |
|    clip_range           | 0.2           |
|    entropy_loss         | -2.89         |
|    explained_variance   | -3.82         |
|    learning_rate        | 1e-06         |
|    loss                 | 0.00055       |
|    n_updates            | 10            |
|    policy_gradient_loss | -0.000276     |
|    value_loss           | 0.00299       |
-------------------------------------------
...
|    n_updates            | 19530        |
|    policy_gradient_loss | -0.00161     |
|    value_loss           | 9.41e-05     |
------------------------------------------
```

Key observations from the output include:

- **Frame Rate (fps):** Initially recorded at 352 fps, then slightly dropping to 239 fps, which can indicate changes in computational load as the training progresses.

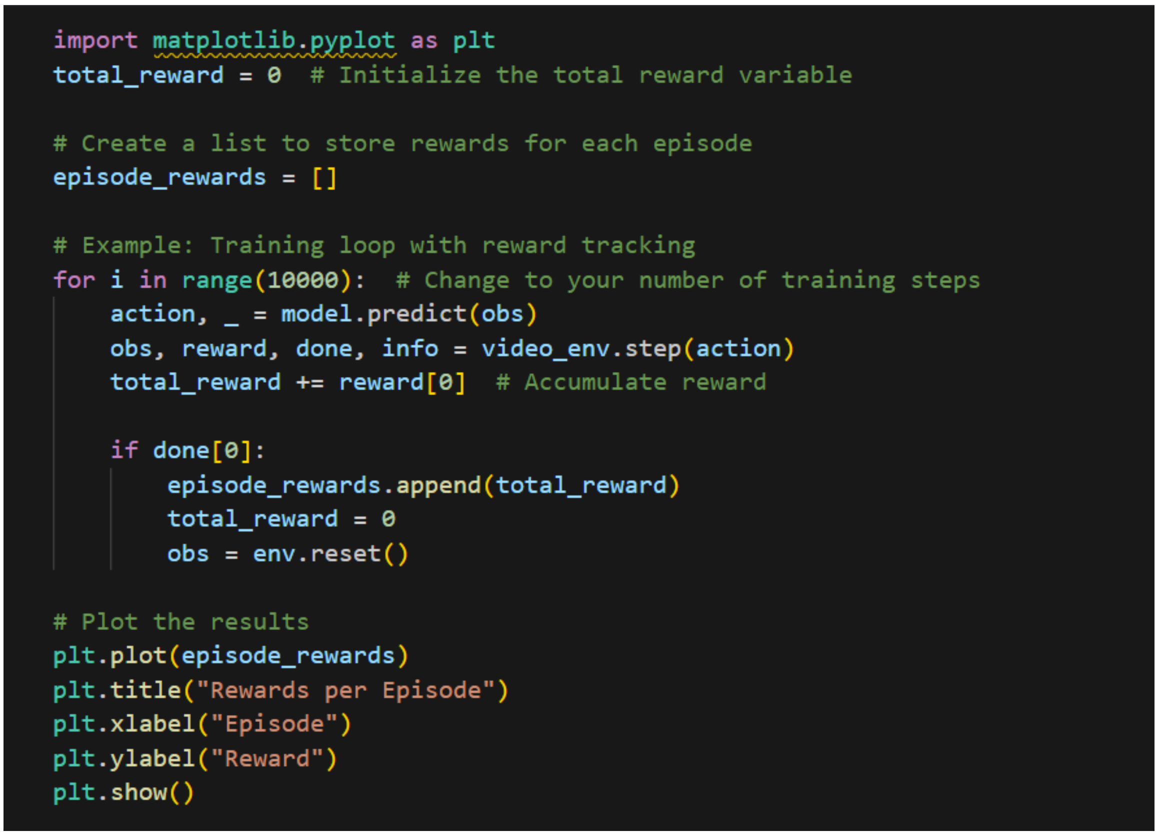
- **Iterations and Timesteps:** The training output logs iterations and cumulative timesteps, which help in monitoring the progress towards the 2-million timestep goal.

- **Approximate KL Divergence (approx_kl):** Extremely low values (e.g., 4.2477623e-06) indicate that policy updates are well-controlled, a sign of stable training.

- **Clip Fraction and Range:** A clip fraction of 0 with a clip range of 0.2 shows that during these iterations, the updates remained within the desired bounds, ensuring smooth policy updates.

- **Loss Metrics:** The entropy loss, policy gradient loss, and value loss are tracked, providing insights into the optimization process:
  - **Entropy Loss:** A negative value (e.g., -2.89) is common and indicates that the agent is maintaining sufficient exploration.
  - **Policy Gradient Loss and Value Loss:** These metrics (e.g., -0.000276 and 0.00299) give an indication of the error between the predicted and target values, which decreases as training converges.

- **Number of Updates:** As training progresses (e.g., 19530 updates), the evolution of these loss metrics helps determine the convergence behavior of the model.



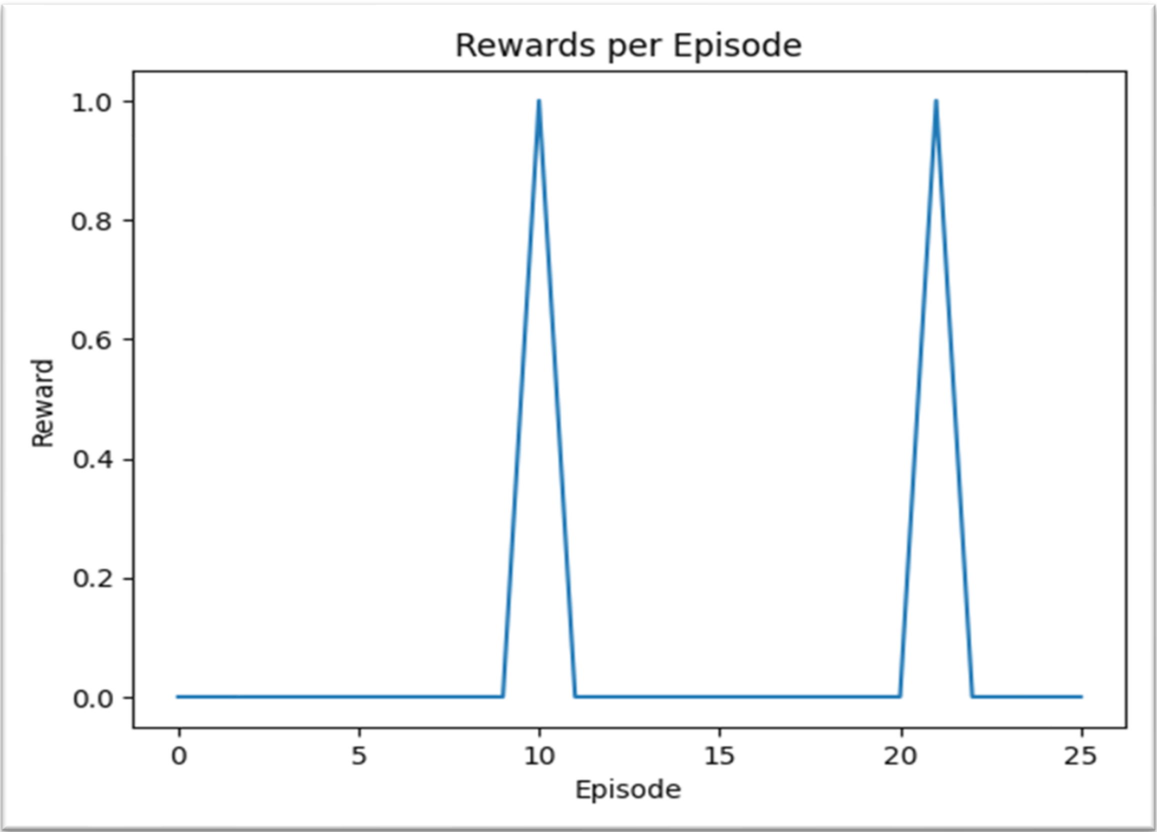
```python
import matplotlib.pyplot as plt
total_reward = 0  # Initialize the total reward variable

# Create a list to store rewards for each episode
episode_rewards = []

# Example: Training loop with reward tracking
for i in range(10000):  # Change to your number of training steps
    action, _ = model.predict(obs)
    obs, reward, done, info = video_env.step(action)
    total_reward += reward[0]  # Accumulate reward

    if done[0]:
        episode_rewards.append(total_reward)
        total_reward = 0
        obs = env.reset()

# Plot the results
plt.plot(episode_rewards)
plt.title("Rewards per Episode")
plt.xlabel("Episode")
plt.ylabel("Reward")
plt.show()
```



This detailed output is essential to understanding both the performance and stability of the PPO algorithm during training. These metrics are then compared with those obtained from the DQN training to provide a comprehensive view of the agent's performance under different RL approaches.

5.2. Qualitative Insights

- **Training Stability:**
  PPO maintained stable learning with controlled policy updates, as indicated by the low KL divergence and controlled loss metrics. The recorded metrics demonstrate that the model did not experience significant policy shifts, which is crucial for on-policy methods.

- **Agent Behavior:**
  Video recordings and reward plots further confirmed that the PPO agent learned to effectively recognize and shoot correct letters. Over time, cumulative rewards increased steadily, reflecting improved performance.

- **Comparison with DQN:**
  While DQN showed rapid early learning, its performance plateaued due to the challenges of handling sparse and delayed rewards. In contrast, PPO provided more robust long-term performance, which is corroborated by its detailed training metrics and smoother learning curves.

6. Key Challenges

- **Perception:** Agents needed to parse the top-screen word and identify matching letters among several distractors.
- **Generalization:** Memorizing specific visual features led to brittle policies when letter placements varied.
- **Environment Constraints:** The environment only awarded rewards for hitting letters in strict sequence, making the task harder.
- **Reward Shaping:** Sparse feedback delayed the learning curve and introduced local optima challenges.

7. Conclusion

This project successfully applied two leading reinforcement learning algorithms to a challenging Atari game. PPO stood out in terms of training stability and long-term performance, while DQN delivered quick early learning but struggled with the complexity of reward scheduling.

Our work demonstrates the potential of deep RL even in visually abstract tasks and lays the groundwork for future extensions that could involve custom environments or enhanced game dynamics. The insights gained from this study emphasize the value of model architecture choice, training stability, and the critical importance of the underlying reward structure in determining overall agent success.

8. Future Work

- Develop a customized game environment that relaxes restrictions on letter shooting order.
- Investigate advanced DQN variants such as Double DQN and Dueling DQN.
- Combine PPO with auxiliary learning signals such as curiosity or count-based exploration.
- Implement curriculum learning strategies that introduce simpler tasks before full gameplay.
- Explore model explainability methods to understand what features the agents are learning.
- Experiment with reward shaping techniques to improve convergence speed and performance robustness.
- Test models under adversarial inputs or noisy frames to assess real-world generalization.

9. References

- ⍰ **Stable-Baselines3** – Implementations of RL algorithms in PyTorch (used for PPO and DQN in your project).
  🔗 https://github.com/DLR-RM/stable-baselines3
  📘 Docs: https://stable-baselines3.readthedocs.io/
- ⍰ **Gymnasium (formerly OpenAI Gym)** – Standard RL interface used to manage the game environment.
  🔗 https://github.com/Farama-Foundation/Gymnasium
  📘 Docs: https://gymnasium.farama.org/
- ⍰ **ALE-py (Arcade Learning Environment)** – Interface for running Atari games like Word Zapper.
  🔗 https://github.com/mgbellemare/Arcade-Learning-Environment
  📘 Docs: https://ale-py.readthedocs.io/
- ⍰ **AutoROM** – Utility that downloads and sets up Atari ROMs legally.
  🔗 https://github.com/Farama-Foundation/AutoROM