LLM Agents & Deep Q-Learning with Atari Games

Atari: `ALE/Zaxxon-v5`

Nithin Yash Menezes NU ID: 002498030

November 6, 2025

Overview

This project implements a Deep Q-Learning (DQN) agent on a non-toy Atari environment, **Zaxxon** (`ALE/Zaxxon-v5`), using Gymnasium (ALE-Farama). The work includes: a functioning DQN with experience replay and target updates, analysis of design choices, theoretical answers on Q-learning and LLM agents, an alternative exploration policy, and a 60 fps video demonstration recorded via `RecordVideo`.

Colab Notebook (code & runs) https://colab.research.google.com/drive/1tJzHp9MaNftm4W9pUPZy scrollTo=J7mLvANtTmrN

Environment/Tools Google Colab (GPU), `gymnasium`, `ale-py`, `autorom`, `torch`, `opencv-python`, `moviepy`, `tqdm`, `pandas`. Atari envs registered via `gym.register_envs(ale_py)`; AutoROM used to install ROMs with accepted license.

1 Functional Requirements

1.1 Baseline Performance

Environment: `ALE/Zaxxon-v5` (observation RGB 210×160, action space `Discrete(18)`; we use preprocessed grayscale and frame stacking).
Baseline DQN settings (as implemented):

- CNN encoder (Atari-style): conv layers $\rightarrow$ 3136-dense $\rightarrow$ 512 ReLU $\rightarrow$ $|\mathcal{A}|$ outputs.
- Replay buffer: 10k; batch size: 32; optimizer: Adam (LR $= 1\times10^{-4}$); loss: MSE.
- Discount $\gamma = 0.99$; target net sync each episode; ϵ-greedy: start $1.0 \rightarrow 0.05$ with decay 0.995.
- Preprocessing: *grayscale*, *resize* 84×84, *4-frame stack*.

Observed behavior in runs of 30–200 episodes: as exploration decays, survival length increases and early sparse-reward pickup improves; performance remains modest (as expected for short training on a challenging shooter), while stability is good.

1.2 Environment Analysis

States: pixel observations preprocessed to $s \in \mathbb{R}^{4\times84\times84}$ (stack of 4 grayscale frames) to capture dynamics.
Actions: `Discrete(18)` including `NOOP`, `FIRE`, directional and diagonal combinations (with FIRE variants).
Q-table size: intractable for pixel states; function approximation (CNN) is used to represent $Q_\theta(s,a)$.

1.3 Reward Structure

Native ALE Zaxxon rewards are used (positive for enemy/objective hits, negative on crash/death, often zero otherwise). For the stronger trainer, optional reward clipping to $[-1, 1]$ was enabled to stabilize training while preserving sign structure.

1.4 Bellman Equation Parameters and Variants

The baseline target:

$$y = r + \gamma \max_{a'} Q_{\text{target}}(s', a').$$

We experimented with:

- **Lower γ (0.95):** slightly favors short-horizon tactics; can improve early gains but may underweight longer strategies.
- **Lower LR (6.25×10^{-5}):** smoother updates with slower learning dynamics.

We also implemented a **stronger trainer**:

$$y = r + \gamma(1 - d)\, Q_{\text{target}}\big(s', \arg\max_{a'} Q_{\text{policy}}(s', a')\big),$$

i.e., *Double-DQN* target with *Huber loss*, gradient clipping, a larger replay buffer, and a per-step ϵ schedule. This reduced value overestimation and stabilized returns.

1.5 Policy Exploration (beyond ϵ-greedy)

In addition to ϵ-greedy, we evaluated **Softmax/Boltzmann** exploration:

$$P(a \,|\, s) = \frac{\exp(Q(s, a)/\tau)}{\sum_b \exp(Q(s, b)/\tau)},$$

which provided more graded action diversity early on and sometimes avoided getting stuck in poor local behaviors at the cost of slower early exploitation.

1.6 Exploration Parameters

Baseline: ϵ decays *per-episode* from 1.0 to 0.05 with factor 0.995. In the stronger trainer, a *per-step linear* schedule drives ϵ towards 0.05 over a long horizon (e.g., 500k steps). Faster decay pushes earlier exploitation; slower decay encourages broader exploration in this sparse-reward shooter. End-of-episode ϵ values are reported in logs.

1.7 Performance Metrics

We report episode returns, lengths, and end-of-episode ϵ in Colab logs. In 30–200 episode runs, average steps/episode increased across training while returns remained modest (consistent with short wall-clock training on Zaxxon). A longer run with the stronger trainer further stabilized learning and produced visibly smoother policy rollouts.

1.8 Q-Learning Classification

Q-learning/DQN is **value-based**. The agent learns $Q(s, a)$ and acts greedily (modulo exploration). It does not directly optimize a parameterized policy π_θ; policy-based methods (e.g., REINFORCE, PPO) do.

1.9 Deep Q-Learning vs. LLM Agents

DQN optimizes discounted return via explicit environment interaction with replay and target networks; the world model is implicit in value function approximation. LLM agents optimize token likelihood and/or a reward model (e.g., RLHF) over language/tool trajectories; they plan through text decomposition, tool use, and reflection, with evaluation tied to preferences or task success rather than episodic return.

1.10 Bellman: Expected Lifetime Value

The state value under policy π is the expected discounted sum of future rewards:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \,\middle|\, s_0 = s, \pi\right], \qquad Q^{\pi}(s,a) = \mathbb{E}[r_1 + \gamma V^{\pi}(s_1) \mid s_0 = s, a_0 = a]\,.$$

DQN approximates $Q^{(s,a)}$; greedy action selection over Q yields an optimal policy.

1.11 RL Concepts for LLM Agents

Reward shaping, curriculum design, off-policy reuse of logged interactions, and credit assignment map naturally to LLM training and agentic workflows (e.g., RLHF/DPO; tool-use agents with self-reflection and preference feedback).

1.12 Planning in RL vs. LLM Agents

Traditional RL planning uses explicit dynamics (model-based rollouts, value iteration, MCTS). LLM planning orchestrates chain-of-thought, subgoal decomposition, and tool calls; a verifier/-critic can score plans. In our case, DQN plans low-level control in Zaxxon; an LLM could act as a high-level mission planner specifying subgoals for the DQN controller.

1.13 Q-Learning Algorithm (Pseudo/Math)

Double-DQN target used in the stronger trainer:

$$y = r + \gamma(1-d)\, Q_{\text{target}}\big(s', \arg\max_{a'} Q_{\text{policy}}(s', a')\big).$$

Sketch:

```
Initialize  (policy), ¯ (target ← ), replay D
for each episode:
  s ← 4-frame stack from preprocessed observations
  while not done:
    a ← -greedy( Q(s) ) or softmax(Q/)
    s', r, d ← env.step(a); push (s,a,r,s',d) to D
    if len(D) > warmup:
      sample batch; compute Double-DQN targets; Huber/MSE loss
      backward; clip grads; optimizer step
    periodically: ¯ ←
    s ← s'
save  as weights (.pt)
```

1.14 LLM & DQN Integration

A practical hybrid: an **LLM planner** translates a natural-language objective into waypoints/-subgoals; the **DQN controller** executes pixel-wise control; a **critic** (LLM or learned reward model) scores rollouts to refine prompts and subgoals. This architecture enables instruction-following control in visual environments.

1.15 Code Attribution

Authored in this project (Colab):

- Baseline trainer `dqn_zaxxon.py`: CNN Q-network, replay, target sync, ϵ-greedy, Atari pre-processing, saving `dqn_zaxxon.pt`.
- Stronger trainer: Double-DQN target, Huber loss, gradient clipping, large replay, per-step ϵ, target scheduling, resumable checkpoints.
- Recording/evaluation utilities: 60 fps `RecordVideo` longplay and `eval_policy`.

External libraries/APIs used: Gymnasium/ALE-py/AutoROM, PyTorch, OpenCV, MoviePy. No external implementation code was copied; libraries were used via public APIs.

1.16 Code Clarity

PEP-8 style, modular functions: `preprocess`, `train_dqn/train_zaxxon`, `record_longplay`, `eval_policy`. Logs provide *return*, *episode length*, *epsilon*, *step counts*. Checkpoints: `checkpoints/zaxxon_ep*` plus `dqn_zaxxon.pt`.

1.17 Licensing

Original code released under the **MIT License**. Libraries retain their licenses (`gymnasium`, `ale-py`, `autorom`, `torch`, `opencv-python`, `moviepy`).

1.18 Professionalism

Consistent naming, clear notebook flow (install → env check → baseline trainer → stronger trainer → 60 fps video → evaluation), explicit attribution, and reproducible steps.

Appendix A: Key Implementation Snippets

```
class DQN(nn.Module):
    def __init__(self, n_actions):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(4, 32, 8, 4), nn.ReLU(),
            nn.Conv2d(32, 64, 4, 2), nn.ReLU(),
            nn.Conv2d(64, 64, 3, 1), nn.ReLU()
        )
        self.head = nn.Sequential(
            nn.Linear(7*7*64, 512), nn.ReLU(),
            nn.Linear(512, n_actions),
        )
    def forward(self, x):
        x = x / 255.0
        x = self.conv(x)
        return self.head(x.view(x.size(0), -1))
```

```
def preprocess(obs):
    g = cv2.cvtColor(obs, cv2.COLOR_RGB2GRAY)
    return cv2.resize(g, (84, 84))
# Stack length = 4 (deque), used in all train/eval/record loops
```

```
mp4 = record_longplay(weights="dqn_zaxxon.pt", env_id="ALE/Zaxxon-v5",
                      episodes=3, fps=60)
```

Appendix B: Reproducibility (Colab Steps)

1. Install & ROMs:

```
pip install "gymnasium[accept-rom-license]" "gymnasium[atari,accept-rom-license]" \
  "ale-py" "autorom[accept-rom-license]" opencv-python moviepy tqdm pyyaml pandas==2.2.2
AutoROM --accept-license
```

2. Register: `gym.register_envs(ale_py)`; verify `ALE/Zaxxon-v5`.
3. Train baseline (`dqn_zaxxon.py`) → produces `dqn_zaxxon.pt`.
4. Stronger trainer: Double-DQN + Huber + large buffer + resume.
5. Record at 60 fps with `RecordVideo`; evaluate via `eval_policy`.

1.19 Code Attribution and Licensing

All implementation work, including model architecture, replay buffer, training loop, preprocessing, and gameplay recording, was written by **Nithin Yash Menezes** within a Google Colab environment for the INFO 7375: LLM Agents & Deep Q-Learning Assignment.

The project utilizes the following open-source libraries:

- **Gymnasium**, **ALE-py**, and **AutoROM** — licensed under the MIT License (Farama Foundation).
- **PyTorch** — BSD-style License (Meta AI).
- **OpenCV** — Apache License 2.0.
- **MoviePy** — MIT License.
- **NumPy**, **Pandas**, and **TQDM** — BSD License.

All code in the file `dqn_zaxxon.py`, the training notebook, and the recording pipeline was authored independently. Conceptual references were drawn from:

- Mnih et al. (2015), *Human-level control through deep reinforcement learning.*
- Official documentation of Gymnasium Atari environments and PyTorch tutorials.

All original code contributions are released under the **MIT License**, allowing reuse and modification with appropriate credit.