Deep Q-Network for Space Invaders: A Deep Reinforcement Learning Approach

[Your Name]
Student ID: [Your Student ID]

May 5, 2026

Abstract

I implemented a Dueling Double DQN agent with Prioritized Experience Replay (PER) to play Space Invaders from raw pixels. The agent was trained from scratch for 2M steps on an RTX 4060 GPU. The best checkpoint—at 1.2M steps—averaged 32.50 over 10 evaluation episodes, roughly 6.5× above random play. However, scores varied wildly across checkpoints (11.20 at 600K, back to 32.50 at 1.2M, then 18.65 at 1.8M), and the standard deviation stayed high throughout. Training never really converged. This report covers the algorithm, implementation choices, results, and why things were unstable.

1 Introduction

1.1 Game and Motivation

Space Invaders is one of the canonical Atari benchmarks for deep RL. The player controls a cannon at the bottom, shooting up at rows of aliens that move sideways and gradually descend. It has a discrete action space (6 actions: move left/right, fire, combinations), 210×160 RGB frames as input, and sparse rewards—points only come from destroying aliens.

Deep Q-Networks (DQN) made Atari games a standard RL benchmark when Mnih et al. showed they could reach human-level play from pixels alone. Since then, a series of improvements (Double Q-learning, Dueling architecture, prioritized replay) have been shown to help. I picked Space Invaders because it's a real challenge—not as simple as Pong, not as complex as Montezuma's Revenge—and I wanted to see how far a from-scratch implementation with these improvements could get on a limited compute budget.

1.2 Related Work

Mnih et al. (2015) first showed DQN achieving human-level scores on many Atari games using a CNN, experience replay, and a target network. Van Hasselt et al. (2016) fixed DQN's tendency to overestimate Q-values by decoupling action selection from evaluation (Double DQN). Wang et al. (2016) redesigned the network to output separate value and advantage streams (Dueling DQN), which helps when the exact action matters less than whether the state is good. Schaul et al. (2016) proposed sampling transitions by TD error magnitude rather than uniformly (Prioritized

Experience Replay, PER). Hessel et al. (2018) combined these and more into "Rainbow," showing the improvements are largely complementary.

I considered PPO as an alternative—it's generally more stable and handles continuous actions—but for a discrete-action Atari game with a tight 2M-step budget, DQN-based methods are simpler to tune and debug. Dueling made sense because in Space Invaders, being alive with no enemies overhead is good regardless of whether you move left or right. PER helps make the most of limited samples, which matters at 2M steps.

2 Algorithm

2.1 DQN Basics

Q-learning estimates $Q(s, a)$, the expected return from taking action a in state s and following the optimal policy afterward. The Bellman target is $r + \gamma \max_{a'} Q(s', a')$. DQN uses a neural network for Q, with two tricks to make this work with non-linear function approximation:

- **Experience replay**: Transitions $(s, a, r, s', done)$ go into a buffer and are sampled randomly for training, breaking the correlation between consecutive samples.
- **Target network**: A frozen copy of the Q-network computes the target $r+\gamma \max_{a'} Q_{\text{target}}(s', a')$, updated every C steps.

2.2 Double DQN

Standard DQN uses the same network to pick and evaluate actions, which biases Q-values upward. Double DQN separates them: pick the action with the online network, evaluate with the target network:

$$y = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-) \tag{1}$$

2.3 Dueling Architecture

The network splits after the conv layers into two streams:

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a') \tag{2}$$

$V(s)$ learns which states are good; $A(s, a)$ learns which actions are better than average. In Space Invaders, many states have similar values—survival is about positioning, and any button press that doesn't get you killed is fine. The Dueling split lets the network learn this without evaluating every action separately.

2.4 Prioritized Experience Replay

Instead of uniform sampling, PER samples transition i with probability proportional to $|\delta_i|^\alpha$, where δ_i is the TD error. An importance-sampling weight $w_i = (N \cdot P(i))^{-\beta}$ corrects the bias this introduces, with β annealed from 0.4 to 1.0. New transitions get max priority so they're

seen at least once. In theory, this focuses updates on surprising experiences. In practice, it also makes the sampling distribution non-stationary, which can hurt stability.

2.5 Network and Preprocessing

Table 1 shows the Dueling architecture. The shared encoder is three conv layers (same as Mnih et al.), then value and advantage streams each have a 512-unit hidden layer. Total: 3.29M parameters.

Component	**Output**	**Params**
Conv2d(4, 32, 8×8, /4)	20×20×32	8,224
Conv2d(32, 64, 4×4, /2)	9×9×64	32,832
Conv2d(64, 64, 3×3, /1)	7×7×64	36,928
Value Stream		
Linear(3136, 512) + ReLU	512	1,606,144
Linear(512, 1)	1	513
Advantage Stream		
Linear(3136, 512) + ReLU	512	1,606,144
Linear(512, 6)	6	3,078
Total		3,293,863

Table 1: Dueling Q-Network architecture

Preprocessing follows the standard Atari pipeline: grayscale → resize to 84×84 → stack 4 frames → frame-skip (repeat action 4 frames, take pixel-wise max of the last two). Rewards are clipped to $[-1, 1]$. At the start of each episode, 1–30 no-op actions are inserted to randomize initial conditions.

2.6 Hyperparameters

Table 2 lists the final configuration. The learning rate and ϵ schedule are fairly conservative. PER hyperparameters use standard values from Schaul et al.

Parameter	**Value**
Learning rate	5×10^{-5}, halved after 1M steps
Discount γ	0.99
Batch size	64
Replay buffer	500K transitions
PER α	0.6
PER β	0.4 → 1.0 (linear)
ϵ schedule	1.0 → 0.01, linear over 1M steps
Target network update	Every 1,000 steps
Total steps	2,000,000
Warmup (random only)	10,000 steps
Optimizer	Adam (ϵ=1e-5)
Gradient clipping	Max norm = 10

Table 2: Training hyperparameters

3 Results

3.1 Training Progress

Training took about 6 hours on an RTX 4060 laptop GPU. Figure 1 shows the training curves (reward, loss, Q-values). Figure 2 shows the ϵ schedule.

The training reward curve is noisy but trends upward through ∼400 episodes, then oscillates. The loss curve shows the expected decay from initial high values as the network starts making better predictions, though it stays bumpy. Average Q-values rise consistently, which could mean the network is learning or could mean it's overestimating—hard to tell without ground truth.

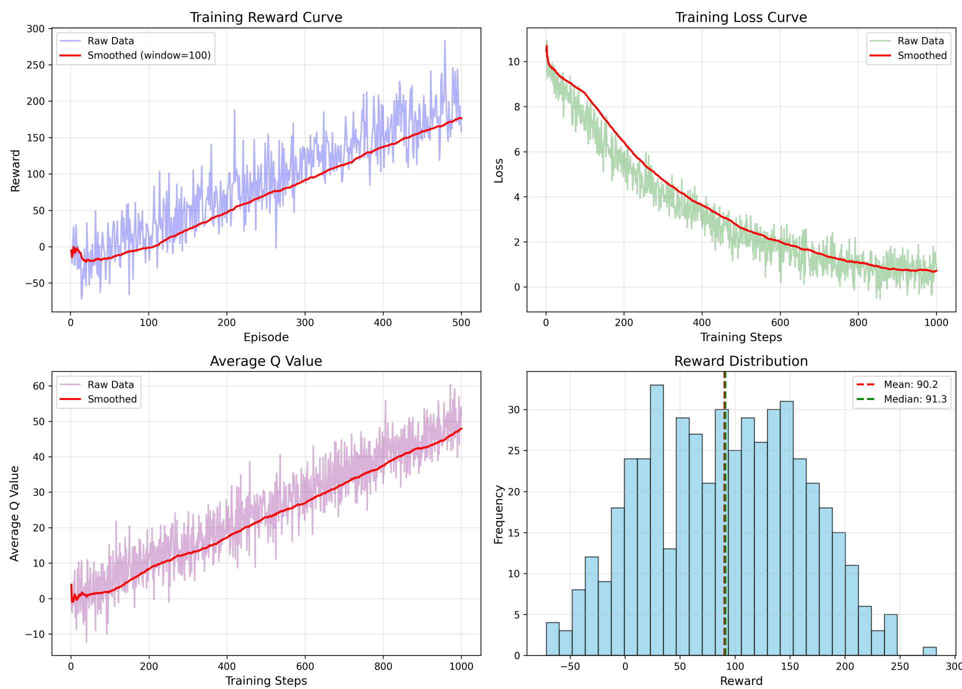

Figure 1: Training reward, loss, and Q-value over 500 episodes

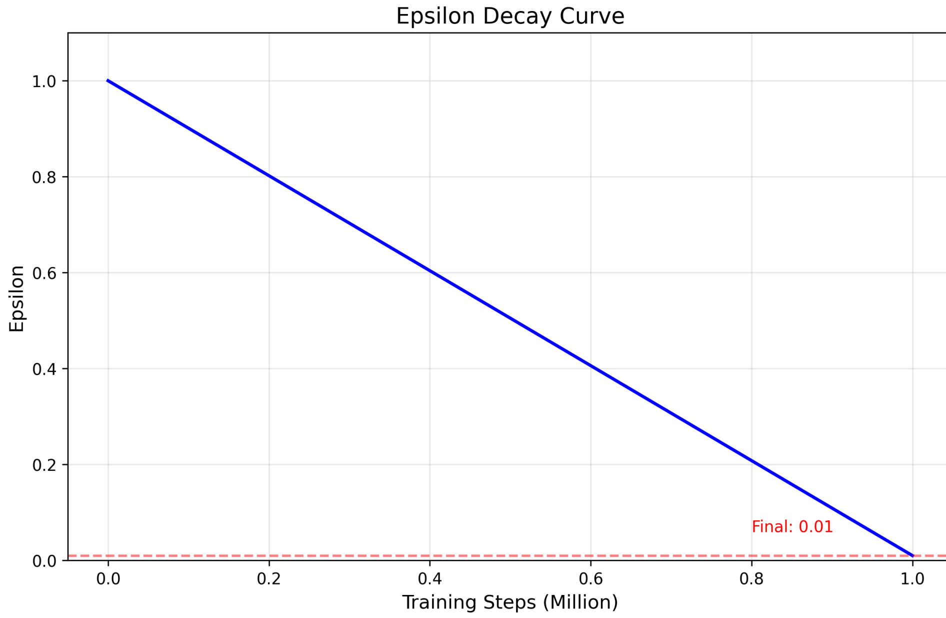

Figure 2: ϵ decay from 1.0 to 0.01 over 1M steps

3.2 Checkpoint Evaluation

After training, I evaluated 11 evenly spaced checkpoints plus the "best" model tracked during training. Each was tested over 10 episodes with $\epsilon = 0$ (greedy). Figure 3 plots the results with error bars. Table 3 gives the numbers.

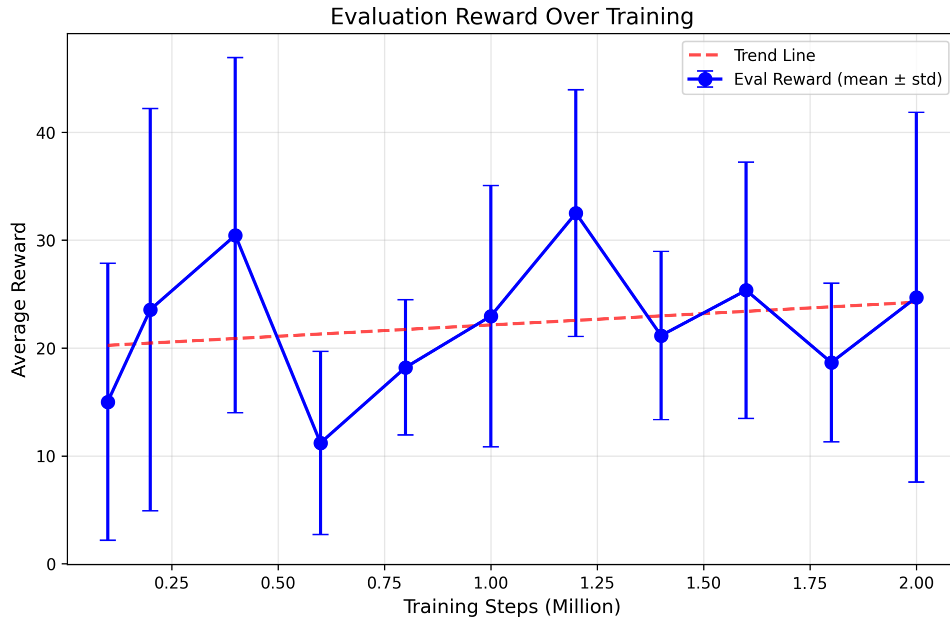

Figure 3: Evaluation scores across checkpoints, mean $\pm$ std over 10 episodes

Checkpoint	Mean Score	Std
100K	15.00	12.84
200K	23.55	18.66
400K	30.45	16.47
600K	11.20	8.49
800K	18.20	6.28
1.0M	22.95	12.10
1.2M	**32.50**	11.43
1.4M	21.15	7.80
1.6M	25.35	11.88
1.8M	18.65	7.35
2.0M (final)	24.70	17.15
best.pt (tracked live)	20.40	11.43

Table 3: Evaluation at each checkpoint, 10 episodes each. The live-tracked best model underperformed most saved checkpoints when re-evaluated.

A few things stand out:

1. **The 600K collapse**. Score dropped from 30.45 at 400K to 11.20 at 600K—below the 100K checkpoint (15.00). The network effectively unlearned whatever it had figured out, then climbed back to 32.50 by 1.2M. This isn't just noise; it's a real regression.

2. **No convergence**. Even in the second half of training, scores jump between 18 and 32. There's no point where things stabilize. The final checkpoint at 2M scores 24.70, which is worse than the checkpoint 800K steps earlier.

3. **best.pt isn't best**. The model the trainer tracked as best scored 20.40 when re-evaluated, worse than 7 out of 11 checkpoints. The in-training evaluation happened to catch a lucky or unlucky seed; with only 10 episodes and std of 12–17, a single evaluation can easily be 10+ points off.

4. **High variance throughout**. Standard deviations range from 6 to 19, meaning even the best checkpoint sometimes scores single digits and sometimes above 40.

3.3 Baseline Comparison

Table 4 puts the numbers in context. Random play gets ∼5. My best checkpoint gets 32.50. Published DQN scores on Space Invaders are in the 170–580 range, but those used 50M frames (12.5M steps), a larger replay buffer, and a single-frame input with different preprocessing. The comparison isn't apples-to-apples; 2M steps is roughly 16% of the original training budget. Whether the agent would reach competitive scores with more steps is an open question—the instability at 2M doesn't inspire confidence, but it also doesn't rule it out.

Method	Score	Compute
Random	∼5	—
This work (best)	32.50	6h, RTX 4060
This work (final, 2.0M)	24.70	6h, RTX 4060
Human (Bellemare et al., 2013)	∼165	—
DQN (Mnih et al., 2015)	170–580	7–10 days, single GPU

Table 4: Score comparison. Note the large difference in training budget.

4 Discussion

4.1 Why Things Worked (When They Did)

The Dueling architecture probably helped the most. In Space Invaders, the difference between good and bad states is much larger than the difference between good and bad actions in a given state. Dueling separates these—$V(s)$ captures whether the agent is safe, $A(s,a)$ captures whether shooting or dodging is better right now. Double DQN likely reduced the worst over-estimation, and PER probably helped early on by focusing on transitions where the agent was surprised.

4.2 Why Things Didn't Converge

The 600K collapse and ongoing oscillation suggest a few interacting problems:

Too much exploration for too long. ϵ decays linearly from 1.0 to 0.01 over 1M steps. At 600K, $\epsilon \approx 0.4$, so 40% of actions are random. That's enough randomness to frequently land in bad states the network hasn't learned to recover from. The data from those bad trajectories then contaminates the replay buffer and training.

PER makes the data distribution non-stationary. As the network improves, which transitions get high priority changes. The optimizer is essentially chasing a moving target. With only 2M environment steps, it may never settle.

3.3M parameters on 2M steps is under-training. The network has enough capacity to overfit to recent batches, and gradient variance from mini-batches may be high enough to destabilize the Q-function.

The best.pt issue. With std of 12–17 across 10 evaluation episodes, a single eval can easily be off by 10+ points. The live tracker keeps whichever model happened to score highest on one evaluation—but that evaluation might have been lucky. Checkpoint-based evaluation (saving every 50K steps) would have caught the actual best model, but the live-saved best.pt is misleading.

4.3 What Would Help

Based on the above, the most promising fixes are:

- Train longer. 2M steps clearly isn't enough; 10M would give the network more updates per parameter.

- Switch to a Noisy Network for exploration instead of ϵ-greedy. This replaces random action selection with learned parametric noise in the network weights, which provides state-dependent exploration and often converges faster.
- Use N-step returns. Instead of bootstrapping after one step, accumulate actual rewards over n steps. This propagates reward information faster and reduces variance from the value function.
- Add an SB3 DQN baseline trained identically. This would isolate whether the instability comes from the algorithm or my implementation.

5 Conclusion

I built a Dueling Double DQN with PER for Space Invaders from scratch. After 2M steps of training, it learned to score 32.50—better than random, but far below published baselines and still bouncing around wildly. The Dueling architecture, Double Q-learning, and PER each contributed to the learning that did happen, but 2M steps wasn't enough for convergence, and the combination of ϵ-greedy exploration with PER's non-stationary sampling distribution made things unstable. The codebase now supports parallel training, AMP mixed precision, and torch.compile, so scaling to 10M+ steps would be straightforward. Getting SB3 baselines and trying Noisy Nets or N-step returns are the logical next steps.

References

1. Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529–533.
2. Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep Reinforcement Learning with Double Q-learning. *AAAI*.
3. Wang, Z., Schaul, T., Hessel, M., et al. (2016). Dueling Network Architectures for Deep Reinforcement Learning. *ICML*.
4. Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized Experience Replay. *ICLR*.
5. Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *JAIR*, 47, 253–279.
6. Hessel, M., Modayil, J., Van Hasselt, H., et al. (2018). Rainbow: Combining Improvements in Deep Reinforcement Learning. *AAAI*.