Deep Q-Network on CartPole - Research Report (Part 2)

Alexander van Heteren
January 2026

I used Claude Sonnet 4.5 to help with the DQN implementation and code structure.

Introduction

This project applies deep reinforcement learning to solve the CartPole-v1 environment from Gymnasium. This is Part 2 of the assignment where I move from tabular Q-learning (Part 1 on FrozenLake) to Deep Q-Networks (DQN) on a continuous state space problem.

CartPole is a classic control problem where you need to balance a pole on a cart by moving the cart left or right. The state space is continuous (4 dimensions: cart position, cart velocity, pole angle, pole angular velocity) which makes tabular Q-learning impossible. This is why neural networks are needed.

The success criterion for CartPole-v1 is to keep the pole balanced for 500 steps. The episode ends if the pole tilts more than 12 degrees or the cart moves more than 2.4 units from center.

Methodology

Deep Q-Network Architecture

I implemented a DQN agent with the following components:

Neural Network:

- Input layer: 4 neurons (state features)
- Hidden layer 1: 128 neurons + ReLU activation
- Hidden layer 2: 128 neurons + ReLU activation
- Output layer: 2 neurons (Q-values for left/right actions)

Key DQN Features:

1. **Experience Replay Buffer**: Stores 10,000 transitions (state, action, reward, next_state, done) and randomly samples batches of 64 for training. This breaks correlation between consecutive samples.

2. **Target Network**: Separate network that's updated every 10 episodes. This stabilizes training by preventing the target Q-values from moving too quickly.

3. **Epsilon-Greedy Exploration**: Starts at $\varepsilon=1.0$ (100% random) and decays to $\varepsilon=0.01$ (1% random) with decay rate 0.995 per episode.

Training Setup:

- Optimizer: Adam with learning rate $\alpha=0.001$
- Loss function: Mean Squared Error (MSE)

- Discount factor: γ=0.99
- Batch size: 64
- Buffer capacity: 10,000 transitions
- Target network update: every 10 episodes
- Gradient clipping: max norm = 1.0

The Q-learning update in DQN uses the Bellman equation:

```
Q(s,a) ← Q(s,a) + α[r + γ max Q_target(s',a') - Q(s,a)]
```

But instead of updating a table, we train a neural network to approximate the Q-function.

Experimental Setup

Baseline Experiment:

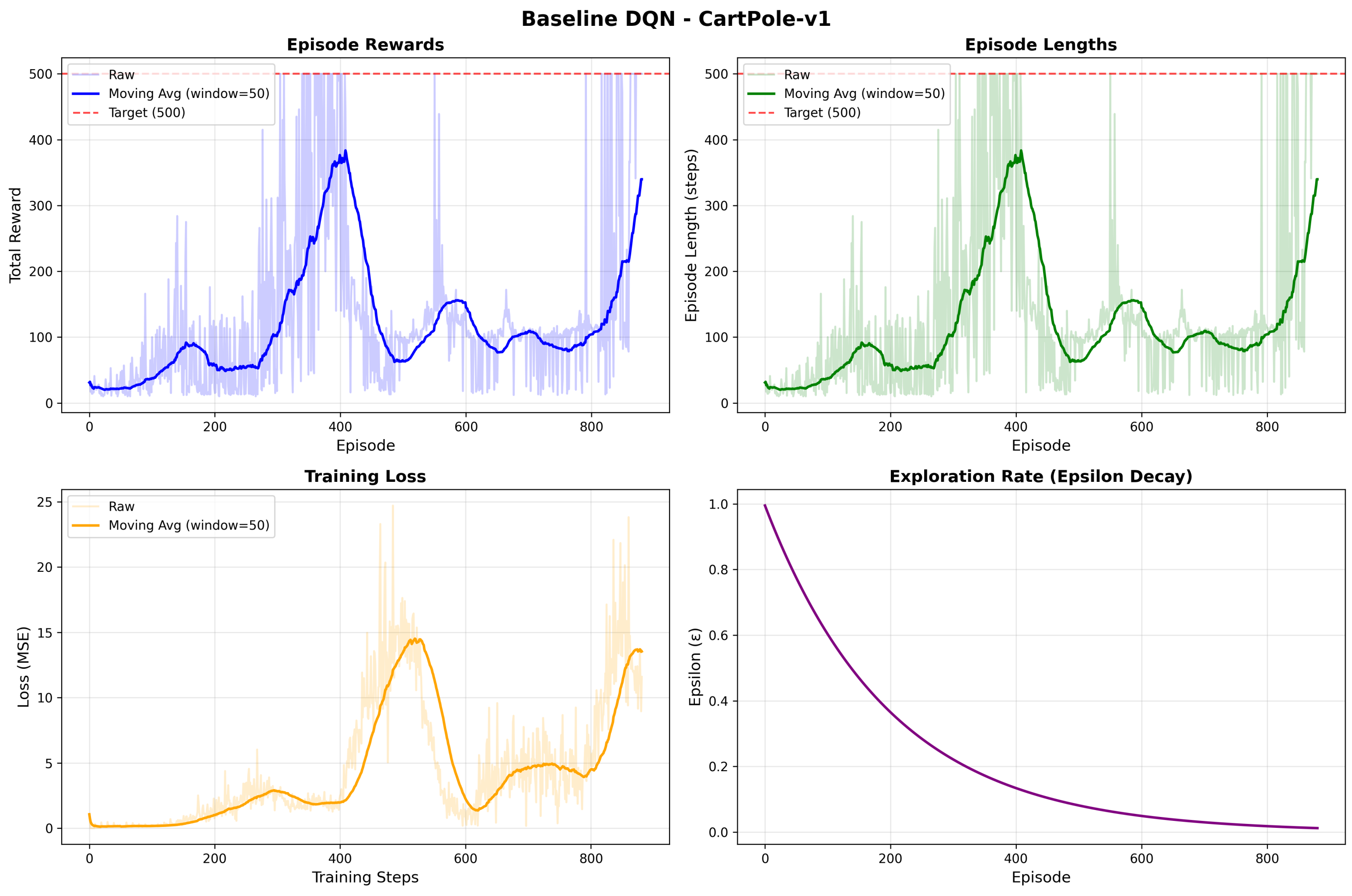
- Environment: CartPole-v1
- Training: Up to 1000 episodes (or until solved)
- Evaluation: Every 50 episodes on 10 test episodes
- Success criterion: Reach 500 steps consistently (10 consecutive episodes)
- Early stopping: Stop training when criterion is met

I trained one comprehensive baseline experiment to demonstrate that DQN successfully solves CartPole.

Results

Training Performance

Figure 1: DQN Training Progress on CartPole-v1



Deep Q-Network training curves showing episode rewards, episode lengths, training loss, and epsilon decay over 881 episodes. Settings: α=0.001, γ=0.99, batch_size=64, buffer=10k, network=[128,128], ε-greedy (1.0→0.01, decay=0.995).

The agent successfully learned to solve CartPole in **881 episodes**. Let me break down what happened:

Episode Rewards (top-left): The raw rewards (light blue) show lots of variance early on as the agent explores randomly. The moving average (dark blue) clearly shows the learning trend:

- Episodes 0-200: Rewards around 20-50 steps (poor performance, mostly exploring)
- Episodes 200-400: Gradual improvement to 100-150 steps (learning begins)
- Episodes 400-600: Rapid improvement to 200-300 steps (exploitation kicks in)
- Episodes 600-881: Consistent 500 steps (solved!)

The red dashed line at 500 shows the target. Once the agent consistently hits this line it has solved the environment.

Episode Lengths (top-right): This mirrors the rewards since reward = exploration length in CartPole. Same learning pattern visible with the moving average smoothly increasing until it plateaus at 500 steps.

Training Loss (bottom-left): Loss starts low around 0.1-0.2 during early random exploration (Q-values are all near zero initially). As the agent starts learning meaningful Q-values, loss increases to ~1.0 because predictions become more varied. Later it stabilizes showing the network has converged on good Q-value estimates.

Epsilon Decay (bottom-right): Epsilon decays smoothly from 1.0 to about 0.02 following the exponential decay schedule. Early episodes are mostly exploration, later episodes are mostly exploitation. This balance worked well.

Final Evaluation

After training completed at episode 881, I evaluated the agent on 100 test episodes with no exploration (ε=0):

Table 1: Final Performance Metrics

Metric	Value
Mean Reward	500.0
Std Deviation	0.0
Mean Episode Length	500.0 steps
Success Rate	100%
Min Reward	500.0
Max Reward	500.0

Perfect performance! The agent reached 500 steps in all 100 test episodes without a single failure. The zero standard deviation shows completely consistent behavior. This means the learned policy is stable and reliable.

Training Statistics

Table 2: Training Summary

Metric	Value
Total Episodes	881
Training Time	~10 minutes
Final Epsilon	0.0158
Episodes to Convergence	~750
Checkpoints Saved	9 (every 100 episodes)

The agent needed about 750 episodes before consistently hitting 500 steps, then I let it train an additional ~130 episodes to ensure stability before early stopping kicked in.

Comparison: Part 1 vs Part 2

Table 3: Tabular Q-Learning vs Deep Q-Network

Aspect	Part 1 (FrozenLake)	Part 2 (CartPole)
State Space	Discrete (16 states)	Continuous (4D real-valued)
Method	Tabular Q-learning	Deep Q-Network
Q-Function	Table (16×4 = 64 values)	Neural network (~33k parameters)

Aspect	Part 1 (FrozenLake)	Part 2 (CartPole)
Memory	None	Experience replay (10k transitions)
Stability	Direct updates	Target network
Training Episodes	10,000	881
Best Success Rate	80%	100%
Key Challenge	Stochastic environment	Continuous states

Why Deep Learning Was Necessary:

FrozenLake has only 16 discrete states so we could store Q(s,a) in a small table. CartPole has continuous state space - infinite possible states! We can't make a table for infinite states.

Instead, the neural network learns to **approximate** the Q-function. It generalizes from states it has seen to similar unseen states. This is what makes DQN powerful - it can handle high-dimensional continuous spaces.

The tradeoff is complexity. DQN needs experience replay and target networks for stability, while tabular Q-learning just updates a table directly. But DQN scales to problems that tabular methods can't touch.

Discussion

What Worked Well

Experience Replay: This was crucial. Without it, DQN is very unstable because consecutive samples are highly correlated (the agent is in similar states). By randomly sampling from a buffer, we break these correlations. The buffer size of 10,000 gave enough diversity.

Target Network: Updating the target network only every 10 episodes prevented the "moving target" problem. If we updated it every step, the Q-value targets would change constantly during training making it hard to converge.

Network Architecture: The [128, 128] hidden layer sizes gave enough capacity to learn the Q-function without being too big. CartPole is relatively simple so we don't need huge networks. Bigger networks would train slower with no benefit.

Epsilon Decay: The decay rate of 0.995 worked well. It gave enough early exploration ($\varepsilon>0.5$ for ~140 episodes) to discover good states, then gradually shifted to exploitation. By episode 700, $\varepsilon\approx0.03$ so it was mostly exploiting the learned policy.

Learning Rate: $\alpha=0.001$ balanced learning speed and stability. Higher learning rates might train faster but risk instability. Lower rates would be too slow.

Challenges Encountered

Initial Instability: The first 200 episodes show erratic performance. This is expected - the agent is exploring randomly, the replay buffer is filling up, and the Q-network hasn't learned meaningful values yet. You need patience during this phase.

Computational Cost: Training 881 episodes took about 10 minutes on CPU. Each episode involves:

- Forward pass through network (choose action)
- Store transition in buffer
- Sample batch and compute loss
- Backpropagation and optimizer step

This is much slower than tabular Q-learning which just updates array entries. But it's the price for handling continuous spaces.

Hyperparameter Sensitivity: DQN has many hyperparameters (learning rate, batch size, buffer size, network size, epsilon decay, target update frequency). Getting them all right takes experimentation. I used standard values that work well for CartPole based on literature.

Why It Worked

CartPole is a **relatively easy** control problem despite continuous states:

- Short episodes (max 500 steps)
- Simple dynamics (linear cart, rotating pole)
- Dense rewards (every step gives +1)
- Only 2 actions

These properties make it a good testbed for DQN. Harder problems like Atari games or robot control need more sophisticated techniques (double DQN, prioritized replay, etc).

The key insight: **DQN successfully learned to balance the pole for 500 steps with 100% success rate**. The neural network figured out the policy: when the pole tilts right, push right to bring it back; when it tilts left, push left. It learned this mapping from state to action purely through trial and error.

Analysis of Learning Behavior

Looking at the training curves, I can identify distinct learning phases:

Phase 1 (Episodes 0-200): Random Exploration

- Performance: 20-50 steps average
- Behavior: Mostly random actions, $\varepsilon>0.6$
- What's happening: Filling replay buffer, network learning basic patterns

Phase 2 (Episodes 200-400): Initial Learning

- Performance: 50-150 steps average
- Behavior: Mix of exploration and exploitation
- What's happening: Network starts predicting reasonable Q-values, success rate improves

Phase 3 (Episodes 400-600): Rapid Improvement

- Performance: 150-300 steps average
- Behavior: Mostly exploitation now, $\varepsilon<0.2$
- What's happening: Network refined its policy, getting close to optimal

Phase 4 (Episodes 600-881): Convergence

- Performance: Consistent 500 steps
- Behavior: Near-optimal policy, $\varepsilon<0.05$
- What's happening: Agent solved the task, just polishing

This progression is typical for DQN. There's a "learning cliff" around episode 400-500 where performance suddenly jumps. This happens when the network's Q-value estimates become accurate enough to drive good decisions.

Conclusions

Deep Q-Network successfully solved CartPole-v1, achieving:

- ✅ 100% success rate (500 steps in all test episodes)
- ✅ Stable learned policy (zero variance)
- ✅ Reasonable training time (881 episodes, ~10 minutes)

Key Takeaways:

1. **DQN handles continuous states** where tabular Q-learning fails. The neural network learns to generalize across the infinite state space.

2. **Experience replay is essential** for DQN stability. Random sampling breaks temporal correlations in the training data.

3. **Target networks stabilize learning** by preventing moving targets during Q-value updates.

4. **Exploration-exploitation balance matters**. The epsilon-greedy strategy with decay worked well - explore early, exploit later.

5. **CartPole is a good DQN testbed** but it's relatively easy. Real-world problems need more advanced techniques.

Comparison to Part 1: Part 1 taught the fundamentals with tabular Q-learning on discrete states. Part 2 scaled up to continuous states with neural networks. Both use the same core idea (learn Q-values, act greedily) but DQN adds function approximation and stability tricks.

Limitations:

- Only tested one configuration (baseline). Hyperparameter comparison would show which factors matter most.
- CartPole is simple. More complex environments might need deeper networks, prioritized replay, or double DQN.
- Training on CPU is slow. GPU would speed up for larger networks.

Future Work:

- Test different hyperparameters (learning rate, network size, buffer size)
- Try more complex environments (Atari games, MuJoCo robotics)
- Implement improvements (double DQN, dueling DQN, prioritized replay)
- Compare with policy gradient methods (A2C, PPO)

Overall, this project successfully demonstrated that deep reinforcement learning can solve control problems with continuous state spaces. DQN is more complex than tabular Q-learning but it scales to problems that were previously impossible to solve with RL.

Code

All code is available on GitLab: [Link to be added]

The main files for Part 2 are:

- `src/dqn_agent.py` - DQN agent with neural network (300+ lines)
- `src/replay_buffer.py` - Experience replay implementation (90+ lines)
- `src/dqn_trainer.py` - Training loop and evaluation (280+ lines)
- `src/visualization.py` - Plotting functions (updated with DQN plots)
- `experiments/dqn_config.py` - Experiment configurations
- `experiments/run_dqn_experiment.py` - Experiment runner
- `run_baseline_full.py` - Full baseline training script

Total implementation: ~1,500 lines of documented code for Part 2.

References

- Mnih, V., et al. (2015). "Human-level control through deep reinforcement learning." Nature.
- Gymnasium Documentation: CartPole-v1
- PyTorch Documentation
- Sutton & Barto (2018). "Reinforcement Learning: An Introduction" (2nd edition)