Solving CartPole with Deep Q-Networks: A Reinforcement Learning Approach

Reinforcement Learning Project

March 10, 2025

Abstract

This report presents a comprehensive study on solving the classic CartPole control problem using Deep Q-Networks (DQN), a reinforcement learning approach. We implement a DQN agent capable of learning to balance a pole on a moving cart by making sequential action decisions. The project explores various improvements to the basic DQN algorithm, including Double DQN implementation, reward shaping, and learning rate management. Our final model achieves a high average score approaching the target threshold of 475 steps out of the maximum 500. We analyze the learning process, challenges, and strategies that led to our improved performance.

Contents

1 Introduction

Reinforcement learning (RL) is a paradigm where an agent learns to make decisions by taking actions in an environment to maximize cumulative rewards. Unlike supervised learning, RL does not require labeled data but instead learns through trial-and-error interactions with the environment. This approach has shown remarkable success in solving complex control problems, from games to robotic tasks.

In this project, we address the CartPole balancing problem—a classic control task in RL literature—using Deep Q-Networks (DQN), a powerful value-based RL algorithm that combines Q-learning with deep neural networks. The CartPole problem serves as an excellent benchmark for testing RL algorithms due to its simple mechanics but challenging control requirements.

Our study aims to:

- Implement a DQN agent for the CartPole environment
- Explore enhancements to the basic DQN algorithm
- Analyze the impact of various hyperparameters on performance
- Overcome challenges like catastrophic forgetting and learning plateaus

The report details our methodology, implementation, experiments, and results, providing insights into both the theoretical foundations and practical considerations of applying DQN to solve control problems.

2 Problem Description

2.1 The CartPole Environment

The CartPole problem, implemented in OpenAI's Gymnasium (formerly Gym) framework, consists of a cart that can move horizontally along a frictionless track, with a pole attached to it by an unactuated joint. The system starts with the pole in an upright position, and the goal is to prevent it from falling over by applying forces to the cart.

2.1.1 State Space

The state is represented by a 4-dimensional vector:

- Cart position (x): Position of the cart on the track
- Cart velocity ($\dot{x}$): Velocity of the cart
- Pole angle (θ): Angle of the pole with respect to the vertical
- Pole angular velocity ($\dot{\theta}$): Rate of change of the pole angle

2.1.2 Action Space

The agent can take two discrete actions:

- Push the cart to the left (action = 0)
- Push the cart to the right (action = 1)

2.1.3 Reward Structure

The environment provides a reward of +1 for every timestep the pole remains upright. The episode ends (terminal state) when:

- The pole angle exceeds ±15 degrees from vertical
- The cart position is more than ±2.4 units from the center
- The episode length reaches 500 timesteps

2.1.4 Success Criteria

For CartPole-v1, the environment is considered solved when the agent achieves an average score of 475 over 100 consecutive episodes, where the maximum possible score per episode is 500.

3 Methods

3.1 Deep Q-Networks (DQN)

Q-learning is a model-free reinforcement learning algorithm that learns the value of an action in a given state. The goal is to learn a policy that maximizes the expected cumulative reward. In environments with large or continuous state spaces, traditional Q-learning becomes intractable. DQN addresses this by approximating the Q-function using a neural network.

3.1.1 Q-Learning Background

In Q-learning, we maintain a table of Q-values for each state-action pair. The optimal Q-function satisfies the Bellman equation:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a')|s,a\right] \tag{1}$$

where:

- $Q^*(s,a)$ is the optimal Q-value for taking action a in state s
- r is the immediate reward

- γ is the discount factor
- s' is the next state
- a' is the next action

3.1.2 Deep Q-Network Architecture

In DQN, we use a neural network to approximate the Q-function. The network takes the state as input and outputs Q-values for all possible actions. Our architecture consists of:

- Input layer: 4 nodes (state dimension)
- Hidden layer 1: 128 neurons with ReLU activation
- Hidden layer 2: 128 neurons with ReLU activation
- Output layer: 2 nodes (one for each action)

3.1.3 Key DQN Components

Our implementation includes several critical components:

Experience Replay We store agent experiences (state, action, reward, next state, done) in a replay buffer and sample random batches during training. This breaks correlations between consecutive samples and improves learning stability.

Double DQN We implement Double DQN to address the overestimation bias in standard DQN. The action selection and evaluation are decoupled:

- Policy network selects the next action
- Target network evaluates the Q-value of that action

Target Network We maintain a separate target network that is updated less frequently than the policy network, providing stable Q-value targets for learning.

Epsilon-Greedy Exploration We use an epsilon-greedy policy that balances exploration and exploitation:

- With probability ϵ, choose a random action
- With probability $1 - \epsilon$, choose the action with the highest Q-value

4 Implementation Details

4.1 Software Framework

The project was implemented using:

- Python 3.11
- PyTorch 2.0.1 for neural network implementation and training
- Gymnasium 0.28.1 for the CartPole environment
- NumPy 1.24.3 for numerical operations
- Matplotlib 3.7.1 for visualization

4.2 DQN Agent Implementation

Our agent implementation includes the following key components:

4.2.1 Replay Buffer

We implemented a simple replay buffer using a fixed-size deque:

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.memory = deque(maxlen=capacity)

    def push(self, *args):
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Listing 1: Replay Buffer Implementation

4.2.2 Neural Network Architecture

We implemented the Q-network using PyTorch:

```python
class DQN(nn.Module):
    def __init__(self, state_size, action_size, hidden_size
    =128):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, action_size)

        # Initialize weights with Xavier/Glorot initialization
        nn.init.xavier_uniform_(self.fc1.weight)
        nn.init.xavier_uniform_(self.fc2.weight)
        nn.init.xavier_uniform_(self.fc3.weight)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```
Listing 2: DQN Network Architecture

4.2.3 Double DQN Implementation

We implemented Double DQN in the learning process:

```python
# Double DQN implementation
# Get argmax actions from policy network
with torch.no_grad():
    # Get the actions that would be selected by the policy
   network
    policy_next_actions = self.policy_net(next_states).max(1)
   [1].unsqueeze(1)

    # Get the Q-values for these actions from the target
   network
    Q_targets_next = self.target_net(next_states).gather(1,
   policy_next_actions)

# Compute Q targets for current states
Q_targets = rewards + (self.gamma * Q_targets_next * (1 - dones
   ))
```
Listing 3: Double DQN Implementation

4.2.4 Learning Rate Management

We implemented learning rate decay and reset functionality to address plateaus in learning:

```python
def reset_learning_rate(self, new_lr=None):
    """Reset the learning rate to boost training."""
    if new_lr is None:
        new_lr = self.initial_lr

    # Update learning rate parameter in optimizer
    for param_group in self.optimizer.param_groups:
        param_group['lr'] = new_lr

    # Create a new scheduler
    self.scheduler = optim.lr_scheduler.ExponentialLR(
        self.optimizer, gamma=self.lr_decay)

    return new_lr
```
Listing 4: Learning Rate Management

4.3 Training Approach

We implemented several specialized training approaches:

4.3.1 Reward Shaping

We enhanced the reward structure to guide learning:

```python
# Penalize early termination with progressive penalty
if done and t < max_t - 1:
    # Scale penalty based on how early the failure occurs
    time_factor = 1.0 - (t / max_t)
    modified_reward = -1.0 - (early_factor * progress_factor)

# Bonus for maintaining balance longer
elif t > 200 and not done:
    # Progressive reward increase for longer episodes
    bonus = 1.0 + (t / 500) * 0.2
    modified_reward = reward * bonus
```

Listing 5: Reward Shaping

4.3.2 Curriculum Learning

We implemented a simple form of curriculum learning:

```python
# Create curriculum difficulty - start with easier cases if
    struggling
if last_10_avg < 400 and len(scores_window) >= 10:
    # Easier starting condition (more stable)
    env.unwrapped.state = np.array([0, 0, 0.01, 0], dtype=np.
    float32)
    state = env.unwrapped.state
```

Listing 6: Curriculum Learning

5 Experimentation

5.1 Hyperparameter Tuning

We experimented with various hyperparameters:

Parameter	**Initial Value**	**Final Value**
Hidden layer size	64	128
Learning rate	0.001	0.0005
Replay buffer size	10,000	100,000
Batch size	64	128
Epsilon decay	0.995	0.997
Target update rate (τ)	0.001	0.002
Learning rate decay	0.9999	0.9995

Table 1: Hyperparameter evolution during experimentation

5.2 Network Architecture Experiments

We experimented with different network architectures:

- Single hidden layer with 64 neurons
- Two hidden layers with 64 neurons each
- Two hidden layers with 128 neurons each (final)
- Three hidden layers (128, 128, 64) - temporarily used

The final architecture with two hidden layers of 128 neurons each provided the best balance of capacity and training efficiency.

5.3 Training Phases

Our training process evolved through several phases:

5.3.1 Phase 1: Basic DQN

Initial implementation with standard DQN achieved mediocre performance, with scores plateauing around 100.

5.3.2 Phase 2: Enhanced DQN

Added Double DQN, increased network size, and improved reward shaping, reaching scores around 200.

5.3.3 Phase 3: Advanced Tuning

Fine-tuned hyperparameters and implemented better learning rate management, reaching scores of 300-350.

5.3.4 Phase 4: Specialized Training

Created specialized training process with curriculum learning and adaptive reward shaping, pushing scores to 425+.

6 Results

6.1 Learning Curves

Our training showed clear progress through different phases. The learning curves demonstrate:

- Initial slow learning (0-300 episodes)
- Rapid improvement phase (300-600 episodes)
- Performance plateau (600-1000 episodes)
- Specialized training breakthrough (beyond 1000 episodes)

6.2 Performance Analysis

6.2.1 Impact of Double DQN

Double DQN significantly reduced overestimation bias and improved stability, leading to more consistent learning and higher scores.

6.2.2 Effect of Reward Shaping

Enhancing the reward structure to penalize early failures and reward long balancing periods effectively guided the agent toward better policies.

6.2.3 Learning Rate Management

One critical finding was that learning rate decay could lead to premature convergence. Resetting the learning rate when plateaus occurred allowed for continued improvement.

6.3 Final Performance

Our best model achieved an average score of 428.68 over 100 consecutive episodes, approaching but not quite reaching the target score of 475. However, many individual episodes reached the maximum score of 500, demonstrating the agent's capability to solve the task completely under favorable conditions.

6.4 Challenges Encountered

6.4.1 Catastrophic Forgetting

We observed that the agent would sometimes lose previously learned skills, with performance degrading after initial improvements. Our solution included:

- Larger replay buffer (100,000 experiences)
- More stable target network updates
- Better learning rate scheduling

6.4.2 Learning Plateaus

The agent frequently reached performance plateaus. We addressed this with:

- Learning rate resets
- Network fine-tuning with slight noise addition
- Curriculum learning techniques

7 Discussion

7.1 Analysis of Approach

Our approach to solving the CartPole problem demonstrated both the strengths and limitations of DQN:

Strengths

- Rapid initial learning with proper reward shaping
- Ability to learn effective control policies from raw state data
- Adaptability to hyperparameter adjustments

Limitations

- Sensitivity to hyperparameters
- Tendency to converge to suboptimal policies
- Challenge of maintaining consistent performance over extended training

7.2 Improvement Opportunities

Several opportunities for further improvement exist:

Prioritized Experience Replay Although we initially implemented prioritized experience replay, we removed it for speed. A more efficient implementation could provide benefits without the computational overhead.

Dueling Networks Implementing a dueling network architecture could improve performance by separating state value and action advantage estimation.

Noisy Networks Replacing epsilon-greedy exploration with noisy networks might provide more structured exploration.

8 Conclusion

In this project, we successfully implemented a Deep Q-Network agent for the CartPole problem, achieving scores approaching the environment's solving criteria of 475. Our work demonstrated the effectiveness of various DQN enhancements, including Double DQN, adaptive reward shaping, and learning rate management.

The results highlight both the power and challenges of deep reinforcement learning for control problems. While we were able to achieve high performance, the sensitivity to hyperparameters and the need for careful management of the learning process underscore the complexity of applying these methods in practice.

Key takeaways from this project include:

- The importance of proper reward shaping in guiding the learning process
- The critical role of learning rate management in preventing plateaus
- The value of curriculum learning in tackling challenging control tasks
- The effectiveness of Double DQN in reducing value overestimation

Future work could explore more advanced techniques such as prioritized experience replay, dueling architectures, and distributional RL to further improve performance and stability.

9 References

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). *Human-level control through deep reinforcement learning.* Nature, 518(7540), 529-533.

[2] Van Hasselt, H., Guez, A., & Silver, D. (2016). *Deep reinforcement learning with double q-learning.* In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 30, No. 1).

[3] Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). *Prioritized experience replay.* arXiv preprint arXiv:1511.05952.

[4] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016). *Dueling network architectures for deep reinforcement learning.* In International conference on machine learning (pp. 1995-2003). PMLR.

[5] *Gymnasium Documentation.* Available at: https://gymnasium.farama.org/

[6] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). *PyTorch: An imperative style, high-performance deep learning library.* Advances in neural information processing systems, 32, 8026-8037.