Reinforcement Learning

Lecture 6: Deep Q-Networks (DQN)

Taehoon Kim

Sogang University MIMIC Lab
https://mimic-lab.com

Fall Semester 2025

Session Goals

By the end of this lecture, you will be able to:

1. Explain why tabular Q-learning fails with high-dimensional state spaces
2. Understand how DQN addresses divergence risks with experience replay and target networks
3. Derive the DQN learning target and implement the Huber loss objective
4. Implement a production-grade DQN agent in PyTorch with:
    - Replay buffer, target network, and ϵ-greedy exploration
    - Mixed precision (AMP) and torch.compile() acceleration
    - Checkpoint save/restore and TensorBoard logging
5. Run ablations on hyperparameters and interpret results

Prerequisites: Q-learning (Lecture 5), PyTorch basics (Lecture 2)

Foundations Overview

Focus to kick things off

- Diagnose why tabular Q-learning breaks in large/continuous spaces
- Build up the DQN recipe: replay buffer, target network, loss
- Reason about stability tricks (Double DQN, Huber loss, AMP)

Flow

1. Revisiting tabular Q-learning pitfalls
2. Introducing neural function approximation
3. Deriving the DQN update and variants

Goal: Bridge tabular intuition to deep RL foundations before coding.

The Curse of Dimensionality

Tabular Q-Learning Limitations:

- State space explosion: CartPole with 10 bins/dimension $\rightarrow 10^4 = 10,000$ states
- Memory requirements: $O(|S| \times |A|)$
- No generalization between similar states
- Must visit every state-action pair

Real-world Examples:

- Atari games: 210×160 pixels $\times$ 128 colors $\approx 10^{120,000}$ states
- Robotics: Continuous joint angles and velocities
- Autonomous driving: High-dimensional sensor inputs

Solution: Function Approximation with Neural Networks

Function Approximation in Q-Learning

From Table to Function:

Tabular Q-Learning:

- $Q : S \times A \to \mathbb{R}$
- Stored as table $Q[s, a]$
- Update: $Q[s, a] \leftarrow$ target
- Exact values per state

Deep Q-Learning:

- $Q_\theta : S \times A \to \mathbb{R}$
- Neural network with parameters θ
- Update: $\theta \leftarrow \theta - \alpha \nabla_\theta L$
- Approximate values

Benefits of Neural Networks:

- Automatic feature extraction
- Generalization to similar states
- Handles high-dimensional inputs (images, etc.)
- Compact representation

Why Naive Neural Q-Learning Fails

Deadly Triad of Instability:

1. **Function Approximation**: Non-tabular representation
2. **Bootstrapping**: Using estimates to update estimates
3. **Off-policy Learning**: Learning from old experiences

Specific Problems:

- **Moving Targets**: $Q_\theta(s', a')$ changes as we update θ
- **Correlation**: Sequential samples are highly correlated
- **Feedback Loops**: Updates affect future targets
- **Overestimation**: Max operator causes positive bias

Result: Divergence, oscillation, or poor performance

DQN Innovation 1: Experience Replay

Breaking Correlation with Memory:

- Store transitions $(s, a, r, s', done)$ in replay buffer $\mathcal{D}$
- Sample random mini-batches for training
- Breaks temporal correlation
- Improves sample efficiency (reuse experiences)

Replay Buffer Implementation:

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

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)
```

Experience Replay: Why It Works

Without Replay:

- Samples: $s_t, s_{t+1}, s_{t+2}, \ldots$
- High correlation
- Recent bias
- Catastrophic forgetting
- Unstable gradients

With Replay:

- Samples: $s_{17}, s_{203}, s_5, \ldots$
- I.I.D.-like sampling
- Balanced experience
- Better coverage
- Stable gradients

Key Parameters:

- Buffer size: Typically 10^4 to 10^6 transitions
- Batch size: Usually 32-256
- Start learning after: 1000+ transitions (warmup)

DQN Innovation 2: Target Network

Stabilizing the Learning Target:

- Maintain two networks: Online Q_θ and Target Q_{θ^-}
- Use target network for computing TD targets
- Update target network periodically (every C steps)

The Key Insight:

$$\text{Without target network:} \quad y = r + \gamma \max_{a'} Q_\theta(s', a') \tag{1}$$

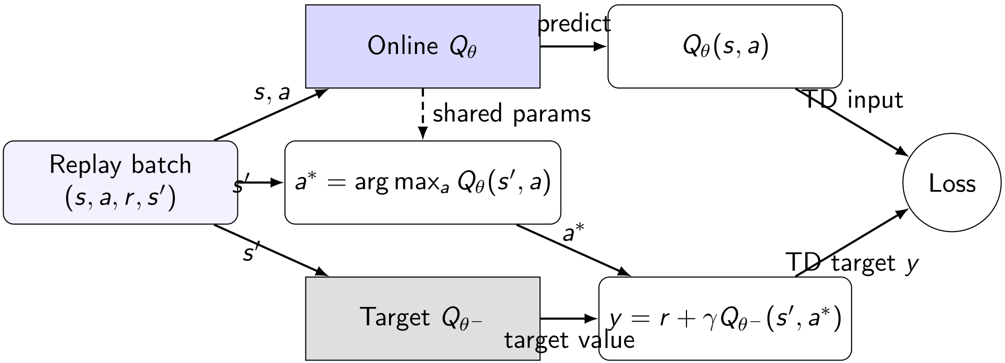
$$\text{With target network:} \quad y = r + \gamma \max_{a'} Q_{\theta^-}(s', a') \tag{2}$$

Target θ^- remains fixed during updates, breaking harmful feedback loops!

Update Strategies:

- Hard update: $\theta^- \leftarrow \theta$ every C steps
- Soft update: $\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$ each step

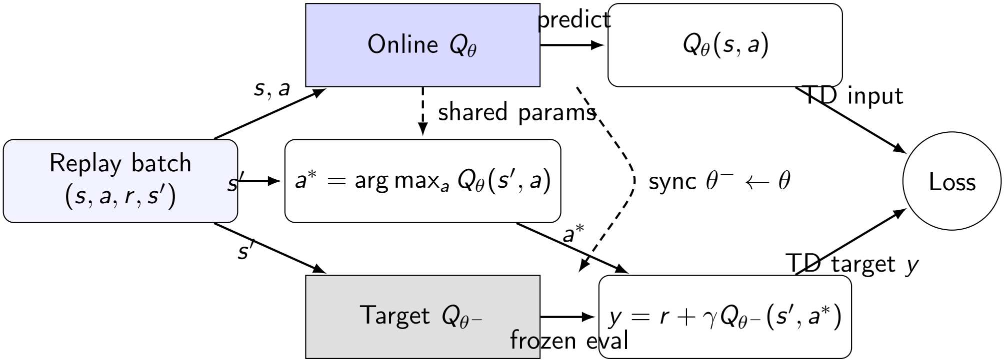
Target Network Flow Forward



Forward pass:

1. Sample a mini-batch from replay to evaluate $Q_\theta(s, a)$.
2. Use the online network to select a^* for each next state s'.
3. Combine a^* with the frozen target network to construct TD targets y.

Target Network Flow Updates



Updates and stability:

1. Backpropagate the loss through Q_θ only. Keep Q_{θ^-} frozen.
2. Periodically copy parameters ($\theta^- \leftarrow \theta$) to refresh the target.

DQN Loss Function

The DQN Objective:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(y - Q_\theta(s,a)\right)^2\right] \tag{3}$$

where the target is:

$$y = r + \gamma(1 - done) \cdot \max_{a'} Q_{\theta^-}(s', a') \tag{4}$$

Huber Loss (Smooth L1):

$$\mathcal{L}_\delta(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| \leq \delta \\ \delta(|x| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \tag{5}$$

Benefits: Less sensitive to outliers, prevents gradient explosion

The Complete DQN Algorithm

Initialize replay buffer $\mathcal{D}$ with capacity N
Initialize Q-network Q_θ with random weights
Initialize target network Q_{θ^-} with $\theta^- = \theta$
for episode $= 1$ to M **do**
  Initialize state s_1
  for $t = 1$ to T **do**
    Select action $a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q_\theta(s_t, a) & \text{otherwise} \end{cases}$
    Execute a_t, observe r_t, s_{t+1}
    Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample mini-batch from $\mathcal{D}$
    Compute targets $y_i = r_i + \gamma \max_{a'} Q_{\theta^-}(s_i', a')$
    Update θ by minimizing $L = \frac{1}{|B|} \sum_i (y_i - Q_\theta(s_i, a_i))^2$
    Every C steps: $\theta^- \leftarrow \theta$
  end for
end for

Double DQN: Reducing Overestimation

The Overestimation Problem:

- Max operator in Q-learning causes positive bias
- $\mathbb{E}[\max_a Q(s, a)] \geq \max_a \mathbb{E}[Q(s, a)]$ (Jensen's inequality)
- Errors accumulate through bootstrapping

Double DQN Solution:

- Decouple action selection from evaluation
- Use online network to select actions
- Use target network to evaluate them

$$\text{DQN:} \quad y = r + \gamma \max_{a'} Q_{\theta^-}(s', a') \tag{6}$$

$$\text{Double DQN:} \quad y = r + \gamma Q_{\theta^-}(s', \arg\max_{a'} Q_\theta(s', a')) \tag{7}$$

Exploration Strategy: ϵ-Greedy

Balancing Exploration and Exploitation:

```python
def select_action(state, epsilon):
    if random.random() < epsilon:
        return env.action_space.sample()  # Explore
    else:
        q_values = q_network(state)
        return q_values.argmax()          # Exploit
```

Epsilon Scheduling:

- Linear decay: $\epsilon_t = \epsilon_{start} - t \cdot \frac{\epsilon_{start} - \epsilon_{end}}{T}$
- Exponential decay: $\epsilon_t = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) \cdot e^{-t/\tau}$
- Step decay: Reduce by factor at milestones

Typical values: $\epsilon_{start} = 1.0$, $\epsilon_{end} = 0.01$

DQN Hyperparameters

Parameter	**Typical Value**	**Description**
Learning rate	10^{-4} to 10^{-3}	Gradient descent step size
Batch size	32-256	Mini-batch size
Buffer size	10^4 to 10^6	Replay buffer capacity
Target update	1000-10000 steps	Hard update frequency
γ	0.99	Discount factor
ϵ_{start}	1.0	Initial exploration
ϵ_{end}	0.01-0.1	Final exploration
Hidden layers	[128, 128]	Network architecture
Optimizer	Adam	Gradient optimizer
Loss	Huber	Loss function

Common Pitfalls and Solutions

1. **Insufficient Warmup**
   - Problem: Learning from small buffer
   - Solution: Start after 1000+ transitions
2. **Exploration Collapse**
   - Problem: ϵ decays too quickly
   - Solution: Longer decay schedule
3. **Gradient Explosion**
   - Problem: Unstable training
   - Solution: Gradient clipping, Huber loss
4. **Reward Scale**
   - Problem: Rewards too large/small
   - Solution: Reward clipping or normalization
5. **Target Update Frequency**
   - Problem: Too frequent or too rare
   - Solution: Tune based on environment

DQN Variants and Extensions

Major DQN Improvements:

- **Double DQN (2015)**: Reduce overestimation bias
- **Prioritized Replay (2015)**: Sample important transitions more
- **Dueling DQN (2016)**: Separate value and advantage streams
- **Rainbow (2017)**: Combine all improvements
- **C51 (2017)**: Distributional Q-learning
- **Noisy Networks (2017)**: Parameter noise for exploration

Rainbow Components:

1. Double Q-learning
2. Prioritized replay
3. Dueling networks
4. Multi-step learning
5. Distributional RL
6. Noisy networks

When to Use DQN

Good fit for DQN:

- Discrete action spaces
- High-dimensional observations (images)
- Need sample efficiency
- Off-policy learning beneficial
- Deterministic environments

Consider alternatives when:

- Continuous actions $\rightarrow$ DDPG, TD3, SAC
- On-policy preferred $\rightarrow$ PPO, A2C
- Simple state space $\rightarrow$ Tabular Q-learning
- Safety critical $\rightarrow$ Conservative algorithms

Success stories: Atari games, resource allocation, trading

Key Equations for DQN

Bellman Optimality with Function Approximation:

$$Q^*(s, a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right] \tag{8}$$

Semi-gradient Update:

$$\theta_{t+1} = \theta_t + \alpha\,[y_t - Q_\theta(s_t, a_t)]\,\nabla_\theta Q_\theta(s_t, a_t) \tag{9}$$

Convergence Conditions (Tabular):

- All state-action pairs visited infinitely
- Learning rate satisfies Robbins-Monro conditions
- $\sum_t \alpha_t = \infty$, $\sum_t \alpha_t^2 < \infty$

Note: Neural Q-learning convergence not guaranteed!

Efficient Replay Buffer

```python
class ReplayBuffer:
    def __init__(self, capacity, obs_dim):
        # Pre-allocate arrays for efficiency
        self.observations = np.zeros((capacity, obs_dim),
                                     dtype=np.float32)
        self.actions = np.zeros(capacity, dtype=np.int64)
        self.rewards = np.zeros(capacity, dtype=np.float32)
        self.next_observations = np.zeros((capacity, obs_dim),
                                          dtype=np.float32)
        self.dones = np.zeros(capacity, dtype=np.float32)

        self.position = 0
        self.size = 0
        self.capacity = capacity

    def push(self, obs, action, reward, next_obs, done):
        idx = self.position
        self.observations[idx] = obs
        self.actions[idx] = action
        # ... store other components
        self.position = (self.position + 1) % self.capacity
        self.size = min(self.size + 1, self.capacity)
```

Q-Network Implementation

```python
class QNetwork(nn.Module):
    def __init__(self, obs_dim, n_actions,
                 hidden_sizes=(128, 128)):
        super().__init__()

        layers = []
        input_size = obs_dim

        # Build hidden layers
        for hidden_size in hidden_sizes:
            layers.append(nn.Linear(input_size, hidden_size))
            layers.append(nn.ReLU())
            input_size = hidden_size

        # Output layer (Q-values for each action)
        layers.append(nn.Linear(input_size, n_actions))

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)  # Shape: [batch_size, n_actions]
```

Target Network Mechanism

```python
def hard_update(target_net, online_net):
    """Copy weights from online to target network"""
    target_net.load_state_dict(online_net.state_dict())

def soft_update(target_net, online_net, tau=0.005):
    """Polyak averaging update"""
    with torch.no_grad():
        for target_param, param in zip(
            target_net.parameters(),
            online_net.parameters()
        ):
            target_param.data.copy_(
                tau * param.data + (1 - tau) * target_param.data
            )

# Usage in training loop
if step % target_update_freq == 0:
    hard_update(target_network, q_network)
# OR for soft updates:
soft_update(target_network, q_network, tau=0.005)
```

DQN Loss Computation

```python
def compute_dqn_loss(batch, q_network, target_network,
                     gamma=0.99):
    obs, actions, rewards, next_obs, dones = batch

    # Current Q-values for taken actions
    q_values = q_network(obs)  # [B, n_actions]
    q_values = q_values.gather(1, actions.unsqueeze(1))
    q_values = q_values.squeeze()  # [B]

    # Compute targets with target network
    with torch.no_grad():
        next_q_values = target_network(next_obs)  # [B, n_actions]
        next_q_max = next_q_values.max(1)[0]  # [B]

        # TD targets
        targets = rewards + gamma * (1 - dones) * next_q_max

    # Huber loss
    loss = F.huber_loss(q_values, targets)
    return loss
```

Double DQN: Reducing Overestimation

```python
def compute_double_dqn_loss(batch, q_network, target_network,
                            gamma=0.99):
    obs, actions, rewards, next_obs, dones = batch

    # Current Q-values
    q_values = q_network(obs).gather(1, actions.unsqueeze(1))
    q_values = q_values.squeeze()

    with torch.no_grad():
        # Action selection with online network
        next_q_online = q_network(next_obs)
        next_actions = next_q_online.argmax(1, keepdim=True)

        # Action evaluation with target network
        next_q_target = target_network(next_obs)
        next_q_values = next_q_target.gather(1, next_actions)
        next_q_values = next_q_values.squeeze()

        targets = rewards + gamma * (1 - dones) * next_q_values

    return F.huber_loss(q_values, targets)
```

Main Training Loop

```python
def train_dqn(env, n_episodes=1000):
    q_network = QNetwork(obs_dim, n_actions).to(device)
    target_network = QNetwork(obs_dim, n_actions).to(device)
    target_network.load_state_dict(q_network.state_dict())

    optimizer = optim.Adam(q_network.parameters(), lr=1e-3)
    buffer = ReplayBuffer(capacity=10000, obs_dim=obs_dim)

    for episode in range(n_episodes):
        obs, _ = env.reset()
        episode_reward = 0

        while not done:
            # Select action (epsilon-greedy)
            action = select_action(obs, epsilon)
            next_obs, reward, done, _, _ = env.step(action)

            # Store and learn
            buffer.push(obs, action, reward, next_obs, done)
            if len(buffer) >= batch_size:
                batch = buffer.sample(batch_size)
                loss = compute_dqn_loss(batch, q_network,
                                        target_network)
                # ... optimize
```

TensorBoard Integration

```python
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter('runs/dqn_experiment')

# Log scalars
writer.add_scalar('loss/td_error', loss.item(), step)
writer.add_scalar('metrics/episode_reward', reward, episode)
writer.add_scalar('metrics/epsilon', epsilon, step)

# Log hyperparameters and metrics
writer.add_hparams(
    {'lr': 1e-3, 'batch_size': 32, 'gamma': 0.99},
    {'final_reward': best_reward}
)

# Visualize in terminal:
# tensorboard --logdir runs/
```

Key Metrics to Track: Loss, episode rewards, Q-values, epsilon, learning rate

Saving and Loading Models

```python
def save_checkpoint(agent, filepath):
    checkpoint = {
        'q_network': agent.q_network.state_dict(),
        'target_network': agent.target_network.state_dict(),
        'optimizer': agent.optimizer.state_dict(),
        'episode': agent.episode,
        'epsilon': agent.epsilon,
        'replay_buffer': agent.buffer,  # Optional
        'config': agent.config
    }
    torch.save(checkpoint, filepath)

def load_checkpoint(filepath):
    checkpoint = torch.load(filepath, map_location=device)

    agent.q_network.load_state_dict(checkpoint['q_network'])
    agent.target_network.load_state_dict(
        checkpoint['target_network'])
    agent.optimizer.load_state_dict(checkpoint['optimizer'])
    # ... restore other components

    return checkpoint['episode']
```

Proper Evaluation

```python
def evaluate(agent, env, n_episodes=10):
    """Evaluate agent without exploration"""
    eval_rewards = []

    for episode in range(n_episodes):
        obs, _ = env.reset(seed=seed + episode)
        episode_reward = 0
        done = False

        while not done:
            # Greedy action selection (no exploration)
            with torch.no_grad():
                q_values = agent.q_network(
                    torch.FloatTensor(obs).unsqueeze(0))
                action = q_values.argmax().item()

            obs, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            episode_reward += reward

        eval_rewards.append(episode_reward)

    return np.mean(eval_rewards), np.std(eval_rewards)
```

Hyperparameter Tuning Strategy

Grid Search Example:

```
param_grid = {
    'lr': [1e-3, 3e-4, 1e-4],
    'batch_size': [32, 64, 128],
    'target_update': [100, 500, 1000],
    'buffer_size': [10000, 50000]
}
```

Important Relationships:

- Larger buffer $\rightarrow$ more stable but slower
- Frequent target updates $\rightarrow$ less stable
- Larger batch $\rightarrow$ more stable gradients
- Higher γ $\rightarrow$ longer-term planning

Start with:

- Published hyperparameters for similar tasks
- Conservative values (small LR, large buffer)
- Tune one parameter at a time

Debugging Checklist

Common Issues and Solutions:

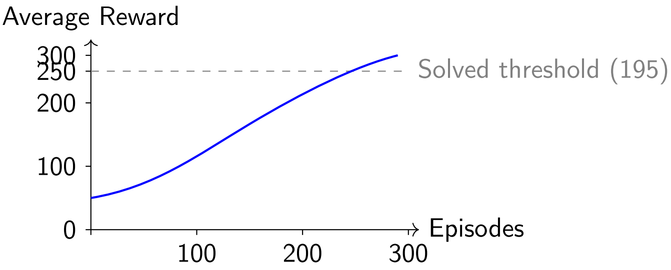
1. **Q-values exploding**
   - Check reward scale
   - Verify target network updates
   - Use gradient clipping
2. **No learning progress**
   - Increase exploration (ϵ)
   - Check replay buffer filling
   - Verify loss computation
3. **Unstable training**
   - Reduce learning rate
   - Increase batch size
   - Use Huber loss instead of MSE
4. **Poor final performance**
   - Tune exploration schedule
   - Increase network capacity
   - Try Double DQN

Expected Results: CartPole-v1

Training Progression:

- Episodes 1-50: Random behavior (reward $\approx$ 20)
- Episodes 50-100: Learning begins (reward $\approx$ 50-100)
- Episodes 100-150: Rapid improvement (reward $\approx$ 150-200)
- Episodes 150+: Solved (reward = 500, maximum)

Typical Training Curve:



Component Impact Analysis:

Configuration	**Episodes to Solve**	**Final Reward**
Full DQN	150	500
No target network	Diverges	N/A
No replay buffer	300+	200
Small buffer (1000)	250	450
No exploration decay	400+	300
MSE loss (not Huber)	200	480
Double DQN	130	500

Key Insights:

- Target network is essential for stability
- Replay buffer significantly improves efficiency
- Proper exploration schedule crucial
- Double DQN provides modest improvement

Real-world DQN Applications

Game Playing:

- Atari 2600 games (original DQN paper)
- StarCraft II micro-management
- Poker and card games

Robotics:

- Robotic grasping
- Navigation in discrete spaces
- Task scheduling

Resource Management:

- Data center cooling (Google)
- Network routing
- Inventory management

Finance:

- Portfolio optimization
- Trading strategies
- Risk management

Performance Optimization Tips

Memory Efficiency:

- Use numpy arrays in replay buffer
- Store observations as uint8 if possible
- Implement circular buffer
- Clear gradients with `set_to_none=True`

Computation Speed:

- Batch environment steps (vectorized envs)
- Use torch.compile() for JIT optimization
- Enable AMP on compatible GPUs
- Profile with torch.profiler

Training Stability:

- Normalize observations
- Clip rewards to [-1, 1] range
- Use learning rate scheduling
- Monitor gradient norms

Key Mathematical Results

TD Error:

$$\delta_t = r_t + \gamma \max_{a'} Q_{\theta^-}(s_{t+1}, a') - Q_\theta(s_t, a_t) \tag{10}$$

Gradient Update:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(\theta_t) \tag{11}$$

Loss Gradient:

$$\nabla_\theta L = -\mathbb{E}_{(s,a,r,s')\sim\mathcal{D}} \left[\delta_t \nabla_\theta Q_\theta(s, a)\right] \tag{12}$$

Optimal Q-function satisfies:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \quad \forall s, a \tag{13}$$

Summary

Key Theoretical Contributions:

1. **Problem:** Q-learning fails with function approximation
2. **Solution 1:** Experience replay breaks correlation
3. **Solution 2:** Target network stabilizes learning
4. **Enhancement:** Double DQN reduces overestimation

Mathematical Foundation:

- Semi-gradient TD learning
- Bellman optimality with approximation
- Convergence not guaranteed but works in practice

Critical Insights:

- Deadly triad: FA + bootstrapping + off-policy
- Replay enables i.i.d. sampling assumption
- Target network breaks feedback loops