Reinforcement Learning

Lecture 7: DQN Project (CartPole Case Study)

Taehoon Kim

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

Fall Semester 2025

Today's Agenda

- Review DQN architecture and revisit prior experiments
- Diagnose overfitting and instability in value-based deep RL
- Walk through implementation building blocks (buffer, network, training loop)
- Compare advanced DQN variants and interpret real experiments
- Practice debugging workflows and consolidate project takeaways

We will close with a live code walkthrough from `exp09_complete_dqn_project.py` to demonstrate the debugging workflow end-to-end.

Section Overview: Hands-on Experiments

Structure

- Nine runnable scripts (`exp01`–`exp09`) build the DQN project incrementally
- Each slide pair details objectives, tasks, and expected outputs
- Artifacts land in `experiments/figures/`, `runs/`, or run-specific checkpoint folders

Execution Tips

- Activate the virtual environment and export deterministic env vars before sweeps
- Use `python lecture07/experiments/exp09_integrated_test.py` for end-to-end validation

Experiment Scripts

- `exp01_environment_setup.py`: Verify env, device, seeding *(instrumentation + reproducibility)*
- `exp02_q_learning_basics.py`: Bellman update and Q-net demo *(tabular vs. neural view)*
- `exp03_replay_buffer.py`: Circular buffer and sampling analysis *(data decorrelation)*
- `exp04_dqn_network.py`: Architectures, target updates, inits *(model capacity sweeps)*
- `exp05_training_loop.py`: End-to-end DQN training loop *(buffer → optimizer)*
- `exp06_hyperparameter_tuning.py`: LR/batch/target sweeps *(search automation)*
- `exp07_advanced_dqn.py`: Double/Dueling comparisons *(variant ablations)*
- `exp08_debugging_visualization.py`: Gradients and Q diagnostics *(failure triage)*
- `exp09_complete_dqn_project.py`: Integrated DQN project *(production checklist)*

Expected Outputs

- Figures: `../experiments/figures/`
  - `dqn_training_results.png` (from exp05)
  - `advanced_dqn_comparison.png` (from exp07)
  - `q_value_analysis.png`, `training_diagnostics.png` (from exp08)
- Logs: `runs/` (from exp09)
- Checkpoints: under each run directory (from exp09)

Learning Objectives

By the end of this lecture you will

- Implement and validate a DQN pipeline end-to-end (from buffer to evaluation)
- Understand and prevent overfitting in value-based RL
- Master hyperparameter tuning for DQN
- Implement advanced DQN variants (Double, Dueling)
- Create reproducible RL experiments
- Debug common DQN issues effectively

Prerequisites

- Understanding of Q-learning (Lecture 5)
- Basic DQN concepts (Lecture 6)
- PyTorch proficiency

Section Overview: DQN Review & Setup

Goals

- Revisit the DQN building blocks before deep dives
- Ground the CartPole case study in environment specs and reproducibility
- Align code folders, helper utilities, and artifact layout

Keep in Mind

- Slides cite code from `exp04`–`exp06`
- Deterministic seeds (`setup_seed`) and device helpers carry into every lab

DQN Architecture Recap

Key Components:

- Neural network Q-function
- Experience replay buffer
- Target network
- ϵ-greedy exploration

Update Rule:

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

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

where H_κ is Huber loss

H_κ denotes the piecewise smooth L_1 loss, reducing sensitivity to outliers in TD errors.

CartPole-v1 Environment

Observation Space:

- Cart position: $x \in [-4.8, 4.8]$
- Cart velocity: $\dot{x} \in [-\infty, \infty]$
- Pole angle: $\theta \in [-0.418, 0.418]$ rad
- Pole angular velocity: $\dot{\theta} \in [-\infty, \infty]$

Action Space:

- 0: Push cart to the left
- 1: Push cart to the right

Reward:

- +1 for each step the pole remains upright
- Episode ends if pole falls or cart leaves bounds

Project Structure

```
DQN_Project/
|-- experiments/
|    |-- exp01_environment_setup.py
|    |-- exp02_q_learning_basics.py
|    |-- exp03_replay_buffer.py
|    |-- exp04_dqn_network.py
|    |-- exp05_training_loop.py
|    |-- exp06_hyperparameter_tuning.py
|    |-- exp07_advanced_dqn.py
|    |-- exp08_debugging_visualization.py
|    |-- exp09_complete_dqn_project.py
|-- runs/              # TensorBoard logs
|-- checkpoints/       # Model checkpoints
```

Reproducibility Setup

```python
def setup_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

device = torch.device(
    'cuda' if torch.cuda.is_available()
    else 'mps' if hasattr(torch.backends, 'mps') and torch.backends.mps.is_available()
    else 'cpu'
)
```

Key: Consistent seeding across all libraries

Reproducibility Checklist

- ☐ Fixed seeds for Python, NumPy, PyTorch
- ☐ Environment reset with deterministic seed (e.g., `state, info = env.reset(seed=42)`)
- ☐ Library versions documented
- ☐ Hardware information logged
- ☐ Configuration saved with unique hash
- ☐ Checkpoints include RNG states
- ☐ TensorBoard logging enabled
- ☐ Evaluation uses fixed seeds

Goal: Anyone can reproduce your results exactly

Section Overview: Overfitting & Instability

Key Questions

- How do we recognize when value-based agents memorize replay data?
- Which metrics from `exp05`–`exp07` flag brittle policies?
- What mitigation knobs (data, network, regularizers) should we reach for first?

Artifacts Referenced

- Training curves + diagnostics: `exp05_training_loop.py`
- Variant ablations: `exp07_advanced_dqn.py`

Overfitting in Value-Based RL

Three Common Symptoms:

1. Loss oscillates despite sufficient data
2. Greedy policy becomes brittle after improvement
3. Performance varies strongly across seeds

Causes:

- Agent exploits narrow replay window
- Memorizes recent transitions
- Fails to generalize to full on-policy distribution
- Replay buffer sampling bias limits coverage
- Non-stationary targets (policy + target net drift)

Remedies for Overfitting

Data Diversity:

- Large replay buffer
- Diverse exploration
- Stratified sampling
- Data augmentation (for image observations)

Network Stability:

- Target network
- Soft updates
- Gradient clipping

Regularization:

- Huber loss
- Reward scaling
- Early stopping

Architecture:

- Appropriate capacity
- Dropout (use carefully; can harm temporal consistency)
- Batch normalization

Target Network Mechanism

Problem: Moving target instability

- Q-learning target depends on network being updated
- Creates feedback loop and oscillations

Solution: Frozen target network

$$y = r + \gamma \max_{a'} Q_{\theta^-}(s', a') \tag{3}$$

where θ^- is periodically updated:

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

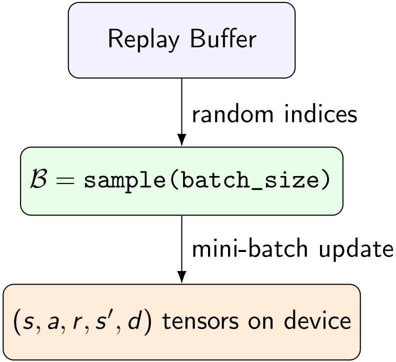
Experience Replay Benefits

Decorrelation

- Breaks temporal correlations in data
- Enables i.i.d. sampling assumption
- Stabilizes gradient estimates

Data Efficiency

- Reuses experiences multiple times
- Enables off-policy learning
- Smooths learning over rare events



See `ReplayBuffer.sample()` in `exp03_replay_buffer.py`.

Section Overview: Preprocessing & Hyperparameters

What We Cover

- Minimal preprocessing choices for CartPole vs. image domains
- Sensitivity ranking for LR, batch size, target update, exploration
- How `exp05` and `exp06` log sweeps for reproducibility

Link to Experiments

- Hyperparameter sweeps: `exp06_hyperparameter_tuning.py`
- Training loop knobs: `exp05_training_loop.py`

Preprocessing Pipeline

Common Preprocessing Steps:

1. Frame skipping (action repeat)
2. Frame stacking (temporal context)
3. Reward scaling/clipping
4. State normalization

For CartPole:

- Minimal preprocessing needed (already low-dimensional)
- Optional: state normalization
- Optional: frame stacking for velocity estimation

State normalization in exp05_training_loop.py

```
state = torch.from_numpy(state).float()
state = (state - obs_rms.mean) / (obs_rms.var.sqrt() + 1e-8)
```

Key Hyperparameters in DQN

Learning Dynamics

- Learning rate: 10^{-4} to 10^{-3}
- Batch size: 32 to 256
- Gradient clipping: 10.0

Exploration

- ϵ start: 1.0, end: 0.01–0.1
- Schedules: linear, exponential, or stepped

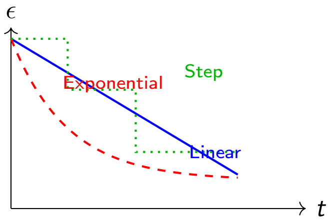
Memory & Updates

- Buffer size: 10K–100K
- Target update: 500–10K steps
- Discount γ: 0.99

Network

- Hidden dimensions: 128–256
- Activation: ReLU; Optimizer: Adam

Exploration schedules (visualized)



Hyperparameter Sensitivity Analysis

Most Critical (High Impact):

- Learning rate: Too high $\rightarrow$ instability
- Target update frequency: Balance stability vs staleness
- Exploration schedule: Coverage vs exploitation

Moderately Important:

- Batch size: Gradient variance
- Buffer size: Sample diversity
- Network capacity: Expressiveness (too large $\rightarrow$ slower convergence, overfitting risk)

Less Sensitive:

- Gradient clipping threshold
- Optimizer momentum

Section Overview: Implementation Details

Focus Areas

- Replay buffer memory layout and sampling logic
- Network definitions + loss computation snippets (`exp04`, `exp05`)
- Training loop scaffolding (epsilon decay, device transfers, tensor shapes)

Remember

- Add `[fragile]` when slides show code listings
- Reference exact helper names to mirror the experiments

Efficient Replay Buffer

```python
class ReplayBuffer:
    def __init__(self, state_dim, capacity, device):
        self.capacity = capacity
        self.pos = 0
        self.full = False
        # Pre-allocate arrays for efficiency
        self.states = np.zeros((capacity, state_dim), dtype=np.float32)
        self.actions = np.zeros(capacity, dtype=np.int64)
        self.rewards = np.zeros(capacity, dtype=np.float32)
        self.next_states = np.zeros((capacity, state_dim), dtype=np.float32)
        self.dones = np.zeros(capacity, dtype=np.bool8)
```

Key: Pre-allocation for memory efficiency

Circular Buffer Logic

```python
def add(self, state, action, reward, next_state, done):
    self.states[self.pos] = state
    self.actions[self.pos] = action
    self.rewards[self.pos] = reward
    self.next_states[self.pos] = next_state
    self.dones[self.pos] = done

    self.pos = (self.pos + 1) % self.capacity
    self.full = self.full or self.pos == 0

def sample(self, batch_size):
    max_idx = self.capacity if self.full else self.pos
    idx = np.random.randint(0, max_idx, size=batch_size)
    # Returns (s, a, r, s', d) tensors on the target device
```

DQN Network Architecture

```python
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=128):
        super().__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))  # [B, state_dim] -> [B, hidden]
        x = F.relu(self.fc2(x))  # [B, hidden] -> [B, hidden]
        x = self.fc3(x)          # [B, hidden] -> [B, action_dim]
        return x
```

DQN Loss Computation

```python
def compute_loss(batch, policy_net, target_net, gamma):
    states, actions, rewards, next_states, dones = batch

    # Current Q values
    current_q = policy_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

    # Target Q values
    with torch.no_grad():
        next_q = target_net(next_states).max(1)[0]
        target_q = rewards + gamma * next_q * (1 - dones)

    # Huber loss for stability
    loss = F.smooth_l1_loss(current_q, target_q)
    return loss
```

Tensor Shape Tracking

Critical for debugging:

- States: [batch_size, state_dim]
- Actions: [batch_size]
- Q-values: [batch_size, action_dim]
- Selected Q: [batch_size]
- Targets: [batch_size]

Common shape errors:

- Forgetting to squeeze/unsqueeze
- Mismatched batch dimensions
- Wrong gather dimension

Training Loop Structure

```python
for episode in range(num_episodes):
    state, _ = env.reset()
    done = False

    while not done:
        # Select action (epsilon-greedy)
        action = select_action(state, epsilon)

        # Environment step (Gymnasium API)
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated  # Replaces legacy Gym "done" flag

        # Store and update
        memory.push(state, action, reward, next_state, done)
        loss = update_network()

        state = next_state
```

Epsilon Decay Strategies

Linear

$$\epsilon_t = \epsilon_{start} + t \cdot \frac{\epsilon_{end} - \epsilon_{start}}{T}$$

Exponential

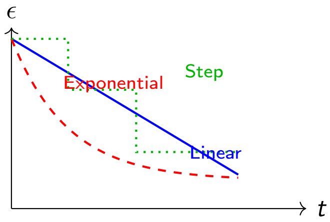
$$\epsilon_t = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end})e^{-t/\tau}$$

Step

$$\epsilon_t = \epsilon_{start} \cdot \alpha^{\lfloor t/k \rfloor}$$

Guidance

- Linear: predictable annealing for small projects
- Exponential: fast drop, good for long horizons
- Step: coarse schedule for staged curricula



Section Overview: Diagnostics Snapshot

Focus

- Surface quantitative evidence from `exp05`, `exp07`, and `exp08`
- Highlight observed metrics before diving into derivations
- Frame troubleshooting questions for the live walkthrough

Artifacts

- Figures stored under `../experiments/figures/`
- Console metrics captured alongside each experiment run

Section Overview: Advanced Variants

Objectives

- Contrast vanilla DQN with Double and Dueling implementations
- Clarify when soft vs. hard target updates help stability
- Tie back to metrics plotted in `exp07_advanced_dqn.py`

Scripts Referenced

- `exp07_advanced_dqn.py` (variant sweeps)
- `exp04_dqn_network.py` (architecture definitions)

Double DQN

Problem: Overestimation bias

- Standard DQN: max operator causes systematic overestimation
- Accumulates over iterations

Solution: Decouple selection and evaluation

$$a^\star = \arg\max_a Q_\theta(s', a) \quad \text{(selection)} \tag{4}$$
$$y = r + \gamma Q_{\theta^-}(s', a^\star) \quad \text{(evaluation)} \tag{5}$$

Result: Reduced bias, more stable learning

Double DQN Implementation

```python
# Standard DQN
with torch.no_grad():
    next_q = target_net(next_states).max(1)[0]
    target_q = rewards + gamma * next_q * (1 - dones)

# Double DQN
with torch.no_grad():
    # Use policy network to select actions
    next_actions = policy_net(next_states).argmax(1)
    # Use target network to evaluate
    next_q = target_net(next_states).gather(1, next_actions.unsqueeze(1)).squeeze(1)
    target_q = rewards + gamma * next_q * (1 - dones)
```

Dueling DQN Architecture

Motivation: Separate value and advantage

- Value $V(s)$: How good is this state?
- Advantage $A(s, a)$: How much better is this action?

Architecture:

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a') \tag{6}$$

Benefits:

- Better generalization across actions
- More efficient learning of state values
- Particularly effective when actions have similar values

$|\mathcal{A}|$ denotes the number of discrete actions at state s.

Dueling Network Implementation

```python
class DuelingDQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.feature = nn.Sequential(...)  # Shared layers

        # Value stream
        self.value = nn.Sequential(...)    # -> [B, 1]

        # Advantage stream
        self.advantage = nn.Sequential(...) # -> [B, action_dim]

    def forward(self, x):
        features = self.feature(x)
        value = self.value(features)
        advantage = self.advantage(features)
        # Combine with mean subtraction
        q = value + advantage - advantage.mean(dim=1, keepdim=True)
        return q
```

Soft Updates vs Hard Updates

Hard Update:

- $\theta^- \leftarrow \theta$ every C steps
- Sudden changes in target values
- Can cause instability

Soft Update (Polyak averaging):

- $\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$ every step
- $\tau \in [0.001, 0.01]$ typically
- Smoother target evolution
- More stable but slower adaptation

Gradient Clipping

Why clip gradients?

- Prevents exploding gradients
- Stabilizes training
- Though rooted in RNN training, often stabilizes DQN when rewards are sparse

```python
# Clip by norm (recommended)
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=10.0)

# Clip by value
torch.nn.utils.clip_grad_value_(model.parameters(), clip_value=1.0)
```

Experiment 1: Environment Setup

File: `exp01_environment_setup.py` **Objectives:**

- Verify all dependencies installed
- Test GPU/CPU device selection
- Confirm reproducible seeding
- Check CartPole environment

Run: `python exp01_environment_setup.py`

Expected Output:

- System information displayed
- Deterministic behavior confirmed
- Device correctly selected

Experiment 2: Q-Learning Basics

File: `exp02_q_learning_basics.py` **Objectives:**

- Q-table vs Q-network
- Bellman update equation
- Epsilon-greedy exploration
- Discount factor impact

Expected Output:

- Console trace showing $Q(2, 1)$ update from 0.0000 to 0.1000 (TD error 1.0000)
- Epsilon-greedy counts over 1000 trials: $(0.0 \rightarrow 0/1000)$, $(0.1 \rightarrow 51/949)$, $(0.5 \rightarrow 252/748)$, $(1.0 \rightarrow 521/479)$ random/greedy split
- Sample neural Q-network forward pass with shape $[1, 4] \rightarrow [1, 2]$

Key Insights:

- Q-values represent expected returns
- Neural networks approximate Q-tables
- Exploration essential for learning

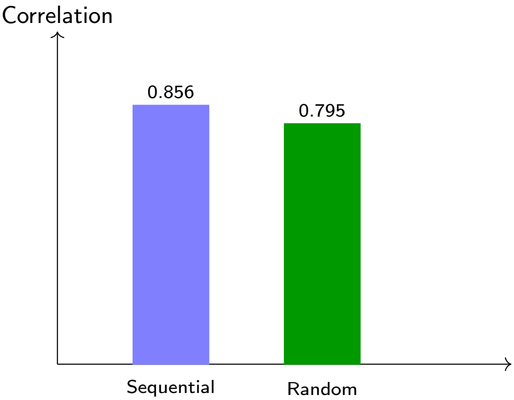
Experiment 3: Replay Buffer

File: `exp03_replay_buffer.py`
Implementation Details

- Circular buffer for memory efficiency
- Random sampling for decorrelation
- Pre-allocated NumPy arrays for zero-copy transfers
- Efficient tensor conversion on the target device

Latest run (CartPole-v1)

- Correlation drop: $0.856 \rightarrow 0.795$ (7.1%)
- Batch diversity: 17/32 unique episodes per sample
- Memory footprint: 4.29 MB for 100k transitions (45 B/transition)



Experiment 4: Network Architectures

File: `exp04_dqn_network.py` **Variations Tested:**

- Basic DQN (2 hidden layers)
- Deep DQN (4 hidden layers)
- Dueling DQN (separate streams)
- Different activation functions

Metrics:

- Parameter count
- Forward pass time
- Gradient flow analysis
- Dead neuron detection

Experiment 5: Complete Training Loop

File: `exp05_training_loop.py` **Components Integrated:**

- Environment interaction
- Experience collection
- Batch sampling and updates
- Target network updates
- Epsilon decay
- Loss tracking

Monitoring:

- Episode rewards
- Training loss
- Exploration rate
- Episode lengths

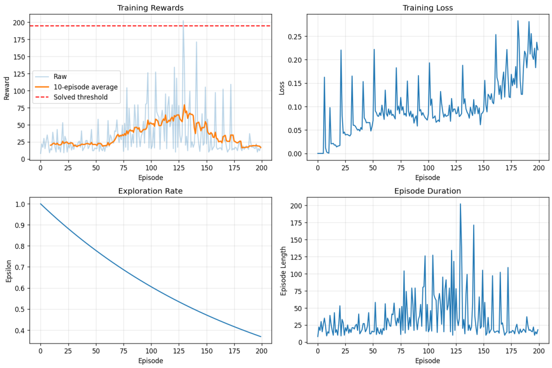
Experiment 5 Results: Training Loop

Key Metrics (CartPole-v1)

- Early vs late episode averages (see console logs)
- Best episode reward observed during training
- Greedy evaluation (10 runs): mean and standard deviation
- Mean Huber loss after warm-up

What to inspect

- Reward curve trending upward but not yet solved (>=195)
- Exploration anneals from $\epsilon = 1.0$ toward the target minimum
- Loss plateaus once replay buffer is well-mixed



Episode returns, loss, exploration rate, and episode length logged by `exp05_training_loop.py`.

Interpretation: if evaluation lags training returns, revisit replay mixing or target updates even when losses look stable.

Experiment 6: Hyperparameter Tuning

File: `exp06_hyperparameter_tuning.py` **Parameters Analyzed:**

- Learning rate: $[10^{-4}, 10^{-2}]$
- Batch size: $[32, 256]$
- Target update: $[100, 2000]$ steps
- Buffer size: $[1K, 50K]$
- Epsilon decay: linear vs exponential

Latest grid search (150 episodes each):

- Learning rate sweep winner: 5×10^{-4} (final avg return **47.4**)
- Smallest batch (32) reached **119** episodic best vs 82 for batch 128
- Fast epsilon decay (0.98) lifted final average to **95.8**
- Optimized setting (LR 5×10^{-4}, batch 64, target update 10) hit **211** reward by episode **81**
- Last-50 episode mean under that config: **59.5**

Experiment 7: Advanced DQN Variants

File: `exp07_advanced_dqn.py` **Latest metrics (400 episodes):**

Variant	$\bar{R}_{\text{last 50}}$	Best Episode	Solved (195+)
Vanilla	9.5	71	0%
Double	9.5	71	0%
Dueling	9.5	50	0%
Double+Dueling	9.5	50	0%

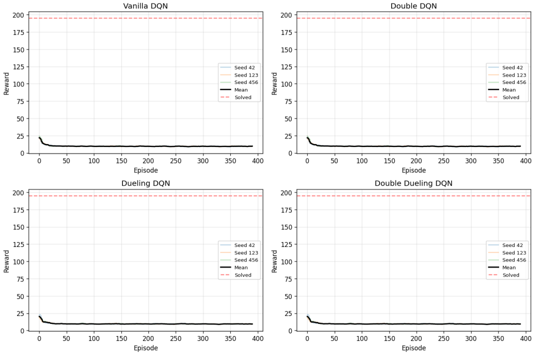
Interpretation: identical outcomes highlight insufficient training budget or exploration, not flaws in Double/Dueling mechanics.

Experiment 7 Results: Variant Comparison

Setup: 400-episode budget, three seeds, four variants (`exp07_advanced_dqn.py`).

- Metric: mean of last-50 episode returns
- Observation: all variants stay low with the current budget
- Action: increase data/episodes or exploration before tweaking architecture

Experiment 7 Results: Learning Curves



Smoothed learning curves for three random seeds per variant recorded by `exp07_advanced_dqn.py`.

Interpretation tip: inspect the shaded variance band—tight bands with flat returns usually indicate insufficient episodes rather than architectural failure.

Experiment 8: Debugging and Visualization

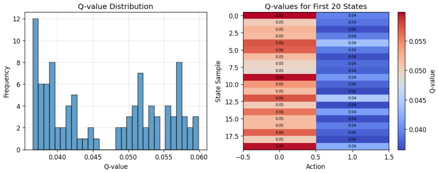
File: `exp08_debugging_visualization.py` **Debugging Toolkit:**

- Gradient flow monitoring
- Dead neuron detection
- Q-value distribution analysis
- Weight magnitude tracking
- Loss landscape visualization

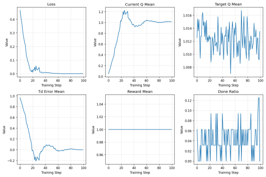
Common Issues Detected:

- Vanishing/exploding gradients
- Q-value overestimation
- Dead ReLU neurons
- Numerical instabilities

Experiment 8 Results: Diagnostics Snapshot



Q-value distribution + heatmap from `exp08_debugging_visualization.py`.



Rolling diagnostics across 100 monitored steps (loss, Q, TD error).

Checklist

- Gradient flow: per-layer norms / NaN scan
- Buffer stats: reward moments + done ratio
- Issue scanner: exploding values, dead neurons

Interpretation: widening Q-value spread signals healthy exploration mix; a flat spread flags under-exploration or tiny batches.

Experiment 9: Production-Ready DQN

File: `exp09_complete_dqn_project.py` **Features Implemented:**

- Configuration management
- TensorBoard logging
- Model checkpointing
- Automatic evaluation
- Resume from checkpoint
- Hyperparameter tracking

Best Practices:

- Modular design
- Type hints
- Comprehensive logging
- Error handling
- Mirror artifact layout (e.g., `runs/2025-11-05-cartpole/checkpoints/`)

TensorBoard Integration

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

writer = SummaryWriter(log_dir='runs/dqn_experiment')

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

# Log histograms
writer.add_histogram('q_values', q_values, step)

# Launch TensorBoard
# tensorboard --logdir runs
```

Checkpointing Strategy

```python
def save_checkpoint(model, optimizer, step, path):
    cuda_state = None
    if torch.cuda.is_available():
        try:
            cuda_state = torch.cuda.get_rng_state_all()
        except RuntimeError:
            cuda_state = None  # CPU-only context

    torch.save({
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'step': step,
        'rng_states': {
            'python': random.getstate(),
            'numpy': np.random.get_state(),
            'torch': torch.get_rng_state(),
            'cuda': cuda_state
        }
    }, path)
```

Performance Optimization Tips

Computational Efficiency:

- Use `torch.no_grad()` for inference
- Pre-allocate tensors when possible
- Batch operations over loops
- Enable cudNN autotuner*

Memory Efficiency:

- Clear gradients with `zero_grad(set_to_none=True)`
- Use in-place operations when safe
- Monitor GPU memory usage
- Implement gradient accumulation if needed

*Safe for ReLU-only networks; other activations may change numerics or determinism.

Common Pitfalls and Solutions

Pitfall 1: Forgetting torch.no_grad()

- Causes memory leak
- Solution: Always use for target computation

Pitfall 2: Wrong tensor shapes

- Silent broadcasting errors
- Solution: Assert shapes, use comments

Pitfall 3: Incorrect done handling

- Bootstrap from terminal states
- Solution: Multiply next Q by (1 - done)

```
target = reward + gamma * next_q * (1 - done.float())
```

Debugging Strategies

When training fails:

1. Check gradient flow (not zero, not exploding)
2. Verify loss decreasing (at least initially)
3. Monitor Q-value magnitudes (reasonable range)
4. Test with simpler environment first
5. Reduce problem complexity (smaller network, etc.)

Diagnostic plots:

- Loss over time
- Q-value distribution
- Episode rewards (smoothed)
- Gradient norms per layer

Hyperparameter Starting Points

For CartPole-v1:

- Learning rate: 10^{-3}
- Batch size: 128
- Buffer size: 10,000
- Target update: 1000 steps
- Hidden dims: [128, 128]
- Epsilon: $1.0 \rightarrow 0.01$ over 50K steps

For Atari games:

- Learning rate: 2.5×10^{-4}
- Batch size: 32
- Buffer size: 1,000,000
- Target update: 10,000 steps

Proper Evaluation Protocol

Requirements:

- Deterministic policy (epsilon = 0)
- Multiple episodes (10-30)
- Use unseen seeds for fair evaluation
- Report mean and std

Metrics to track:

- Episode return (total reward)
- Episode length
- Success rate (if applicable)
- Q-value estimates

Scaling to Complex Environments

For Image-based observations:

- Add CNN layers
- Frame stacking (4-8 frames)
- Grayscale conversion
- Downsampling (84x84 typical)

For Continuous actions:

- Switch to DDPG (Lillicrap et al., 2016) or TD3 (Fujimoto et al., 2018)
- Use actor-critic architecture
- Add action noise for exploration

Project Extensions

Algorithm improvements:

- Prioritized experience replay
- N-step returns
- Distributional DQN (C51)
- Rainbow DQN (all improvements)
- Noisy Nets (Fortunato et al., 2018)

Engineering improvements:

- Distributed training
- Vectorized environments
- ONNX export for deployment
- Real-time visualization

Section Overview: Wrap-up

We Close With

- Key takeaways linking theory slides to `exp09_complete_dqn_project.py`
- Recommended extensions and evaluation protocol reminders
- Next week's prep (rerun exp09 integrated test + update slides)

Action Items

- Regenerate figures if configs change (store under `experiments/figures/`)
- Push logs/checkpoints only when referenced as teaching artifacts

Key Takeaways

1. **Reproducibility is essential**
   - Fixed seeds, version logging, configuration tracking
2. **Start simple, add complexity gradually**
   - Basic DQN $\rightarrow$ Double $\rightarrow$ Dueling $\rightarrow$ Combined
3. **Debug systematically**
   - Monitor gradients, Q-values, losses
4. **Hyperparameters matter**
   - Learning rate and exploration most critical
   - Automate sweeps with Optuna or Ray Tune for reproducible tuning
5. **Combine techniques for best results**
   - Double + Dueling + proper tuning

Complete Implementation Checklist

- ✓ Environment setup and verification
- ✓ Replay buffer implementation
- ✓ Q-network architecture
- ✓ Training loop with proper updates
- ✓ Target network mechanism
- ✓ Epsilon-greedy exploration
- ✓ Loss computation and optimization
- ✓ Logging and visualization
- ✓ Checkpointing and recovery
- ✓ Evaluation protocol
- ✓ Hyperparameter tuning
- ✓ Advanced variants (Double, Dueling)

Performance Benchmarks

CartPole-v1 Results:

Method	Episodes to Solve	Final Score
Random Policy	Never	$\sim$20
Basic DQN	200-300	195+
Double DQN	150-250	200
Dueling DQN	180-280	198+
Double Dueling DQN	100-200	200

Solved = 195+ average reward over 100 episodes (seeds averaged over 3 runs).

Next Week: Policy Gradient Methods

Moving from value-based to policy-based:

- Direct policy optimization
- REINFORCE algorithm
- Policy gradient theorem
- Baseline techniques
- Continuous action spaces

Preparation:

- Review gradient computation
- Understand log probabilities
- Practice with distributions in PyTorch

Resources

Papers:

- DQN: Mnih et al. (2015) - Human-level control
- Double DQN: van Hasselt et al. (2016)
- Dueling DQN: Wang et al. (2016)
- Rainbow: Hessel et al. (2018)

Final Thoughts

"The key to success in DQN is
patience, systematic debugging,
and careful hyperparameter tuning"

Remember:

- Start with working baseline
- Change one thing at a time
- Always verify improvements
- Document everything