Deep Q-Learning Projects

Final Project Presentation

By Adil Akhmetov & Perizat Yessenova

🎮 Atari Breakout & 🚦 Traffic Light Control

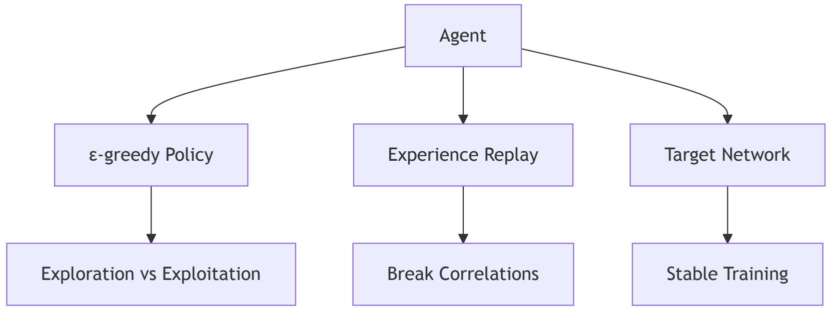
PyTorch + Gymnasium

Project Overview

Two implementations of **Deep Q-Learning (DQN)**:

1. **Atari Breakout (1976)**
   - Classic arcade game
   - Visual input (CNN)
   - 4 discrete actions
2. **Traffic Light Control**
   - Real-world application
   - State-based input (MLP)
   - 2 discrete actions

Core DQN Components



Task 1: Atari Breakout

Deep Q-Learning for Classic Gaming

Breakout: Problem Definition

Goal

Train an agent to play Breakout by breaking bricks with a ball

Environment

- **Platform**: Gymnasium ALE/Breakout-v5
- **Original frame**: 210×160 RGB pixels
- **Reward**: +1 per brick destroyed
- **Episode ends**: When ball is missed

Actions (4 total)

Action	Description
NOOP	Do nothing
FIRE	Launch ball
LEFT	Move paddle left
RIGHT	Move paddle right

State Preprocessing Pipeline

Transformations

```
# 1. Convert to grayscale
grayscale_obs=True

# 2. Resize to 84×84
screen_size=84

# 3. Frame skip (4 frames)
frame_skip=4

# 4. Stack 4 frames
FrameStackObservation(env, stack_size=4)
```

Why Stack Frames?

- Single frame → **no motion information**
- Ball direction is unknown
- Ball speed is unknown

Solution: Stack 4 consecutive frames

- Shape: `(4, 84, 84)`
- Agent "sees" temporal dynamics

CNN Architecture

```python
class DQN_CNN(nn.Module):
    def __init__(self, input_channels=4, num_actions=4):
        # Convolutional layers
        self.conv1 = nn.Conv2d(input_channels, 32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)

        # Input: 84×84 → conv1: 20×20 → conv2: 9×9 → conv3: 7×7

        # Fully connected layers
        self.fc1 = nn.Linear(64 * 7 * 7, 512)
        self.fc2 = nn.Linear(512, num_actions)  # Output: Q-values
```

Layer	Input	Output	Parameters
Conv1	4×84×84	32×20×20	8,224
Conv2	32×20×20	64×9×9	32,832
Conv3	64×9×9	64×7×7	36,928

Experience Replay Buffer

Implementation

```python
class ReplayBuffer:
    def __init__(self, capacity=100000):
        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):
        batch = random.sample(
            self.buffer, batch_size
        )
        return zip(*batch)
```

Why Experience Replay?

1. **Breaks correlation** between consecutive samples
2. **Reuses** past experiences efficiently
3. **Stabilizes** training

Configuration

- Buffer size: **100,000** transitions
- Batch size: **32**
- Warmup: 10,000 steps before training

ε-Greedy Exploration

Strategy

$$a = \begin{cases} \text{random action} & \text{with probability } \varepsilon \\ \arg\max_a Q(s,a) & \text{with probability } 1-\varepsilon \end{cases}$$

Decay Schedule

```
epsilon = epsilon_end +
    (epsilon_start - epsilon_end) *
    max(0, 1 - steps / epsilon_decay)
```

Parameters

Parameter	Value
ε start	1.0
ε end	0.01
Decay steps	100,000

```
ε: 1.0 ━━━━━━━━━\
                 \
                  \━━━━━━━━ 0.01
    0         100K        steps
```

Target Network

The Problem

Without target network:

- Q-values are **moving targets**
- Training becomes **unstable**
- "Chasing a moving goal"

The Solution

Separate **target network** updated periodically:

```
# Every 1000 steps
target_net.load_state_dict(
    policy_net.state_dict()
)
```

Q-Learning Update

$$Q(s, a) \leftarrow r + \gamma \max_{a'} Q_{\text{target}}(s', a')$$

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

# Compute loss
loss = F.smooth_l1_loss(current_q, target_q)
```

Breakout Hyperparameters

Training Configuration

Parameter	Value
Learning Rate	2.5×10^{-4}
Discount (γ)	0.99
Batch Size	32
Buffer Size	100,000
Target Update	Every 1000 steps
Optimizer	Adam
Loss Function	Smooth L1 (Huber)

Training Scale

Parameter	Value
Episodes	5,000 - 10,000
Steps per episode	Up to 10,000
Warmup steps	10,000
Gradient clipping	10

Estimated Training Time

- GPU: 4-8 hours
- CPU: 24-48 hours

Task 2: Traffic Light Control

Adaptive DQN for Real-World Application

Breakout: Training Summary

5k-episode run (completed)

- Episodes 120-220: rewards ≈ 0.0–0.5, ε ≈ 0.92–0.95
- Episodes 1350-1380: rewards ≈ 2.0–3.1, ε ≈ 0.19–0.22
- Episodes 4900-5000: rewards ≈ 4.9, ε ≈ 0.01
- Overall (full 5k): avg reward ≈ 4.0, avg bricks ≈ 4.0

Breakout: Evaluation Snapshot

Segment	Episodes	Avg Reward	Avg Bricks	Notes
Early	120–220	~0.3	~0.3	Mainly launching ball, rare hits
Mid	1350–1380	~2.6	~2.6	Tracks ball, clears a few bricks per life
Completed run	4900–5000	~4.9	~4.9	Stable volleys; still drops after sharp wall bounces

Interpretation

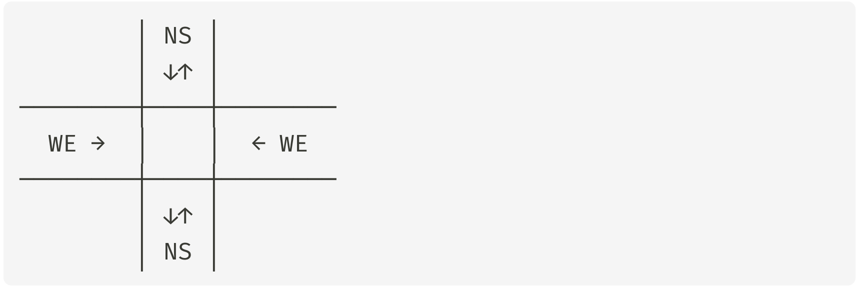
- Average reward == bricks destroyed (unit reward per brick)
- Agent still loses balls after side-wall rebounds; needs longer training
- ε already <0.2 in mid segment; expect further gains as $\varepsilon \to 0.01$

Breakout: Behavior Notes

- Consistently launches ball quickly (FIRE) after reset
- Keeps paddle under ball for straight trajectories
- Misses after sharp wall bounces → indicates need for more training / frame history
- No reward hacking observed; gameplay aligns with intended rules
- Behavior visualization: training curves (reward/loss/ε) via `plot_training_history` in `pf/final_project.ipynb`; gameplay render/GIF not included due to deadline constraints

Traffic Control: Problem Definition

Intersection Setup



Two directions: NS (North-South), WE (West-East)

Traffic Light Phases

Phase	NS	WE
0	🟢 Green	🔴 Red
1	🔴 Red	🟢 Green

Goal

Minimize queue lengths by **adaptive** phase selection

TrafficEnv: Custom Environment

```python
class TrafficEnv:
    # State: [queue_NS, queue_WE, phase]
    def _get_state(self):
        return np.array([self.queue_ns, self.queue_we, self.phase])

    # Dynamics: each step
    def step(self, action):
        # 1. Update phase
        switching = (action != self.phase)
        self.phase = action

        # 2. Cars pass on green (up to 3)
        if self.phase == 0:  # NS green
            self.queue_ns -= min(self.queue_ns, 3)
        else:  # WE green
            self.queue_we -= min(self.queue_we, 3)

        # 3. New cars arrive (Poisson process)
        self.queue_ns += np.random.poisson(1.2)
        self.queue_we += np.random.poisson(0.8)

        # 4. Calculate reward
        reward = -(self.queue_ns + self.queue_we)
        if switching:
            reward -= 0.5  # Penalty for switching
```

DQN Network for Traffic Control

Simple MLP Architecture

```python
class DQN_FC(nn.Module):
    def __init__(self,
                 input_size=3,
                 hidden_size=64,
                 num_actions=2):
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, num_actions)
        )
```

Network Structure

```
Input (3)
    │
    ▼
Linear → ReLU (64)
    │
    ▼
Linear → ReLU (64)
    │
    ▼
Linear → ReLU (64)
    │
    ▼
Linear (2)  ← Q-values
```

Total parameters: ~9,000

Traffic Control Hyperparameters

Agent Configuration

Parameter	Value
Learning Rate	1×10^{-3}
Discount (γ)	0.99
Batch Size	64
Buffer Size	10,000
Target Update	Every 100 steps
Hidden Size	64

Training Configuration

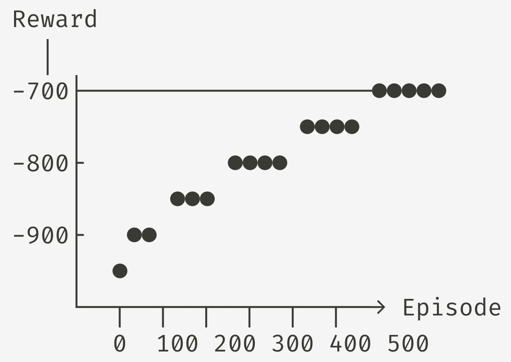
Parameter	Value
Episodes	500
Steps per episode	200
ε start	1.0
ε end	0.01
ε decay	3,000 steps
Switch penalty	0.5

Training Results

Training Progress

```
Episode  50: Reward=-876, Queue=4.10, ε=0.046
Episode 100: Reward=-723, Queue=3.35, ε=0.011
Episode 150: Reward=-734, Queue=3.41, ε=0.010
Episode 200: Reward=-745, Queue=3.43, ε=0.010
Episode 300: Reward=-741, Queue=3.44, ε=0.010
Episode 400: Reward=-747, Queue=3.46, ε=0.010
Episode 500: Reward=-735, Queue=3.42, ε=0.010
```

Learning Curve



Agent learns to maintain **low queue lengths** (~3.4 cars avg)

Comparison: DQN vs Fixed Timing

Experimental Results

Controller	Avg Reward	Avg Queue
Fixed (5 steps)	-1,393.76	6.87
Fixed (10 steps)	-2,111.30	10.51
Fixed (15 steps)	-2,835.30	14.14
Fixed (20 steps)	-3,561.42	17.78
DQN Agent	**-690.36**	**3.16**

Key Findings

🏆 DQN Improvement: **50.5%**

Over best fixed-timing controller

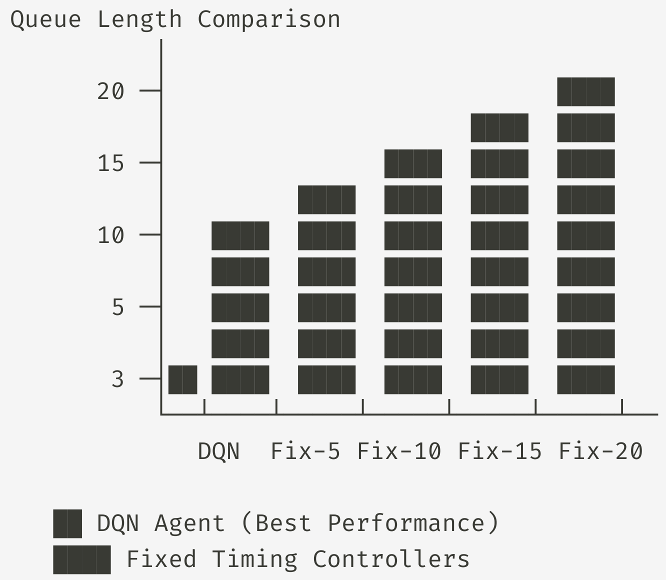
Why DQN Wins

1. **Adapts** to current traffic conditions
2. **Predicts** future queue buildup
3. **Balances** both directions dynamically

Fixed Timing Problems

- Cannot adapt to traffic variations
- Either too fast or too slow switching

Performance Visualization



DQN reduces average queue by 54% compared to best fixed timing!

Key DQN Components Summary

1. Experience Replay

- Stores transitions in buffer
- Random sampling breaks correlations
- Enables sample reuse

2. Target Network

- Separate network for Q-targets
- Periodic updates (every N steps)
- Stabilizes training

3. ε-Greedy Policy

- Balances exploration/exploitation
- Decays over time
- Ensures sufficient exploration

4. Neural Network

- **CNN** for visual input (Breakout)
- **MLP** for state vectors (Traffic)
- Outputs Q-values for all actions

Implementation Highlights

Technologies Used

- **PyTorch** - Deep learning framework
- **Gymnasium** - RL environments
- **NumPy** - Numerical computing
- **Matplotlib** - Visualization

Code Quality

- Type hints throughout
- Modular class design
- Configurable hyperparameters
- Model save/load functionality

Project Structure

```
pf/
├── final_project.ipynb
│   ├── Task 1: Breakout DQN
│   │   ├── DQN_CNN
│   │   ├── ReplayBuffer
│   │   ├── DQNAgent
│   │   └── Training loop
│   │
│   └── Task 2: Traffic Control
│       ├── TrafficEnv
│       ├── DQN_FC
│       ├── TrafficDQNAgent
│       └── Comparison analysis
│
└── presentation.md
```

Conclusions

Task 1: Breakout

- ✅ CNN architecture implemented
- ✅ Frame preprocessing pipeline
- ✅ DQN with all components
- ✅ Ready for 5-10K episode training

Task 2: Traffic Control

- ✅ Custom environment created
- ✅ DQN agent trained (500 episodes)
- ✅ **50.5% improvement** over baseline
- ✅ Adaptive behavior demonstrated

🎯 DQN successfully applied to both visual and state-based RL problems!

Thank You!

Questions?

Repository: `pf/final_project.ipynb`

Key Results:

- Breakout: Full DQN implementation with CNN
- Traffic: 50.5% improvement over fixed timing