PROJECT REPORT
Deep Q-Learning for CartPole Based on
“Reinforcement learning with analogue memristor arrays”
Wang et al., 2019, Nature Electronics

1. Introduction

Reinforcement Learning (RL) enables autonomous decision-making based only on environment feedback. The referenced article demonstrates Deep Q-Learning implemented using analogue memristor arrays to achieve high energy efficiency. In this project, the CartPole control problem is solved using the same algorithmic components of Deep Q-Learning, implemented in software using PyTorch. The goal is to keep the pole balanced as long as possible, using only the reward signal from the environment.

Environment: CartPole-v1
State size: 4 [cart_pos, cart_vel, angle, ang_vel]
Action space: 2 (left, right)

2. Article Requirements and What I Implemented

Requirement 1
✔Deep Q-Learning including Bellman loss, RMSProp optimizer, and experience replay.
• Article reference (page 3–4):
“The agent randomly samples a minibatch… Bellman equation… RMSprop optimizer…”
- My implementation (inside learn method in DQNAgent):

```python
predicted_q = self.q_network(states).gather(1, actions)
with torch.no_grad():
    next_q = self.q_network(next_states).max(1, keepdim=True).values
    next_q[dones] = 0.0

targets = rewards + self.discount * next_q

loss = self.criterion(predicted_q, targets)
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

Requirement 2
✔Three-layer fully connected neural network with ReLU activation.
• Article uses 4-48-48-2 architecture for CartPole (Figure 3b).
- My implementation (network.py):

```python
class DQNNetwork(nn.Module):  3 usages  Saleh Mir Mohamma
    def __init__(self, num_actions, input_dim):  Saleh
        super(DQNNetwork, self).__init__()

        # Fully Connected (FC) model
        self.FC = nn.Sequential(
            nn.Linear(input_dim,  out_features: 48),
            nn.ReLU(),

            nn.Linear( in_features: 48,  out_features: 48),
            nn.ReLU(),

            nn.Linear( in_features: 48, num_actions)
        )
```

Requirement 3
✔Epsilon-greedy exploration with decay over time.
• Article (page 4):
“trial-and-error exploration… gradually switches to greedy policy”
- My implementation:

```python
# Epsilon update using ε(t) = ε_min + (ε_max - ε_min) * exp(-λ * t)
def update_epsilon(self, steps_done):  1 usage  Saleh Mir Mohammad Rezaei
    self.epsilon = self.epsilon_min + (self.epsilon - self.epsilon_min) * np.exp(-self.epsilon_decay * steps_done)
```

```python
# exploration
if np.random.rand() < self.epsilon:
    return self.action_space.sample()                    # randomly picks left or right in CartPole

# exploitation
if not torch.is_tensor(state):
    state = torch.as_tensor(state, dtype=torch.float32, device=device)  # Convert state to tensor

with torch.no_grad():                                    # Disable gradient tracking (faster + no memory waste)
    q_values = self.q_network(state)                     # Compute Q-values: [Q_left, Q_right]
    return torch.argmax(q_values).item()                 # Pick action with the highest expected reward
```

Requirement 4
✔Experience Replay buffer to reduce correlation between transitions.
• Article (page 3):
“experience replay improves learning stability”
- My implementation (ReplayMemory.py):

```python
class ReplayMemory:  2 usages  Saleh Mir Mohammad Rezaei
    """
    Experience Replay buffer storing past transitions for off-policy learning.
    - Improves sample efficiency by reusing experiences
    - Stabilizes training compared to learning only from latest transition
    - deques is an efficient structure for sliding memory buffer
    """

    def __init__(self, capacity):...

    def store(self, state, action, next_state, reward, done):  1 usage (1 dynamic)  Saleh Mir Mohammad Rezaei
        """
        Store a new transition into the replay buffer.
        FIFO behavior: if full, oldest transitions are automatically discarded.
        """
        self.states.append(state)
        self.actions.append(action)
        self.next_states.append(next_state)
        self.rewards.append(reward)
        self.dones.append(done)

    def sample(self, batch_size):  2 usages (1 dynamic)  Saleh Mir Mohammad Rezaei
        """
        Randomly sample a batch of transitions for training.
        Returns tensors directly on the correct device (CPU/GPU).
        """
        # Generate random unique indices,  replace=False → prevents picking same memory twice
        indices = np.random.choice(len(self), size=batch_size, replace=False)
```

Requirement 5
✔Online Learning while interacting with the environment.
• Article (page 5): continuous update during gameplay.
- My implementation (trainer.py):

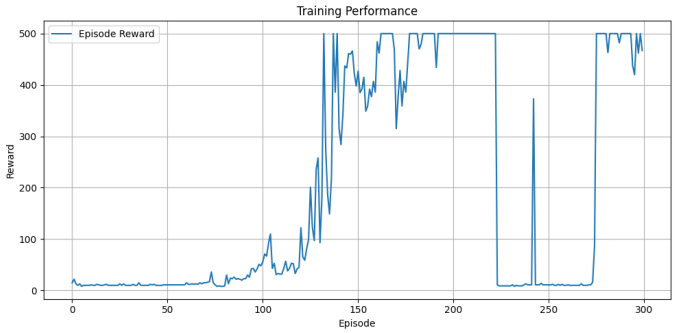
```python
while not done:                                  # The agent keeps taking steps until episode ends
    action = self.agent.select_action(state)     # Using epsilon-greedy strategy—exploration or exploitation
    next_state, reward, terminated, truncated, _ = self.env.step(action) # Environment responds
    done = terminated or truncated
    self.agent.replay_memory.store(state, action, next_state, reward, done) # This is essential for off-policy learning
    if len(self.agent.replay_memory) > self.batch_size:         # Only learn when enough samples collected
        self.agent.learn(self.batch_size, done)
```

The agent learns during each episode, based only on reward outcomes.

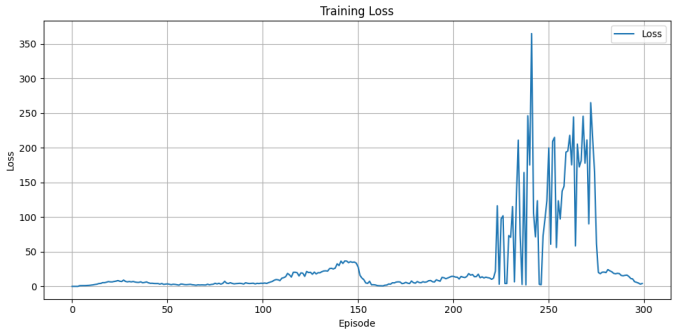
Requirement 6
✔Using one neural network for training
• Article (page 8): Removal of the target Q̂-network. … Given the fixed size of the memristor chip, only the Q-network is used in this work
- My implementation (dqn_agent.py):

```python
# self.q_network(states) → outputs all Q-values
# .gather(1, actions) → picks only Q-values of the taken actions
predicted_q = self.q_network(states).gather(1, actions)       # This is the Q(s, a) value from Bellman equation

# Max future reward if the episode is not terminal
with torch.no_grad():
    next_q = self.q_network(next_states).max(dim=1, keepdim=True).values   # Choose max Q-value for each next state
    next_q[dones] = 0.0

# Now build the Bellman Target
targets = rewards + self.discount * next_q
```

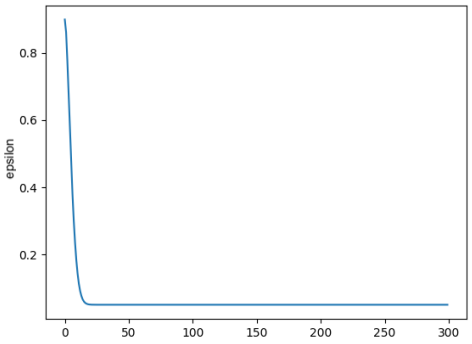
3. Results and Performance



The training curve shows steady improvement from low initial rewards to consistently reaching the maximum score of 500. The occasional performance drops occur because the model uses only a single Q-network without a target network, which can cause instability during learning. Despite this, the agent ultimately succeeds in mastering the CartPole task.



The loss initially decreases as Q-value estimates improve, and then fluctuates significantly during periods of rapid learning when rewards increase sharply. The large spikes in loss between episodes 220–260 correspond to unstable Q-updates caused by bootstrapping directly from the same network being trained. This is a known limitation of vanilla DQN without a target network — the agent can momentarily diverge, leading to sudden overshooting and performance drops. Overall, the loss behavior aligns with the observed reward instability.



The epsilon decay plot shows how the agent transitions from exploration to exploitation during training. At the beginning, epsilon is high (around 0.9), leading to mostly random actions to explore the environment. As training progresses, epsilon rapidly decreases, encouraging the agent to rely more on learned Q-values rather than random actions. After approximately 30–40 episodes, epsilon reaches its minimum threshold of 0.05, ensuring that the agent continues to take occasional exploratory actions while primarily exploiting its learned policy. This decay behavior supports efficient and stable learning by balancing exploration and exploitation over time.