Tarea Práctica: Implementación de Deep Q-Networks (DQN) con PyTorch y Gymnasium

Este documento describe la tarea práctica sobre Deep Q-Networks (DQN). El objetivo es implementar un agente DQN utilizando PyTorch para resolver un entorno clásico de control proporcionado por la librería `gymnasium`. Se basará en los conceptos teóricos vistos en `00_intro.md` y la estructura de implementación de agentes vista en `rl_algorithms/`.

---

1. Objetivo

Implementar desde cero un agente DQN capaz de aprender una política para resolver entornos de `gymnasium` con espacios de observación continuos (o de alta dimensionalidad) y espacios de acción discretos. Se evaluará la comprensión de los componentes clave de DQN:

- Red Neuronal para aproximar la Q-function.
- Experience Replay Buffer.
- Target Network para estabilizar el aprendizaje.
- Estrategia de exploración ε-greedy con decaimiento.

---

2. Entornos Sugeridos

Se recomienda implementar y probar el agente DQN en al menos uno de los siguientes entornos de `gymnasium`:

1. **`CartPole-v1`**:
    - **Objetivo**: Balancear un poste sobre un carrito móvil.
    - **Espacio de Observación**: Continuo (4 dimensiones: posición del carro, velocidad del carro, ángulo del poste, velocidad angular del poste).
    - **Espacio de Acción**: Discreto (2 acciones: mover el carro a la izquierda o a la derecha).
    - **Recompensa**: +1 por cada paso de tiempo que el poste se mantiene vertical.
    - **Condición de Éxito (Típica)**: Mantener el balance por un promedio de 195.0 (o 475.0 en versiones más recientes de `gymnasium`) pasos durante 100 episodios consecutivos.
2. **`LunarLander-v2`**:
    - **Objetivo**: Aterrizar suavemente un módulo lunar en una plataforma de aterrizaje.
    - **Espacio de Observación**: Continuo (8 dimensiones: coordenadas x, y, velocidades x, y, ángulo, velocidad angular, indicadores de contacto de las patas).
    - **Espacio de Acción**: Discreto (4 acciones: no hacer nada, encender motor izquierdo, encender motor principal, encender motor derecho).
    - **Recompensa**: Varía según la distancia al aterrizaje, velocidad, ángulo, contacto con el suelo, y uso de combustible. Aterrizar exitosamente da una recompensa alta (+100), estrellarse da una penalización (-100).

PROF

- **Condición de Éxito (Típica)**: Obtener un promedio de recompensa de 200.0 sobre 100 episodios consecutivos.

Se recomienda comenzar con `CartPole-v1` por ser más simple y rápido de entrenar.

---

3. Componentes a Implementar

El código debe estar organizado modularmente. Se sugiere una estructura similar a la de `rl_algorithms/`, pero adaptada a DQN:

- `dqn_agent.py`: Contendrá la clase principal `DQNAgent`.
- `models.py`: Definirá la arquitectura de la red neuronal (Q-Network) usando `torch.nn`.
- `replay_buffer.py`: Implementará la clase `ReplayBuffer`.
- `train_dqn.py`: Script principal para configurar, entrenar, evaluar y visualizar el agente.
- `requirements.txt`: Listará las dependencias (`gymnasium`, `torch`, `numpy`, `matplotlib`).

3.1. Red Neuronal Q-Network (`models.py`)

- Usar `torch.nn.Module` para definir la red.
- Se presenta el codigo para un shallow network, pero pueden usar cualquier otra red neuronal (se recomienda una CNN)
- Para `CartPole-v1`, una red neuronal simple fully-connected (MLP) con 2-3 capas ocultas (e.g., 64, 128 o 256 neuronas por capa) y activación ReLU suele ser suficiente.
  - **Entrada**: El tamaño del espacio de observación (`obs_space.shape[0]`, e.g., 4 para CartPole).
  - **Salida**: El tamaño del espacio de acción (`action_space.n`, e.g., 2 para CartPole), produciendo un Q-value estimado para cada acción.
- **Ejemplo de Estructura (MLP simple):**

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class QNetwork(nn.Module):
    def __init__(self, obs_dim, action_dim):
        super(QNetwork, self).__init__()
        self.layer1 = nn.Linear(obs_dim, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, action_dim)

    def forward(self, x):
        # Asegurarse que x es un tensor flotante
        if not isinstance(x, torch.Tensor):
            x = torch.tensor(x, dtype=torch.float32)
        # Mover a device si es necesario (gestionar fuera o dentro)
        # x = x.to(device)
        x = F.relu(self.layer1(x))
```

```
        x = F.relu(self.layer2(x))
        return self.layer3(x) # Salida son Q-values (logits)
```

- Para `LunarLander-v2`, una arquitectura similar pero posiblemente con más neuronas (e.g., 256) puede ser necesaria.
- Asegurarse de que el `forward` method acepta un estado (o un batch de estados) y devuelve los Q-values correspondientes.
- **Hint**: No olvides gestionar el `device` (`.to(device)`) donde se ejecutan los cálculos (CPU o GPU).

3.2. Replay Buffer (`replay_buffer.py`)

- Implementar una clase `ReplayBuffer`.
- **Hint**: Usar `collections.deque(maxlen=capacity)` es eficiente. Considera usar `collections.namedtuple` para almacenar las transiciones de forma organizada:

```
from collections import namedtuple, deque
import random

Transition = namedtuple('Transition',
                        ('state', 'action', 'reward', 'next_state',
'done'))
```

- Debe tener métodos para:
  - `__init__(self, capacity)`: Inicializar el buffer (e.g., `self.memory = deque([], maxlen=capacity)`).
  - `push(self, *args)`: Almacenar una transición. Usar `self.memory.append(Transition(*args))`. Asegúrate de que los datos (`state`, `next_state`, etc.) sean convertibles a tensores más tarde (NumPy arrays o ya tensores).
  - `sample(self, batch_size)`: Muestrear aleatoriamente un batch de transiciones del buffer.
    - Usar `random.sample(self.memory, batch_size)`.
    - **Hint**: La salida de `random.sample` será una lista de `Transition` tuples. Necesitas "descomprimirla" para obtener batches separados de estados, acciones, etc.

```
# Ejemplo dentro de sample:
transitions = random.sample(self.memory, batch_size)
# Transponer el batch (convertir lista de Transiciones a
Transición de listas/tuplas)
batch = Transition(*zip(*transitions))

# Convertir cada componente a tensores de PyTorch
# Cuidado con los tipos de datos (float para estados/rewards,
long para acciones, bool/uint8 para dones)
state_batch = torch.cat([torch.tensor(s,
dtype=torch.float32).unsqueeze(0) for s in batch.state])
```

PROF

```
        action_batch = torch.tensor(batch.action,
        dtype=torch.long).unsqueeze(1) # Necesita shape [batch_size,
        1] para gather
        reward_batch = torch.tensor(batch.reward,
        dtype=torch.float32).unsqueeze(1)
        next_state_batch = torch.cat([torch.tensor(ns,
        dtype=torch.float32).unsqueeze(0) for ns in batch.next_state])
        done_batch = torch.tensor(batch.done,
        dtype=torch.float32).unsqueeze(1) # Float para poder
        multiplicar por él

        return state_batch, action_batch, reward_batch,
        next_state_batch, done_batch
```

- `__len__(self)`: Devolver `len(self.memory)`.

3.3. Agente DQN (`dqn_agent.py`)

- Clase `DQNAgent`.
- **`__init__`**:
  - Inicializar la `policy_net` y la `target_net` (con la misma arquitectura definida en `models.py`). **Hint**: Moverlas al `device` apropiado (`self.device = torch.device(...)`, `self.policy_net.to(self.device)`).
  - Copiar los pesos de `policy_net` a `target_net`: `self.target_net.load_state_dict(self.policy_net.state_dict())`.
  - **Importante**: Poner la `target_net` en modo evaluación: `self.target_net.eval()`. Esto desactiva capas como Dropout si las hubiera. La `policy_net` se mantiene en modo `train()`.
  - Inicializar el optimizador (e.g., `torch.optim.Adam(self.policy_net.parameters(), lr=learning_rate)`) para la `policy_net`.
  - Inicializar el `ReplayBuffer(capacity)`.
  - Guardar hiperparámetros: `gamma`, `epsilon_start`, `epsilon_end`, `epsilon_decay` (o pasos para decaimiento), `batch_size`, `target_update_frequency`, `learning_rate`.
  - Guardar referencia al entorno (`env`), `action_space`, `observation_space` y `device`.
  - Inicializar `epsilon` actual a `epsilon_start`.
  - Inicializar contador de pasos `steps_done = 0`.
- **`choose_action(self, state, evaluate=False)`**:
  - Implementar la lógica ε-greedy.
  - **Hint**: Calcular `epsilon` actual. Puede ser un decaimiento lineal o exponencial:

```
        # Ejemplo decaimiento exponencial (calcular fuera o dentro,
        basado en steps_done)
        # epsilon = self.epsilon_end + (self.epsilon_start -
        self.epsilon_end) * \
        #           math.exp(-1. * self.steps_done /
```

PROF

```
        self.epsilon_decay_steps)
        # O simplemente decrementar linealmente hasta epsilon_end
```

- Convertir el `state` (NumPy array) a un tensor de PyTorch: `state_tensor = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)` (añade dimensión de batch).
- Obtener Q-values de la `policy_net`. **Importante**: Usar `with torch.no_grad():` para desactivar el cálculo de gradientes durante la inferencia.

```
with torch.no_grad():
    # t.max(1) devuelve valor máximo y índice; solo necesitamos el índice (acción)
    q_values = self.policy_net(state_tensor)
```

- Si `evaluate=True` o `random.random() > epsilon`:
  - Seleccionar la acción con el máximo Q-value: `action = q_values.max(1)[1].view(1, 1)`
- Si no (explorar):
  - Seleccionar una acción aleatoria: `action = torch.tensor([[self.env.action_space.sample()]], device=self.device, dtype=torch.long)`
- Devolver la acción seleccionada como un entero: `return action.item()`.
- **Nota**: El decaimiento de `epsilon` y el incremento de `steps_done` pueden manejarse aquí o, más comúnmente, en el bucle de entrenamiento después de cada paso.

- **`learn(self)`**:
  - Método principal para la actualización de la red.
  - Verificar si el buffer tiene suficientes muestras: `if len(self.replay_buffer) < self.batch_size: return`.
  - Muestrear un `batch` del `ReplayBuffer`: `states, actions, rewards, next_states, dones = self.replay_buffer.sample(self.batch_size)`. **Hint**: Asegúrate que los tensores muestreados estén en el `device` correcto (`states = states.to(self.device)`, etc.).
  - **Calcular Q(s, a)**: Los Q-values que la `policy_net` *predice* para las acciones que *realmente* se tomaron.
    - Obtener todos los Q-values para los estados del batch: `all_current_q_values = self.policy_net(states)`
    - Seleccionar solo los Q-values correspondientes a las acciones tomadas en el batch: `current_q_values = all_current_q_values.gather(1, actions)`
    - `gather(1, actions)` selecciona a lo largo de la dimensión 1 (acciones) usando los índices proporcionados por `actions`. `actions` debe tener shape `[batch_size, 1]`.

PROF

- **Calcular V(s') = max_a' Q_target(s', a')**: El valor máximo predicho por la `target_net` para el siguiente estado.
    - Obtener los Q-values del siguiente estado de la `target_net`: `next_q_values = self.target_net(next_states)`
    - Seleccionar el máximo Q-value a lo largo de la dimensión de acción: `max_next_q_values = next_q_values.max(1)[0].unsqueeze(1)`
    - `.max(1)[0]` obtiene los valores máximos.
    - `.unsqueeze(1)` les da shape `[batch_size, 1]` para que coincida con `current_q_values`.
    - **Importante**: Usar `.detach()` para evitar que los gradientes fluyan hacia la `target_net`: `max_next_q_values = max_next_q_values.detach()`
- **Calcular el valor objetivo (Target Q-value)**: $y_j = r_j + \gamma \max_{a'} Q_{\text{target}}(s'_j, a') (1 - d_j)$
    - `target_q_values = rewards + (self.gamma * max_next_q_values * (1 - dones))`
    - Multiplicar por `(1 - dones)` asegura que el valor futuro sea cero si el estado es terminal (`done=1`).
- **Calcular la pérdida (Loss)**: Usar un criterio como `SmoothL1Loss` (Huber loss, menos sensible a outliers que MSE) o `MSELoss`.
    - `criterion = nn.SmoothL1Loss()`
    - `loss = criterion(current_q_values, target_q_values)`
- **Realizar el paso de optimización**:
    - `self.optimizer.zero_grad()` # Limpiar gradientes antiguos
    - `loss.backward()` # Calcular gradientes
    - `torch.nn.utils.clip_grad_value_(self.policy_net.parameters(), 100)` # Opcional: Recortar gradientes para estabilidad
    - `self.optimizer.step()` # Actualizar pesos de policy_net

PROF

- **`update_target_network(self)`**:
    - Copiar los pesos de `policy_net` a `target_net`.
    - `self.target_net.load_state_dict(self.policy_net.state_dict())`

3.4. Bucle de Entrenamiento (`train_dqn.py`)

- **Configuración**:
    - Definir hiperparámetros (capacidad del buffer, batch size, gamma, lr, target update frequency, número de episodios, epsilon params).
    - **Device**: Configurar el dispositivo (`device = torch.device("cuda" if torch.cuda.is_available() else "cpu")`).
- **Inicialización**:
    - Crear el entorno: `env = gymnasium.make("CartPole-v1")` (o `LunarLander-v2`).

- Crear el agente DQN: `agent = DQNAgent(env.observation_space.shape[0], env.action_space.n, device=device, **hyperparams)`.
- Listas para guardar recompensas, etc.: `episode_rewards = []`.
- Contador total de pasos: `total_steps = 0`.

- **Bucle Principal (por `i_episode` in range(`num_episodes`))**:
  1. Resetear entorno: `state, info = env.reset()`.
  2. **Hint**: Convertir estado inicial a tensor: `state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)`.
  3. Inicializar recompensa acumulada: `current_episode_reward = 0`.
  4. Bucle Interno (usar `itertools.count()` o un `for t in range(max_steps_per_episode)`)
     a. Elegir acción: `action = agent.choose_action(state.cpu().numpy()[0])` # choose_action puede esperar numpy
     b. Ejecutar acción en el entorno: `observation, reward, terminated, truncated, _ = env.step(action)`. `done = terminated or truncated`.
     c. Acumular recompensa: `current_episode_reward += reward`.
     d. **Hint**: Preparar datos para el buffer. Convertir `reward` a tensor, `action` a tensor, `observation` a tensor. `done_tensor` indicando si el *nuevo* estado `observation` es terminal.
     ```
     python reward_tensor = torch.tensor([reward], device=device) action_tensor = torch.tensor([[action]], device=device, dtype=torch.long) # Asegurar [[action]] done_tensor = torch.tensor([done], device=device, dtype=torch.float32) # Float para multiplicar if done: next_state_tensor = None # O un tensor de ceros si tu buffer lo requiere else: next_state_tensor = torch.tensor(observation, dtype=torch.float32, device=device).unsqueeze(0)
     ```
     e. Almacenar transición en el buffer: `agent.replay_buffer.push(state, action_tensor, reward_tensor, next_state_tensor, done_tensor)`.
     f. Actualizar estado: `state = next_state_tensor`.
     g. Incrementar contador total de pasos: `total_steps += 1`.
     h. Llamar a `agent.learn()` para realizar un paso de optimización (se encargará de verificar si hay suficientes muestras).
     i. Actualizar la `target_net` cada `target_update_frequency` pasos: `if total_steps % agent.target_update_frequency == 0: agent.update_target_network()`.
     j. **Hint**: Actualizar Epsilon (si el decaimiento se basa en `total_steps`).
     k. Terminar el episodio si `done` es True (`break` del bucle interno).
  5. Registrar recompensa del episodio: `episode_rewards.append(current_episode_reward)`.
  6. **Hint**: Realizar decaimiento de épsilon (si el decaimiento se basa en episodios).
  7. Opcional: Evaluar periódicamente el rendimiento sin exploración y guardar el mejor modelo (`torch.save(agent.policy_net.state_dict(), 'best_model.pth')`).
  8. Imprimir progreso (e.g., cada 10 episodios).
- **Visualización**: Graficar la recompensa por episodio (usar `matplotlib`). Aplicar una media móvil para suavizar la curva y ver la tendencia.

PROF

- **Evaluación Final**: Ejecutar el agente entrenado (cargando el mejor modelo si se guardó, y usando `evaluate=True` en `choose_action`) durante varios episodios y mostrar el rendimiento promedio. Opcionalmente, renderizar algunos episodios (`env = gymnasium.make("CartPole-v1", render_mode="human")`).

---

4. Librerías Requeridas (`requirements.txt`)

```
# Actualizado a versiones estables ~ mediados 2024
gymnasium[classic_control,box2d]>=0.29.1,<1.0
torch>=2.1.0,<2.3.0
numpy>=1.24.0,<1.27.0
matplotlib>=3.7.0,<3.10.0
# collections (viene con Python stdlib)
```

(`box2d` es necesario para `LunarLander-v2`. Instalar con: `pip install gymnasium[box2d]`)
(**Nota**: Las versiones exactas pueden variar, pero estas son referencias recientes y compatibles.)

---

5. Ejercicios Propuestos

Estos ejercicios están diseñados para profundizar la comprensión y experimentar con la implementación de DQN.

Parte 1: Implementación y Verificación Básica

1. **Implementar Componentes**: Completa las clases `DQNAgent`, `ReplayBuffer`, y la red neuronal en `models.py` siguiendo las especificaciones y hints proporcionados.
    - **Hint**: Empieza creando cada clase y método con la estructura básica. Rellena la lógica paso a paso.
    - **Debug Hint**: Usa `print()` generosamente para verificar las formas (`.shape`) y tipos (`.dtype`) de los tensores en puntos clave (e.g., salida del buffer, antes/después de `gather`, entrada/salida de redes).
2. **Entrenamiento en `CartPole-v1`**:
    - **Tarea**: Configura `train_dqn.py` para `CartPole-v1`. Elige hiperparámetros razonables (puedes buscar valores comunes online o empezar con los sugeridos: `BUFFER_SIZE=10000`, `BATCH_SIZE=128`, `GAMMA=0.99`, `EPS_START=0.9`, `EPS_END=0.05`, `EPS_DECAY=1000` (pasos), `TARGET_UPDATE=500` (pasos), `LR=1e-4`). Entrena durante ~500-1000 episodios.
    - **Hint**: ¡Comienza con un número bajo de episodios (e.g., 50) y `TARGET_UPDATE` pequeño (e.g., 10) solo para verificar que el bucle corre sin errores y la pérdida disminuye (aunque sea un poco)! Luego incrementa los valores para un entrenamiento real.
    - **Preguntas**:
        - ¿La curva de recompensa muestra una tendencia ascendente? ¿Alcanza el agente la condición de éxito (promedio de ~475 en 100 episodios para v1)?

- Observa el comportamiento del agente renderizado después del entrenamiento. ¿Parece estable?

Parte 2: Análisis de Componentes DQN

3. **Efecto del Replay Buffer**:
    - **Tarea**: Reduce drásticamente el `BUFFER_SIZE` (e.g., a 500 o igual al `BATCH_SIZE`). Vuelve a entrenar.
    - **Preguntas**:
        - ¿Cómo afecta un buffer pequeño al aprendizaje? ¿Es más rápido, más lento, más inestable?
        - Compara la curva de recompensas con la del buffer grande. ¿Por qué el Experience Replay es crucial para la estabilidad? (Relaciona con correlaciones temporales y eficiencia de datos).
4. **Efecto de la Target Network**:
    - **Tarea**: Modifica el código para que la `target_net` se actualice en *cada* paso (`TARGET_UPDATE=1`). Alternativamente, modifica `learn` para usar `policy_net` en lugar de `target_net` para calcular `max_next_q`. Vuelve a entrenar.
    - **Preguntas**:
        - ¿Cómo afecta la actualización frecuente (o la eliminación) de la target network al aprendizaje? ¿Converge el agente? ¿Es estable?
        - Explica el problema del "objetivo móvil" y cómo la target network lo mitiga.
5. **Efecto de la Frecuencia de Actualización del Target**:
    - **Tarea**: Compara diferentes valores de `TARGET_UPDATE` (e.g., 10, 500 (inicial), 5000).
    - **Preguntas**:
        - ¿Hay una diferencia notable en la estabilidad o velocidad de convergencia? ¿Qué problemas podrían surgir con una actualización demasiado infrecuente o demasiado frecuente?

PROF

Parte 3: Ajuste de Hiperparámetros

6. **Tasa de Aprendizaje (Learning Rate)**:
    - **Tarea**: Experimenta con diferentes `LR` (e.g., 1e-3, 1e-4 (inicial), 1e-5).
    - **Preguntas**:
        - ¿Cómo afecta el LR a la velocidad y estabilidad del aprendizaje? ¿Un LR demasiado alto causa divergencia? ¿Uno demasiado bajo hace el aprendizaje muy lento?
7. **Tamaño del Batch (Batch Size)**:
    - **Tarea**: Prueba diferentes `BATCH_SIZE` (e.g., 32, 128 (inicial), 512).
    - **Preguntas**:
        - ¿Un batch más grande lleva a un aprendizaje más estable? ¿Afecta la velocidad de cómputo por actualización? ¿Cómo podría interactuar con el tamaño del buffer y la tasa de aprendizaje?

8. **Parámetros de Exploración (Epsilon)**:
    - **Tarea**: Modifica `EPS_DECAY` (e.g., decaimiento más rápido o más lento), `EPS_START`, y `EPS_END`.
    - **Preguntas**:
        - ¿Cómo afecta la duración de la exploración alta inicial al rendimiento? ¿Qué pasa si el decaimiento es demasiado rápido o demasiado lento? ¿Es importante mantener un `EPS_END` pequeño pero > 0?

Parte 4: Entorno Más Complejo

9. **Entrenamiento en `LunarLander-v2`**:
    - **Tarea**: Adapta la configuración para `LunarLander-v2` (ajusta el tamaño de entrada/salida de la red, posiblemente aumenta la capacidad de la red, el tamaño del buffer, y el número de episodios de entrenamiento - puede necesitar 2000+ episodios). Ajusta hiperparámetros si es necesario.
    - **Preguntas**:
        - ¿Logra el agente aprender a aterrizar? ¿Alcanza la condición de éxito?
        - ¿Qué hiperparámetros fueron más críticos para ajustar en este entorno más complejo en comparación con CartPole?
        - ¿Observas algún comportamiento interesante o estrategia aprendida durante el renderizado?

Parte 5: Extensiones (Opcional Avanzado)

10. **Implementar Double DQN (DDQN)**:
    - **Tarea**: Modifica el método `learn` para implementar Double DQN. El cambio clave es cómo se calcula el valor del siguiente estado:
        1. Usa la `policy_net` para *seleccionar* la mejor acción para el siguiente estado: $a^* = \arg\max_{a'} Q_{\text{policy}}(s_{j+1}, a', \theta)$.
        2. Usa la `target_net` para *evaluar* el valor de esa acción seleccionada: $y_j = r_j + \gamma Q_{\text{target}}(s_{j+1}, a^*, \theta^-)$.
    - **Preguntas**: Compara el rendimiento (curva de recompensas, estabilidad) de DDQN con DQN estándar en `CartPole-v1` o `LunarLander-v2`. ¿Reduce DDQN la sobreestimación de los Q-values (difícil de ver directamente, pero a menudo resulta en un aprendizaje más estable)?

PROF

---

6. Entrega

- Código fuente completo (`.py` files, `requirements.txt`).
- Un informe breve (e.g., en un archivo `README.md` o PDF) que describa:
    - La estructura del código.
    - Los hiperparámetros finales utilizados para cada entorno probado.
    - Gráficos de las curvas de recompensa por episodio (con media móvil).
    - Respuestas concisas a las preguntas de los ejercicios seleccionados (especialmente las partes 2, 3 y 4).
    - Cualquier desafío encontrado y cómo se resolvió.

- Observaciones sobre el rendimiento y comportamiento de los agentes entrenados.