Respuestas detalladas para defensa del proyecto

Proyecto: DQN aplicado a CarRacing-v3 (Gymnasium). Fecha: 2025-12-20. Formato: A4.

Este documento responde en detalle 3 preguntas frecuentes, citando explícitamente archivos y funciones del repositorio y mostrando fragmentos de código clave.

1) ¿Qué tipo de procesamiento se está utilizando?

Resumen

El proyecto aplica un preprocesamiento visual clásico para RL con entradas de imagen: (1) conversión a escala de grises, (2) resize a 96×96, (3) normalización a [0,1], y (4) apilado temporal de 4 frames (frame stacking). Esto reduce dimensionalidad, elimina color y permite capturar movimiento (información temporal) sin recurrir a una RNN.

¿Dónde está implementado en el código?

- • Preprocesamiento del frame: src/preprocesamiento.py → process_state_image(frame)
- • Stack inicial: src/preprocesamiento.py → generar_stack_inicial(frame)
- • Actualización del stack: src/preprocesamiento.py → actualizar_stack(stack, frame)
- • Construcción del estado final: src/preprocesamiento.py → generar_state_frame_stack_from_queue(queue)

Estas funciones se usan en el loop de entrenamiento (src/entrenamiento.py) y también en evaluación (tests/probar_modelo.py).

Orden de procesamiento (paso a paso)

- 1. El entorno entrega un frame (obs) RGB de 96×96×3.
- 2. Se transforma a escala de grises y se hace resize a 96×96.
- 3. Se convierte a float32 y se normaliza dividiendo entre 255.
- 4. Se mantiene un deque de longitud 4 con los últimos frames procesados.
- 5. El estado final es np.stack(queue, axis=0) con forma (4, 96, 96).

Fundamento (por qué se hace así)

Este pipeline es habitual en DQN visual: la escala de grises reduce canales y complejidad, la normalización mejora estabilidad numérica, y el stacking permite inferir velocidad/dinámica. En el repositorio se cita como referencia el estilo Atari-DQN (Mnih et al., 2013; Mnih et al., 2015).

Explicación del código (fragmentos clave)

```
# src/preprocesamiento.py

def process_state_image(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    resized = cv2.resize(gray, (96, 96))
    return resized.astype(np.float32) / 255.0


def generar_stack_inicial(frame):
    procesado = process_state_image(frame)
    return deque([procesado] * STATE_STACK, maxlen=STATE_STACK)
```

```
def actualizar_stack(stack, frame):
    stack.append(process_state_image(frame))
    return stack


def generar_state_frame_stack_from_queue(queue):
    return np.stack(queue, axis=0)
```

2) ¿Qué tipo de red neuronal se usa en el DQN?

Resumen

Se utiliza una CNN (Convolutional Neural Network) implementada en PyTorch. La red recibe el estado con forma (4, 96, 96) y produce un vector de 12 Q-valores, uno por cada acción discretizada.

¿Dónde está implementada?

- • Definición de la red: src/agente.py → class DQN(nn.Module)
- • Uso de la red en el agente: src/agente.py → class AgenteDQN

Estructura de la arquitectura (alto nivel)

- • Entrada: 4 canales (stack de frames) × 96 × 96.
- • Bloque conv: Conv2d → ReLU → MaxPool → Conv2d → ReLU → MaxPool.
- • Bloque denso: Flatten → Linear(flat_dim→216) → ReLU → Linear(216→12).

¿Por qué CNN?

Una CNN explota la estructura espacial de la imagen (bordes, texturas, formas) y suele ser la elección estándar para observaciones visuales. En este proyecto la CNN funciona como aproximador de Q(s,a).

Explicación del código (fragmentos clave)

```
# src/agente.py

class DQN(nn.Module):
    def __init__(self, num_acciones):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(STATE_STACK, 6, kernel_size=7, stride=3),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(6, 12, kernel_size=4),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )

        with torch.no_grad():
            dummy = torch.zeros(1, STATE_STACK, STATE_HEIGHT, STATE_WIDTH)
            conv_out = self.conv(dummy)
            flat_dim = conv_out.numel()

        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(flat_dim, 216),
            nn.ReLU(),
            nn.Linear(216, num_acciones)
        )

    def forward(self, x):
        return self.fc(self.conv(x))
```

3) ¿Cuál es mi acción? (steer, gas, brake) y discretización

Acción original del entorno (continua)

En CarRacing-v3, la acción original del entorno es un vector continuo de 3 componentes: (steer, gas, brake). Conceptualmente: steer controla la dirección, gas acelera y brake frena. En Gymnasium se entrega como un vector numérico (usualmente float32) y el agente debe producir 3 valores continuos.

¿Qué hace el proyecto? Discretización a 12 acciones

Como el DQN clásico predice Q-valores para un conjunto finito de acciones, el proyecto discretiza el control continuo en 12 acciones fijas. Cada acción es una tupla (steer, gas, brake) con valores típicos: steer $\in$ {-1,0,1}, gas $\in$ {0,1} y brake $\in$ {0,0.2}.

¿Dónde está implementado?

- • Definición del ACTION_SPACE: src/entrenamiento.py → ACTION_SPACE
- • Misma definición en evaluación: tests/probar_modelo.py → ACTION_SPACE
- • Selección de acción por índice: src/agente.py → seleccionar_accion() y memorize()
- • Conversión a np.float32 antes de env.step(): src/entrenamiento.py y tests/probar_modelo.py

Tipo de variables y flujo de la acción

- • En el espacio discreto, una acción es una tupla de Python (steer, gas, brake).
- • Al ejecutar en el entorno, se convierte a np.array(..., dtype=np.float32).
- • En la memoria, se almacena el índice (int) de la acción para compactar.

Explicación del código (fragmentos clave)

```
# src/entrenamiento.py (y tests/probar_modelo.py)

ACTION_SPACE = [
    (-1, 1, 0.2), (0, 1, 0.2), (1, 1, 0.2),
    (-1, 1, 0), (0, 1, 0), (1, 1, 0),
    (-1, 0, 0.2), (0, 0, 0.2), (1, 0, 0.2),
    (-1, 0, 0), (0, 0, 0), (1, 0, 0)
]

# tests/probar_modelo.py (ejecución de la acción)

accion = agente.seleccionar_accion(estado)
obs, reward, terminated, truncated, _ = env.step(
    np.array(accion, dtype=np.float32)
)

# src/agente.py (memoria compacta)

def memorize(self, state, action, reward, next_state, done):
    action_index = self.action_space.index(action)
    self.memoria.append((state, action_index, reward, next_state, done))
```

Notas para defensa (qué enfatizar)

• La discretización es una decisión de ingeniería para aplicar DQN; alternativas para control continuo incluyen SAC/TD3/DDPG o PPO con política continua. • La tabla de acciones (12) define qué maniobras son posibles; es un hiperparámetro estructural del

agente.