PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
IIC-3697 APRENDIZAJE PROFUNDO

TAREA 3: DEEP REINFORCEMENT LEARNING

Fecha Máxima de Entrega: Miércoles 30 de Junio, 23:59

Objetivo

Estimad@s, en esta tarea tendrán la oportunidad de poner en práctica sus conocimientos sobre aprendizaje reforzado. En particular, implementarán el algoritmo DQN [1] que vimos en clases. Tal como discutimos, entre su amplio abanico de aplicaciones, este algoritmo permite aprender políticas de acción para juegos de Atari. En esta tarea nos centraremos en el juego Enduro.

Para el desarrollo de esta tarea nuevamente utilizarán la plataforma Google Colaboratory. Adicionalmente, utilizarán gym de OpenAI. Este toolkit nos permitirá implementar el ambiente del juego *Enduro-V0*.

```
import gym
env = gym.make('Enduro-v0')
```

1. Enduro-v0

Enduro consiste en maniobrar un auto de carreras en el *National Enduro*, una carrera de resistencia de larga distancia. El objetivo de la carrera es adelantar un cierto número de autos cada día. Hacerlo permitirá que el jugador continúe compitiendo durante el día siguiente. El conductor debe evitar a otros corredores y pasar 200 autos el primer día y 300 autos cada día siguiente.

A medida que pasa el tiempo del juego, la visibilidad también cambia. Por ejemplo, cuando es de noche en el juego, el jugador solo puede ver las luces traseras de los autos que se aproximan. A medida que avanzan los días, los autos también serán más difíciles de evitar. El clima y la hora del día son factores que influyen en la forma de jugar. Durante el día, el jugador puede conducir a través de un parche helado en la carretera que limitará el control del vehículo, o un parche de niebla, que puede reducir la visibilidad.

En el paquete de *gym*, se otorga una recompensa de +1 por cada auto adelantado y -1 por cada auto que sobrepasa al agente. Sin embargo, la recompensa neta no puede caer por debajo de 0. Hay 9 acciones disponibles, las 8 direcciones de la palanca de *Atari* y no hacer nada. En la figura 1 se muestra la interfaz del juego.

Figura 1: Interfaz de juego *Enduro-V0* de *OpenAI gym*

1.1. Estados y pre-procesamiento

En términos del estado del juego, la interface de OpenAI Gym retorna imágenes de 210x160x3, donde 210x160 indica la resolución espacial de cada cuadro, y 3 indica las componentes de color RGB. Cada pixel es representado por valores en el rango [0,255] (uint8). Para la implementación de esta tarea deben realizar una serie de procesamientos a sus estados antes de ingresarlos a la red neuronal:

1. Resolución: para agilizar el aprendizaje, en su tarea desarrollarán código para bajar la resolución de cada cuadro a 84x84x1, es decir, operarán sobre imágenes en niveles de gris con un tamaño de 84x84 pixels. Para esto pueden usar diversas librerías de *Python*, tales como *Skimage, PIL, y OpenCV-Python*. A modo de ejemplo, el siguiente código muestra como realizar el pre-pocesamiento de imágenes usando OpenCV:

```
    import cv2
    def _process_frame84(frame):
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        frame = frame[:155,10:]
        x_t = cv2.resize(frame, (84, 84),  interpolation=cv2.INTER_AREA)
        return x_t.astype(np.uint8)
```

   La segunda línea de la función anterior permite cortar cada cuadro para que sólo quede la interfaz que incluye la pista de carreras, eliminando la tabla con el puntaje.

2. Salto de cuadros: cada vez que se decida una acción, se debe realizar esa misma acción durante 4 pasos seguidos. Esto reduce la frecuencia con la que se toman decisiones, sin impactar el rendimiento del algoritmo, pero acelerando el tiempo de entrenamiento.

3. Historia de cuadros (*frame history*): El input de la red no sólo considera el estado actual del ambiente de juego, sino que también los estados de pasos anteriores. Debido a esto, cada estado quedará definido por el estado actual más 3 estados anteriores. De esta manera, el estado que ingresará a la red queda de tamaño (84,84,4).

Una manera simple de poder implementar los pasos anteriores es con la ayuda de los *wrappers* de *gym*. Los *wrappers* permiten modificar el ambiente de juego original e incorporar el pre-procesamiento y otros tipos de características.

Para incluir la primera parte del pre-procesamiento pueden crear un *wrapper* personalizado de la siguiente manera:

```
class ProcessFrame84(gym.Wrapper):
    def __init__(self, env=None):
        super(ProcessFrame84, self).__init__(env)
        self.observation_space = spaces.Box(low=0, high=255, shape=(84, 84))

    def step(self, action):
        obs, reward, done, info = self.env.step(action)
        return _process_frame84(obs), reward, done, info

    def reset(self):
        return _process_frame84(self.env.reset())
```

Para la parte 2 del pre-procesamiento pueden simplemente incluir el argumento *frameskip* al crear su ambiente de juego:

```
env = gym.make('Enduro-v0',frameskip=4)
```

Para la última parte del pre-procesamiento pueden importar el *wrapper* de gym *FrameStack*.

Juntando todo, el pre-procesamiento al ambiente de juego:

```
from gym.wrappers import FrameStack

env = gym.make('Enduro-v0',frameskip=4)
env = ProcessFrame84(env)
env = FrameStack(env,4)
```

Un ejemplo de estado procesado aparece en la figura 2:

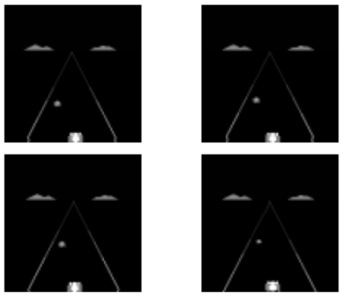

Figura 2: Estado de juego procesado

Como se mencionó en clases, para el caso de espacios continuos de estado, como es el caso de los juegos de Atari, no es posible mantener una tabla de la función $Q(s,a)$ para todas las combinaciones estado-acción. En nuestro caso, usaremos una red convolucional para estimar el valor de la función $Q(s,a)$, utilizando una variante del modelo *Deep Q-learning* (DQN) [1].

2. Dueling DQN

Para implementar DQN tendrán que resolver las ecuaciones que estiman la función $Q(s,a)$.

$$Q(s,a) = r(s,a) + \gamma V^*(s',a') \quad (1)$$

$$Q(s,a) = \mathbb{E}_{s' \sim env}\left[r(s,a) + \gamma \underset{a'}{\text{argmax}} Q(s',a')\right] \quad (2)$$

En esta tarea utilizarán una pequeña variante de DQN llamada *Dueling* DQN [2]. En [2] definen la **ventaja** de una acción a en un cierto estado s como:

$$A(s,a) = Q(s,a) - V(s) \quad (3)$$

Dueling DQN permite estimar el valor de $Q(s,a)$ a partir de la estimación de $V(s,a)$ y $A(s,a)$ con la siguiente arquitectura de red.

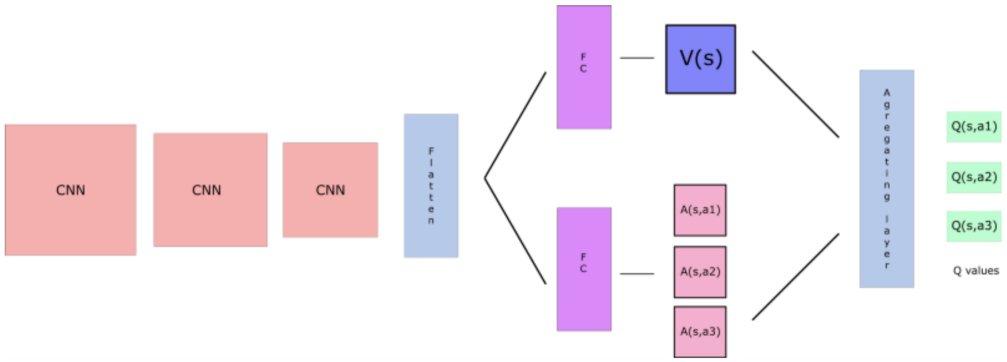

Figura 3: Arquitectura de la red *Dueling* DQN

La arquitectura convolucional inicial es la misma que DQN, luego esta se divide en dos ramas con capas *fully-connected*. La primera permite estimar el valor $V(s)$ del estado de entrada y la segunda la ventaja $A(s,a)$ para cada acción del estado de entrada. El valor $Q(s,a)$ se podría calcular de la ecuación (3), sin embargo, como estamos estimando tanto $V(s)$ como $A(s,a)$ con la red neuronal, la solución para $Q(s,a)$ se indetermina. Para solucionar este problema, los autores de [2] usaron la siguiente ecuación para estimar $Q(s,a)$

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a) \right) \tag{4}$$

Lo que fuerza al valor de $Q(s,a)$ para la acción maximizadora a ser igual a $V(s)$, resolviendo el problema indeterminado.

El siguiente código muestra una posible estructura que pueden utilizar como base para construir su modelo con *Pytorch*.

```
import torch
import torch.nn as nn

class Dueling_DQN(nn.Module):
    def __init__(self, in_channels, num_actions):
        super(Dueling_DQN, self).__init__()
        self.num_actions = num_actions

        self.conv1 = nn.Conv2d(in_channels=in_channels, out_channels=32,
            kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4,
            stride=2)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,
            stride=1)

        self.fc1_adv = nn.Linear(in_features=7*7*64, out_features=512)
        self.fc1_val = nn.Linear(in_features=7*7*64, out_features=512)

        self.fc2_adv = nn.Linear(in_features=512, out_features=num_actions)
        self.fc2_val = nn.Linear(in_features=512, out_features=1)

        self.relu = nn.ReLU()

    def forward(self, x):
        # TODO: forward pass red convolucional

        # TODO: forward pass rama fully-connected para V(s)
```

```
        # TODO: forward pass rama fully-connected para A(s,a)

        # TODO: Calculo de Q(s,a) a partir de ecuacion (4)

        return Q_values
```

Esta nueva arquitectura lo único que cambia es la manera de estimar $Q(s, a)$. El resto de la implementación de DQN que vieron en clases se mantiene igual. Para ello, deben utilizar las siguientes estrategias:

1. Uso de un buffer de memoria para almacenar la experiencia adquirida durante la ejecución del juego (*Experience Replay*). Utilicen un buffer de 500 mil pasos (*steps*). En el caso que tengan problemas de memoria pueden utilizar un buffer con menos pasos, teniendo en cuenta que puede afectar su rendimiento final[1]. Adicionalmente, siguiendo la implementación de DQN y tal como el código que vimos en clases para el juego Flappy Bird, durante los primero 50.000 steps seleccionen acciones bajo una política uniforme y sin realizar entrenamiento, sólo llenar el buffer de memoria.

2. Uso de una red neuronal independiente para estimar la función $\hat{Q}(s, a)$, la cual es considerada como función objetivo durante el entrenamiento, es decir, proporciona los rótulos (labels). Tal como en DQN, la red que estima $\hat{Q}(s, a)$ es actualizada cada C pasos utilizando el valor de los pesos de la red que estima $Q(s, a)$.

3. Uso de estrategia de exploración tipo ϵ-*greedy*. La estrategia considerada por DQN es disminuir ϵ desde 1 a 0.1 en forma lineal durante el primer millón de pasos (steps), y luego mantener el valor constante en 0.1.

En la implementación utilice los mismos hiperparámetros de [1][2].

PARTE 1: Implementación (50 % nota)

Actividad 1

Realice el pre-procesamiento indicado a los estados del juego. ¿Por qué usamos los últimos 4 pasos como input de la red para jugar *Enduro*?

Actividad 2

Implemente el algoritmo de DQN utilizando la arquitectura de red *Dueling* DQN considerando las 3 estrategias mencionadas.

PARTE 2: Detalles de implementación (20 % nota)

Actividad 3

¿Cuál es el beneficio de usar la arquitectura *Dueling* DQN comparado con la arquitectura original de DQN para estimar el valor de $Q(s, a)$?

¿Cómo se traduce este beneficio en la experiencia de juego de Enduro?

Actividad 4

¿Cuál es la ventaja de utilizar un buffer de memoria para los estados del juego? ¿Por qué la estrategia de exploración tipo ϵ-*greedy* comienza con un valor grande que va disminuyendo?

Actividad 5

[1] Consideren guardar sus estados en formato *uint8* para ahorrar memoria. Si utilizaron el *wrapper FrameStack* para el pre-procesamiento revisen la documentación de *LazyFrames*, los cuales también permiten un ahorro de memoria significativo.

[2] También los pueden encontrar en las diapositivas de las clases

¿Cuántos parámetros entrenables tiene el modelo utilizado? ¿Cuántos parámetros de diferencia tiene el modelo de *Dueling* DQN en comparación con el modelo de DQN original?

Actividad 6

Indique todos los hiperparámetros y decisiones relevantes utilizadas para su implementación, tales como tasa de aprendizaje, método de optimización, tamaño de batch, y cualquier otro hiperparámetro utilizado.

PARTE 3: Resultados (30 % nota)

Actividad 7

Realice un gráfico que muestre la evolución del aprendizaje a través del entrenamiento. Específicamente, construya un gráfico que muestre el valor del *score* promedio por episodio en función de las épocas de entrenamiento. Para construir este gráfico considere que una época está dada por el procesamiento de 250 mil pasos de entrenamiento. Para el cálculo del score promedio considere los últimos 100 episodios[3].

3. Consideraciones y formato de Entrega

Si bien el código para implementar su tarea es relativamente simple, no más de 50 líneas, el entrenamiento es bastante intenso. Para poder lograr un buen jugador de *Enduro*, el proceso de entrenamiento debería ser de alrededor de 10 horas, así que no deje el desarrollo de la tarea para último momento.

La tarea debe ser entregada via SIDING en un cuestionario que se habilitará oportunamente. Se debe desarrollar la tarea utilizando un Jupyter Notebook con todas las celdas ejecutadas, es decir, no se debe borrar el resultado de las celdas antes de entregar. Si las celdas se encuentran vacías, se asumirá que la celda no fue ejecutada. Es importante que todas las actividades tengan respuestas explícitas, es decir, no basta con el output de una celda para responder.

El miércoles 16 de junio a las 17:00 hrs se realizará una ayudantía de la tarea. El link se avisará oportunamente. Adicionalmente, durante el desarrollo de la tarea se habilitará un foro para consultas.

Referencias

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533.

[2] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., Freitas, N. (2016, June). Dueling network architectures for deep reinforcement learning. In International conference on machine learning (pp. 1995-2003). PMLR.

[3] Es posible que durante las primeras épocas de entrenamiento, donde la tasa de exploración es alta, el score promedio sea 0 o cercano a 0, pero a medida que la exploración es menor y alcanza su valor mínimo de 0.1 el *score* debería aumentar