DEEP REINFORCEMENT LEARNING

AUTORES Y NIUBs:

Oriol Saguillo González NIUB 20150502

Jordi Romero Suarez NIUB 20081633

Inteligencia Artificial

Grup MB

ÍNDICE

Introducción

El *Deep Reinforcement Learning* es un tema muy hablado en la actualidad gracias a todas las últimas aplicaciones prácticas que se han realizado estos últimos años. En este proyecto, nos centraremos en realizar un breve análisis de este subcampo de la IA, haciendo referencia a las grandes ramas que lo componen. Con ello, llegaremos a realizar una aplicación práctica con el fin de poder visualizar cómo se comporta el *Deep Reinforcement Learning* en un videojuego de Atari, como es el BreakOut.

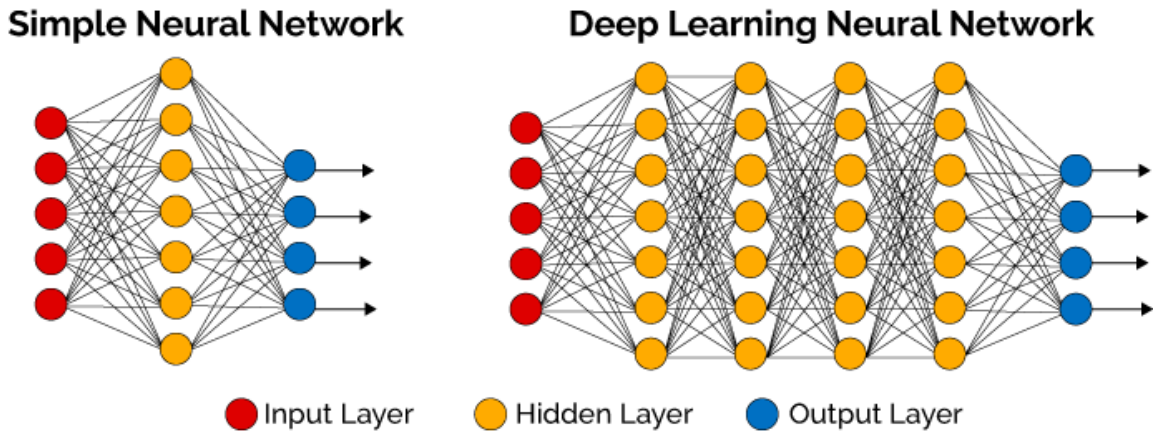
¿Qué es el "Deep Learning"?

Para entender lo que significa el estudio del "*Deep Learning*", a continuación, se introducirá un resumen de su historia.

Las *Artificial Neural Networks* (ANN) son un modelo de *Machine Learning* inspiradas en las redes neuronales biológicas que podemos encontrar en nuestro cerebro. Estas fueron introducidas en 1943 por el neuropsicólogo Warren MCulloch y el matemático Walter Piits a través de un boletín titulado *"A Logical Calculus of Ideas Immanent in Nervous Activity"*. El documento describe un modelo computacional simplificado para entender el comportamiento de las neuronas, es decir, muestra cómo dichas neuronas podrían trabajar simbióticamente en cerebros animales para llevar a cabo cálculos más complejos mediante la lógica proposicional.

Hoy en día se denomina a una red como *Deep Neural Network* (DNN) en el momento en que una *Artificial Neural Network* contiene una pila profunda de capas ocultas.

En el campo del *Deep Learning* se estudian las DNN y, en palabras más generales, los modelos que contienen un volumen elevado de cálculos.



En el estudio de las redes neuronales, estas pueden presentar diversos tipos de topologías arquitecturales, así como formas de aprendizaje diferentes. Para este informe, se explicará en detalle cómo funciona el *Deep Q-Learning*, una de estas posibles formas de aprendizaje, donde su implementación consistirá en una *Deep Q-Network.*

¿Qué es el "Deep Reinforcement Learning"?

El "*Deep Reinforcement Learning*" o "Aprendizaje por Refuerzo Profundo" se define como una categoría de aprendizaje o Inteligencia Artificial similar al comportamiento humano, es decir, basándose en la experiencia. Para ello, cuando se consigue alcanzar un propósito, se "siente" recompensado (de aquí el Refuerzo o *Reinforcement*). Por tanto, el método de mejora es el de "prueba y error", característica que los hace idóneos para entornos dinámicos que se encuentran en constante cambio y evolución. La característica *Deep* o Profunda del aprendizaje por refuerzo hace referencia a las múltiples capas de redes neuronales artificiales, semejantes a las humanas.

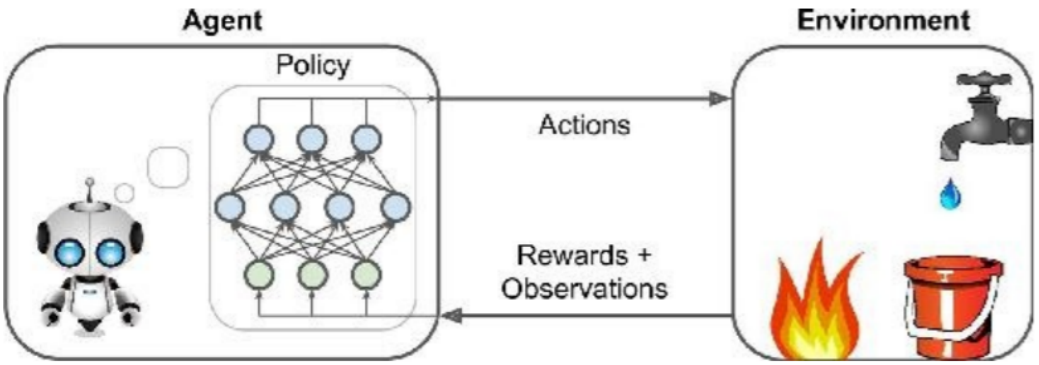
En cuanto a los métodos de *Deep Reinforcement Learning*, estos se obtienen cuando utilizamos redes neuronales profundas para aproximar los siguientes componentes del aprendizaje por refuerzo:

1. Función de valor: $\widehat{v}(s\,;\,\Theta)\ o\ \widehat{q}(s,a\,;\,\Theta)$
2. Política: $\pi\,(a \mid s\,;\,\Theta)$
3. Modelo: función de transición de estado y función de recompensa.

Por último, si nos centramos en las ventajas o beneficios que nos ofrece el *Deep Q-Learning* respeto al *Q-Learning*, descritas por DeepMind, podríamos destacar las siguientes:

1. Cada paso que el agente realiza dentro de su experiencia de aprendizaje es, con alta probabilidad, utilizado en la actualización del peso de las neuronas. Como consecuencia, esto nos permite obtener una mejor eficiencia a la hora de procesar datos recibidos.
2. Aprender directamente de muestras consecutivas (acción tras acción) es ineficiente, debido a las fuertes correlaciones entre las muestras o estados. Además , aleatorizar dichas muestras rompería estas correlaciones y, por ende, se reduciría el *Variance* en el *Update* de los pesos .
3. Sobre el aprendizaje de política, los parámetros actuales determinan la siguiente muestra de datos con la que se entrenarán dichos parámetros, es decir, si la acción maximizadora es "moverse a un bando", las muestras de entrenamiento estarán dominadas por las de dicho bando (independientemente de cual sea). Si el maximizador de la acción cambia de dirección, el patrón de entrenamiento también cambiará.

Así, pues, es fácil observar cómo se forman bucles de retroalimentación no deseados y cómo los parámetros podrían quedarse atascados en un mínimo local deficiente, pudiendo incluso resultar en una divergencia catastrófica.



Deep Q-Learning

Deep Q-Learning es un algoritmo introducido por DeepMind en el documento "*Playing Atari with Deep Reinforcement Learning"* en 2013. En este algoritmo se implementa una nueva técnica en el aprendizaje automático conocida como *Experience Replay.*

Esta técnica se propuso ya que el DNN se adapta fácilmente a los resultados y, una vez estos ya están preajustados, es difícil producir diversas experiencias.

Para resolverlo, el sistema almacena los datos descubiertos para $e_t = (s_t, a_t, r_t, s_{t+1})$, en un conjunto de datos D, donde $D = (e_1, ..., e_N)$, agrupando todos estos episodios en un *replay memory*. A la hora de realizar el aprendizaje, se separa lógicamente de la adquisición de experiencia y se basa en tomar muestras aleatorias de D.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ ***do***
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Este sería el pseudocódigo sobre el *Deep Q-learning*. El *Input* serán los pixeles del juego, es decir, el entorno del agente. Antes de introducirse al algoritmo, la dimensión de la imagen será tratada con la función Φ, con el objetivo de eliminar las partes innecesarias de la imagen que no aportan nada al aprendizaje. El *output* corresponde a los *Q-Values* predichos por cada posible acción del individuo para el estado de entrada.

Se tienen en consideración secuencias de acciones y observaciones, $s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$, y aprenden estrategias de juego que dependen de dichas secuencias.

Por cada episodio, primero se procesa la imagen con Φ y, una vez se encuentra dentro del propio episodio, al igual que el *Q-Learning*, dada la probabilidad de ε se elige entre una acción aleatoria, con el objetivo de explorar; o la mejor acción, obtenida mediante el mejor valor generado por la función de valor, en este caso una función de Q.

Una vez decidida la acción que queremos que el agente ejecute, se observa tanto su recompensa como el nuevo escenario o situación que se le presenta al agente.

Se guarda como el estado emulado S_{t+1}, el estado de donde proviene, la acción que se realizó y la nueva imagen, escenario creado. Una vez asignado, se le aplica la función Φ y se guarda esta transición en el *Replay Memory D.*

En este punto, se realiza un Mini-Batch, que consiste en coger una parte del conjunto de datos y hacer un gradiente descendente. Para ello se selecciona aleatoriamente un subconjunto de D.

Dentro del subconjunto generado elegimos como la recompensa a analizar, $y_i = R_j$ for terminal state s_j o $R_{j+1} + \gamma \max_{a'} Q(s_{j+1},a')$ for non-terminal state s_t. En caso de no ser terminal, cogerá la recompensa de la actual secuencia de estados y acciones, junto con el siguiente *Q-Value* que sea el más grande posible, cogiendo este con un descuento γ, debido a que no es una recompensa segura.

Para actualizar el *Q-Network,* se hará un descenso del gradiente sobre el error cuadrático medio entre los *Q-Values* actuales del agente y los *Q-Values* del objetivo. Una vez se ha realizado este descenso, se adaptarán los pesos de la *Network*.

Deep Q-Learning Variants

Dentro del *Deep Q-Learning,* se pueden encontrar varias modificaciones del *Deep Q-Network* (DQN).

Fixed Q-Value Target

En el algoritmo básico de *Q-Learning*, se utiliza el modelo para hacer predicciones y decidir cuál es el objetivo. Este intercambio de información puede hacer que la *Network* no sea estable, es decir, puede diverger, oscilar, congelarse, etc. Para resolver este grave problema, en 2013 el artículo de los investigadores de DeepMind utilizaron dos DQN, en vez de una: el primer modelo se encarga de aprender paso a paso y mover al agente y, el otro, es el modelo que define los objetivos.

Double DQN

En un artículo de 2015, los investigadores de DeepMind retocaron su algoritmo DQN, aumentando su actuación y estabilizando el entrenamiento. Esta actualización estaba basada en la observación de si la Network, la cual era la responsable de definir los objetivos, era propensa a sobreestimar los *Q-Values* o no. De hecho, supongamos que todas las acciones son idénticas, pero, como estas son aproximaciones, algunas podrían ser más grandes que otras de pura casualidad. Entonces, la *Network* encargada de los objetivos siempre seleccionará el *Q-Value* más grande y, muy probablemente, acabará sobreestimando el verdadero *Q-Value*. Para resolver este malentendido, DeepMind propuso utilizar el modelo de predicciones cuando queramos seleccionar las mejores acciones para los siguientes estados y el modelo de objetivos, cuando queramos estimar los *Q-Values* de estas acciones.

Aún así se siguen intercalando los dos procesos: actuar y aprender, porque la mejora de la política conducirá a comportamientos diferentes que deberían explorar acciones más cercanas a las óptimas y así aprender de ellas.

Prioritized Experience Replay

Antes de empezar la explicación del *Prioritized Experience Replay,* hay que hacer mención al *Temporal Difference Error* (TD Error), ya que es un término esencial para definir cómo coger las mejores experiencias.

El *Temporal Difference Learning* es un agente que aprende de un entorno a través de episodios sin conocimiento previo de dicho entorno.

Por otro lado, el *Temporal Difference Error* es una magnitud que nos sirve para medir qué tan bueno es un estado independientemente de las acciones previas o futuras.

Las experiencias son consideradas importantes si tienen posibilidades de conseguir procesos rápidos de aprendizaje. Una manera de estimar su importancia es medir la magnitud del TD Error. Como las muestras tenderán a darle demasiada importancia a la experiencia, deberemos compensar esta tendencia durante el entrenamiento bajando los pesos de las experiencias en acorde a su importancia. En caso de no llevar hacerlo, el modelo estará con un *overfit* por la importancia de las experiencias seleccionadas y solo estará preparado para ellas.

Dueling DQN

Para entender cómo funciona, primero se debe notar que el *Q-Value* de cada par estado-acción (s, a) se puede expresar como:

$$Q(s, a) = V(s) + A(s, a)$$, donde :

- V(s) es el valor del estado s.
- A(s, a) es la ventaja de tomar la acción "a" en el estado s.

Esta última, estará comparada a todas las acciones posibles en el estado s.
En el Dueling DQN, el modelo estima tanto el valor del estado como la ventaja de cada posible acción. Como la mejor acción debería tener una ventaja de 0, el modelo sustrae la máxima ventaja predecida de todas las predecidas.

Aplicaciones actuales del Deep Reinforcement Learning

Como todas las nuevas tecnologías, las aplicaciones que se pueden llegar a realizar son diversas. En este caso, el *Deep Reinforcement Learning* no se queda atrás, ya que tiene un amplio campo de aplicaciones.

Entrando en uno de estos campos, para este proyecto, el más importante es el de los Juegos. Especialmente en el AlphaGo Zero, usa el *Deep Reinforcement Learning* para aprender el juego de Go desde cero por sí mismo y así, después de 40 días de entrenamiento propio, pueda derrotar al número uno del propio juego, Ke Jie.

Por otro lado, más concretamente dentro de los videojuegos,donde nosotros haremos hincapié serán los grandes bancos de prueba para testear esta tecnología. Algunos de estos ejemplos pueden ser desde los más actuales, como por ejemplo los de Starcraft, como los más antiguos, siendo estos los juegos de Atari. En nuestro caso, es la parte práctica que se realizará con el juego BreakOut de Atari.

Otro de los campos, que es también una de sus áreas clásicas, es la Robótica. A partir del *Deep Reinforcement Learning* se pueden entrenar a robots para que tengan la capacidad de agarrar varios objetos. Con ello, se da pie a la fabricación de productos dentro de una línea de montaje. Para esto, un modelo se entrena inicialmente fuera de la línea y luego se implementa para poder ajustarlo al del robot real.

Finalmente, una aplicación muy sonada estos últimos años es la de la conducción autónoma. Dentro de este campo se encuentran varios aspectos a considerar, como los límites de velocidad, evitar colisiones o zonas por donde no conducir,etc. Gracias al *Deep Reinforcement Learning* se podrían realizar estas y más tareas, como la optimización de la trayectoria, la planificación del movimiento o las rutas óptimas a realizar.

Aplicación Práctica

Para la parte práctica de este proyecto, se decidió realizar una implementación del DQN entrenando un algoritmo del *OpenAI's gym* para el juego Breakout de Atari. El repositorio para descargar se encuentra en Github.

El código que se mostrará a continuación ha sido extraído como parte del curso de práctica "*Deep Reinforcement Learning"* de Coursera y ha sido modificado para el uso de las últimas librerías de TensorFlow. Es aconsejable ejecutar el código en el entorno de Collab del propio Google ya que ofrece gran compatibilidad con las librerías.

A continuación, se realizará un análisis más exhaustivo del código junto con sus librerías con el fin de lograr entenderlas.

Inicialmente, se comentarán los diferentes *imports* que nos encontraremos a lo largo de la implementación, donde la gran parte de estas ya son conocidas tanto en esta asignatura como en otras del mismo curso. A la vez, estudiaremos otras que son nuevas para nosotros.

Estas son las librerías que nos encontraremos:

- **Numpy** → Librería principal que da soporte para la creación de vectores y matrices multidimensionales y, a la vez, da una gran variedad de funciones de alto nivel para poder operar con estas mismas.
- **Pandas** → Librería utilizada para el análisis de datos extraídos y herramientas para su manipulación.
- **Matplotlib** → Librería para generar gráficos a partir de datos obtenidos de la propia librería Numpy.
- **Random** → Librería para generar números pseudoaleatorios.
- **Gym** → Librería para el desarrollo y comparación de algoritmos de *Reinforcement Learning*. Así podemos enseñar a los agentes desde lo más simple, como andar, hasta lo más complejo, como puede ser jugar al ping pong, al pinball o, en nuestro caso, al BreakOut.
- **Cv2** → Librería para la resolución de problemas de visión por computador, en nuestro caso para el reescalado de imágenes.
- **TensorFlow** → Librería principal que se usará como librería encargada del aprendizaje automático.
- **Keras** → Librería de Redes Neuronales que nos permite definir la arquitectura de la red neuronal que queremos construir.

Una vez ya se conocen todas las librerías que se usarán, es momento de entrar en su implementación.

Procesando la imagen del juego

```python
from gym.core import ObservationWrapper
from gym.spaces import Box

# from scipy.misc import imresize
import cv2

class PreprocessAtari(ObservationWrapper):
    def __init__(self, env):
        """A gym wrapper that crops, scales image into the desired shapes and optionally grayscales it."""
        ObservationWrapper.__init__(self,env)

        self.img_size = (84, 84)
        self.observation_space = Box(0.0, 1.0, (self.img_size[0], self.img_size[1], 1))

    def observation(self, img):
        """what happens to each observation"""

        # crop image (top and bottom, top from 34, bottom remove last 16)
        img = img[34:-16, :, :]

        # resize image
        img = cv2.resize(img, self.img_size)

        img = img.mean(-1,keepdims=True)

        img = img.astype('float32') / 255.

        return img
```

Nada más empezar, nos encontramos con la primera clase de todo el código, que será la encargada de reescalar la imagen original del juego de Atari. Esto se debe a que dicha imagen, inicialmente, era muy grande y para el aprendizaje no es necesario tal nivel de detalle y al final ocuparía demasiado tiempo. Com podemos observar, esta sería nuestra función Φ, mencionada anteriormente en la parte teórica.

Por lo tanto, dentro de esta clase, se crean unos ciertos parámetros en los cuales irán incluidas la imágenes del juego y, posteriormente, se pasarán a una escala de grises y se eliminarán aquellas partes que no son necesarias.

Frame buffer

A continuación se pasará a ver la implementación del *FrameBuffer*. Este será el encargado de asegurar que cada observación contenga suficiente información para encontrar acciones óptimas, ya que el agente únicamente puede procesar una observación a la vez. Para poder realizar esto, el *buffer* guardará las últimas 4 imágenes obtenidas, como consecuencia de que el juego original de Atari se movía a 4fps.

```
from gym.spaces.box import Box
from gym.core import Wrapper

class FrameBuffer(Wrapper):
    def __init__(self, env, n_frames=4, dim_order='tensorflow'):
        """A gym wrapper that reshapes, crops and scales image into the desired shapes"""
        super(FrameBuffer, self).__init__(env)
        self.dim_order = dim_order
        if dim_order == 'tensorflow':
            height, width, n_channels = env.observation_space.shape
            """Multiply channels dimension by number of frames"""
            obs_shape = [height, width, n_channels * n_frames]
        else:
            raise ValueError('dim_order should be "tensorflow" or "pytorch", got {}'.format(dim_order))
        self.observation_space = Box(0.0, 1.0, obs_shape)
        self.framebuffer = np.zeros(obs_shape, 'float32')

    def reset(self):
        """resets breakout, returns initial frames"""
        self.framebuffer = np.zeros_like(self.framebuffer)
        self.update_buffer(self.env.reset())
        return self.framebuffer

    def step(self, action):
        """plays breakout for 1 step, returns frame buffer"""
        new_img, reward, done, info = self.env.step(action)
        self.update_buffer(new_img)
        return self.framebuffer, reward, done, info

    def update_buffer(self, img):
        if self.dim_order == 'tensorflow':
            offset = self.env.observation_space.shape[-1]
            axis = -1
            cropped_framebuffer = self.framebuffer[:,:,:-offset]
        self.framebuffer = np.concatenate([img, cropped_framebuffer], axis = axis)
```

Dentro de la clase *FrameBuffer* podemos encontrar diferentes funciones:

- En la función **inicial** (init), se crea un *Wrapper* para así poder reescalarla dentro de las medidas que uno necesite.
- La función **reset**, tal y como su nombre indica, es la responsable de resetear el juego para volver a sus *frames* iniciales.
- La función **step** se encargará de jugar un único paso y así poder retornar su *frame buffer* para estudiarlo.
- A la función **update_buffer** se le pasará una imagen por parámetro, que es la obtenida por la función "step" y, de este modo, actualizaremos el *buffer* con esta nueva imagen obtenida.

Una vez ya se ha definido la clase del *FrameBuffer* con todas sus funciones, se pasa a crear una última función responsable de crear el entorno (**make_env**) a partir de las clases, funciones anteriores y el propio juego.

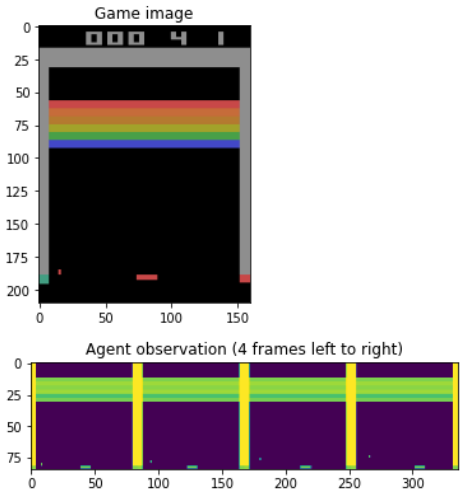
```
def make_env():
    env = gym.make("BreakoutDeterministic-v4")
    env = PreprocessAtari(env)
    env = FrameBuffer(env, n_frames=4, dim_order='tensorflow')
    return env

#Instatntiate gym Atari-Breakout environment
env = make_env()
env.reset()
n_actions = env.action_space.n
state_dim = env.observation_space.shape
```

Después de crear el entorno, realizamos unos *prints* para observar cómo queda dicha imagen del juego junto con 4 frames del *Agent Observation*.

```
# review Atari image, and actual observation of the Agent after processing
for _ in range(50):
    obs, _, _, _ = env.step(env.action_space.sample())


plt.title("Game image")
plt.imshow(env.render("rgb_array"))
plt.show()
plt.title("Agent observation (4 frames left to right)")
plt.imshow(obs.transpose([0,2,1]).reshape([state_dim[0],-1]));
```



Construyendo una red

Ahora que ya se tiene el *Frame Buffer* creado, se pasará a crear una Red Neuronal que pueda mapear las imágenes para indicar los *Q-Values*. Dicha red será llamada a cada paso que realice el agente.

Inicialmente, se importan las ya mencionadas librerías principales para el *Deep Learning*.

```
#Import TensorFlow and Keras
import tensorflow as tf
import tensorflow.compat.v1 as tf1

import keras
from keras.layers import Conv2D, Dense, Flatten, InputLayer
tf1.disable_eager_execution()
tf1.reset_default_graph()
sess = tf1.InteractiveSession()
```

Una vez ya están estas librerías, se empieza a crear la clase del agente DQN:

```
from keras.layers import Conv2D, Dense, Flatten
class DQNAgent:
    def __init__(self, name, state_shape, n_actions, epsilon=0, reuse=False):
        """A simple DQN agent"""
        with tf1.variable_scope(name, reuse=reuse):

            self.network = keras.models.Sequential()

            # Keras ignores the first dimension in the input_shape, which is the batch size.
            # So just use state_shape for the input shape
            self.network.add(Conv2D(32, (8, 8), strides=4, activation='relu',use_bias=False, input_shape=state_shape,kernel_initializer=tf1.variance_scaling_initializer(scale=2)))
            self.network.add(Conv2D(64, (4, 4), strides=2, activation='relu',use_bias=False,kernel_initializer=tf1.variance_scaling_initializer(scale=2)))
            self.network.add(Conv2D(64, (3, 3), strides=1, activation='relu',use_bias=False,kernel_initializer=tf1.variance_scaling_initializer(scale=2)))
            self.network.add(Conv2D(1024, (7, 7), strides=1, activation='relu',use_bias=False,kernel_initializer=tf1.variance_scaling_initializer(scale=2)))
            self.network.add(Flatten())
            self.network.add(Dense(n_actions, activation='linear',kernel_initializer=tf1.variance_scaling_initializer(scale=2)))

            # prepare a graph for agent step
            self.state_t = tf1.placeholder('float32', [None,] + list(state_shape))
            self.qvalues_t = self.get_symbolic_qvalues(self.state_t)

        self.weights = tf1.get_collection(tf1.GraphKeys.TRAINABLE_VARIABLES, scope=name)
        self.epsilon = epsilon

    def get_symbolic_qvalues(self, state_t):
        """takes agent's observation, returns qvalues. Both are tf Tensors"""
        qvalues = self.network(state_t)


        assert tf.debugging.is_numeric_tensor(qvalues) and qvalues.shape.ndims == 2, \
            "please return 2d tf tensor of qvalues [you got %s]" % repr(qvalues)
        assert int(qvalues.shape[1]) == n_actions

        return qvalues

    def get_qvalues(self, state_t):
        """Same as symbolic step except it operates on numpy arrays"""
        sess = tf1.get_default_session()
        return sess.run(self.qvalues_t, {self.state_t: state_t})

    def sample_actions(self, qvalues):
        """pick actions given qvalues. Uses epsilon-greedy exploration strategy. """
        epsilon = self.epsilon
        batch_size, n_actions = qvalues.shape
        random_actions = np.random.choice(n_actions, size=batch_size)
        best_actions = qvalues.argmax(axis=-1)
        should_explore = np.random.choice([0, 1], batch_size, p = [1-epsilon, epsilon])
        return np.where(should_explore, random_actions, best_actions)
```

- Partimos de hacer el constructor de esta clase, donde se inicializará la red y se montará la arquitectura. A la vez que la inicializamos, se le asignan los pesos y el ε correspondiente para su exploración. Un detalle importante dentro del **init** es que, a la hora de añadir las redes, se usará un tipo de función de activación específica. Esta será la responsable de transformar la entrada ponderada, sumada del nodo en su activación o salida para esa entrada. En este caso, se usará la función **RELU** (la misma que utiliza DeepMind): una función lineal por partes que generará la entrada directamente si es positiva; de lo contrario, generará cero.

El porqué de su uso es que el modelo que la usa es más fácil de entrenar y, a menudo, logra un mejor rendimiento

- Más adelante, se encuentra la función **get_symbolic_qvalues** que es responsable de coger la observación del agente y devolver el *Q-Value* correspondiente a las acciones a tomar de dicha observación.
- La función **get_qvalues** realizará lo mismo que la anterior, pero únicamente operará sobre *arrays* de Numpy
- Finalmente, tenemos la función **sample_actions** que elegirá una acción *random* o la máxima del *Q-Value* que reciba.

Ahora que ya está la clase formalizada, se pasará a crear el agente a partir de la dimensión del estado, el número de acciones y el epsilon.

```
agent = DQNAgent("dqn_agent", state_dim, n_actions, epsilon=0.5)
sess.run(tf1.global_variables_initializer())
```

Por último, se define una función donde evaluaremos el rendimiento del agente dentro de un cierto número de partidas que nosotros queramos. A la vez se puede evaluar el rendimiento. En caso que sea *greedy*, elegirá la acción con el mejor *Q-Value*, donde una vez acabe, retornará una recompensa media de todas las partidas.

```
#Evaluate agents performance, in a number of games
def evaluate(env, agent, n_games=1, greedy=False, t_max=10000):
    """ Plays n_games full games. If greedy, picks actions as argmax(qvalues). Returns mean reward. """
    rewards = []
    s = env.reset()
    for _ in range(n_games):
        reward = 0
        for _ in range(t_max):
            qvalues = agent.get_qvalues([s])
            action = qvalues.argmax(axis=-1)[0] if greedy else agent.sample_actions(qvalues)[0]
            s, r, done, _ = env.step(action)

            reward += r
            if done:
              s = env.reset()
              break


        rewards.append(reward)
    return np.mean(rewards)
```

Experience replay

Como se ha comentado anteriormente en la parte más teórica, esta técnica logra solucionar el problema cuando los resultados ya están sobreajustados a partir del almacenaje de experiencias, incluyendo recompensas y acciones del agente.

Para la implementación de esta técnica, inicialmente se crea la clase *ReplayBuffer,* que se encargará de guardar todas las experiencias que haya: la observación, la acción, la recompensa o la siguiente observación, siempre cogiendo las ηη últimas experiencias, como el DQN original. El *sample* será el encargado de dar un conjunto *random* para que pueda entrenar.

```
class ReplayBuffer(object):
    def __init__(self, size):
        """Create Replay buffer.
        Parameters
        ----------
        size: int
            Max number of transitions to store in the buffer. When the buffer
            overflows the old memories are dropped.
        """
        self._storage = []
        self._maxsize = size
        self._next_idx = 0

    def __len__(self):
        return len(self._storage)

    def add(self, obs_t, action, reward, obs_tp1, done):
        data = (obs_t, action, reward, obs_tp1, done)

        if self._next_idx >= len(self._storage):
            self._storage.append(data)
        else:
            self._storage[self._next_idx] = data
        self._next_idx = (self._next_idx + 1) % self._maxsize

    def _encode_sample(self, idxes):
        obses_t, actions, rewards, obses_tp1, dones = [], [], [], [], []
        for i in idxes:
            data = self._storage[i]
            obs_t, action, reward, obs_tp1, done = data
            obses_t.append(np.array(obs_t, copy=False))
            actions.append(np.array(action, copy=False))
            rewards.append(reward)
            obses_tp1.append(np.array(obs_tp1, copy=False))
            dones.append(done)
        return np.array(obses_t), np.array(actions), np.array(rewards), np.array(obses_tp1), np.array(dones)
```

```
def sample(self, batch_size):
    """Sample a batch of experiences.
    Parameters
    ----------
    batch_size: int
        How many transitions to sample.
    Returns
    -------
    obs_batch: np.array
        batch of observations
    act_batch: np.array
        batch of actions executed given obs_batch
    rew_batch: np.array
        rewards received as results of executing act_batch
    next_obs_batch: np.array
        next set of observations seen after executing act_batch
    done_mask: np.array
        done_mask[i] = 1 if executing act_batch[i] resulted in
        the end of an episode and 0 otherwise.
    """
    idxes = [random.randint(0, len(self._storage) - 1) for _ in range(batch_size)]
    return self._encode_sample(idxes)
```

Para finalizar este apartado del informe, se implementará la función **play_and_record,** que se encargará de jugar las partidas con los *n_steps* que nosotros le digamos y guardará el conjunto de experiencias dentro del *replay buffer*. Así mismo, cuando la partida termine, activará el *flag done* a True para indicar que ha llegado a un estado terminal.

```
def play_and_record(agent, env, exp_replay, n_steps=1):
    """
    Play the game for exactly n steps, record every (s,a,r,s', done) to replay buffer.
    Whenever game ends, add record with done=True and reset the game.
    :returns: return sum of rewards over time

    Note: please do not env.reset() unless env is done.
    It is guaranteed that env has done=False when passed to this function.
    """
    # State at the beginning of rollout
    s = env.framebuffer

    # Play the game for n_steps as per instructions above
    reward = 0.0
    for t in range(n_steps):
        # get agent to pick action given state s
        qvalues = agent.get_qvalues([s])
        action = agent.sample_actions(qvalues)[0]
        next_s, r, done, _ = env.step(action)

        # add to replay buffer
        exp_replay.add(s, action, r, next_s, done)
        reward += r
        if done:
            s = env.reset()
        else:
            s = next_s
    return reward
```

Target networks

Para la implementación final, también se usará el llamado "*Target Network*", siendo este una copia de los pesos de la *Neural Network* que usará los *Q-Values* como referencia. Esta sería la implementación práctica del Double DQN teórico mencionado anteriormente.

La aplicación del *Target Network* creará en un principio otro agente exactamente igual al original:

```
target_network = DQNAgent("target_network", state_dim, n_actions)
```

Luego, dentro de la función **load_weights_into_target_network,** se le asignarán los pesos del *target_network* a los pesos del agente respectivo. Todo esto se realiza con una iteración de los pesos del agente y los pesos del *target_networks* donde ambos se añadirán dentro de un *array* común.

```
def load_weigths_into_target_network(agent, target_network):
    """ assign target_network.weights variables to their respective agent.weights values. """
    assigns = []
    for w_agent, w_target in zip(agent.weights, target_network.weights):
        assigns.append(tf1.assign(w_target, w_agent, validate_shape=True))
    tf1.get_default_session().run(assigns)
```

Métricas del DQN

Una vez ya se ha construido todo lo necesario con sus mejoras, deberemos evaluar de alguna forma el rendimiento del agente. Inicialmente se crean unos *placeHolders*, reserva de memoria.

```
# Create placeholders that will be fed with exp_replay.sample(batch_size)
obs_ph = tf1.placeholder(tf.float32, shape=(None,) + state_dim)
actions_ph = tf1.placeholder(tf.int32, shape=[None])
rewards_ph = tf1.placeholder(tf.float32, shape=[None])
next_obs_ph = tf1.placeholder(tf.float32, shape=(None,) + state_dim)
is_done_ph = tf1.placeholder(tf.float32, shape=[None])

is_not_done = 1 - is_done_ph
gamma = 0.99
```

Una vez inicializados estos *placeHolders,* debemos obtener los *Q-Values* junto con las acciones que la red a computado para decir sus *Q-Values*.

```
current_qvalues = agent.get_symbolic_qvalues(obs_ph)
current_action_qvalues = tf.reduce_sum(tf.one_hot(actions_ph, n_actions) * current_qvalues, axis=1)
```

Ahora que ya se tienen tanto los *Q-Values* actuales como las acciones de estos, pasamos evaluar el rendimiento con la función de coste, junto con el *Q-Reference* según las fórmulas a continuación:

$$L = \frac{1}{N}\sum_{i}[Q_{\theta}(s,a) - Q_{reference}(s,a)]^2$$

$$Q_{reference}(s,a) = r(s,a) + \gamma \cdot max_{a'} Q_{target}(s',a')$$

```
# compute q-values for NEXT states with target network
next_qvalues_target =  target_network.get_symbolic_qvalues(next_obs_ph)

# compute state values by taking max over next_qvalues_target for all actions
next_state_values_target = tf.reduce_max(next_qvalues_target, axis=-1)

# compute Q_reference(s,a) as per formula above.
reference_qvalues = rewards_ph + gamma*next_state_values_target*is_not_done

# Define loss function for sgd.
# cost_function = (current_action_qvalues - reference_qvalues) ** 2
cost_function = tf1.reduce_mean(tf1.losses.huber_loss(labels=reference_qvalues, predictions=current_action_qvalues))

optimizer=tf1.train.AdamOptimizer(1e-5)
train_step = optimizer.minimize(cost_function, var_list=agent.weights)

sess.run(tf1.global_variables_initializer())
```

En esta parte del código, acabamos de definir las últimas variables necesarias para el último paso.

Loop principal

Por último, llegamos al final de la implementación donde se juntará todo lo que se ha ido construyendo y así lograr el entrenamiento del agente.

Inicialmente, se crea un *buffer* y se llena a partir del **play_and_record.** Luego, crearemos una función que se encargue de coger un lote de observaciones del propio *buffer,* que más adelante serán utilizadas para entrenar la red.

```
#Create the buffer and fill it.
exp_replay = ReplayBuffer(70000)
play_and_record(agent, env, exp_replay, n_steps=10000)

# take a sample batch of observations from the buffer
def sample_batch(exp_replay, batch_size):
    obs_batch, act_batch, reward_batch, next_obs_batch, is_done_batch = exp_replay.sample(batch_size)
    return {
        obs_ph:obs_batch, actions_ph:act_batch, rewards_ph:reward_batch,
        next_obs_ph:next_obs_batch, is_done_ph:is_done_batch
    }
```

A continuación, se importan algunas librerías más que nos ayudarán a la hora de visualizar los datos que extraigamos, como pueden ser la librería **tqdm** para visualizar el avance de ejecución o **matplotlib** para plotear los resultados.

```
from tqdm import trange
from IPython.display import clear_output
import matplotlib.pyplot as plt
from pandas import DataFrame
moving_average = lambda x, span, **kw: DataFrame({'x':np.asarray(x)}).x.ewm(span=span, **kw).mean().values
%matplotlib inline

mean_rw_history = []
loss_history = []
```

Teniendo todo listo, es la hora de entrenar al agente. Partimos de $\varepsilon = 1$ para que el agente elija antes la exploración que la mejor opción, ya que de momento no se sabe nada del entorno.

A partir de aquí, se llamará a la función encargada de jugar la partida y guardar los resultados. Posteriormente, se adaptará la red a partir de los resultados de la función de coste, con el objetivo de que el agente juegue mejor a partir de cada partida. Aquí ya se está jugando y entrenando a la red, por lo que se realizará a continuación serán diferentes ajustes a los parámetros del agente, como puede ser la carga de pesos entre el agente y la *target network* y la reducción del ε cada 500 iteraciones.

Finalmente, cada 5000 iteraciones, se guardará el "*mean reward per game*" para que, a cada 500 iteraciones, se "ploteen" diferentes resultados como el mencionado anteriormente o el historial de la función de coste.

```
##Train the agent, configure the starting epsilon to one to encourage exploration
agent.epsilon=1
for i in trange(100000):

    # play
    play_and_record(agent, env, exp_replay, 10)

    # train the network
    _, loss_t = sess.run([train_step, cost_function], sample_batch(exp_replay, batch_size=64))
    loss_history.append(loss_t)

    # adjust agent parameters
    if i % 500 == 0:
        load_weigths_into_target_network(agent, target_network)
        # reduce epsilon in every iteration until it reaches 1%
        agent.epsilon = max(agent.epsilon * 0.999, 0.01)


    if i % 5000 == 0:
      #uncomment to store agent's weights every some iterations
#         agent.network.save_weights('/dqn_model_atari_weights.h5')
        mean_rw_history.append(evaluate(make_env(), agent, n_games=3))

    if i % 500 == 0:
        # plot mean reward per game and loss history
        clear_output(True)
        print("buffer size = %i, epsilon = %.5f" % (len(exp_replay), agent.epsilon))

        plt.subplot(1,2,1)
        plt.title("mean reward per game")
        plt.plot(mean_rw_history)
        plt.grid()

        assert not np.isnan(loss_t)
        plt.figure(figsize=[12, 4])
        plt.subplot(1,2,2)
        plt.title("Loss history (moving average)")
        plt.plot(moving_average(np.array(loss_history), span=100, min_periods=100))
        plt.grid()
        plt.show()
```

Vídeo

Por último, podemos ver un vídeo de un modelo que encontramos cuando buscamos un código guía, con el fin de que nos enseñase la forma de implementar el *Deep Reinforcement Learning* en el juego de BreakOut. En este link del vídeo, observaremos como el modelo juega al BreakOut:

Video BreakOut

Conclusiones

A modo de conclusión, queremos expresar lo que podemos extraer de este proyecto. Uno de los mayores beneficios de la realización de este trabajo es el gran aprendizaje que hemos obtenido del *Deep Reinforcement Learning.* La lectura de información, mayoritariamente artículos científicos, la búsqueda constante de medios audiovisuales que nos muestren su funcionamiento así como los testeos con el propio código nos han servido para entender y ser capaces de explicar todo lo citado en los anteriores apartados.

Otra de las aportaciones que nos ha dado este trabajo en equipo es poder ver una aplicación real y actual del *Deep Reinforcement Learning,* yendo a un nivel superior del explicado en clase. Todo ello lo hemos estudiado pasando del *Reinforcement Learning* al ya citado *Deep Reinforcement Learning*.

En resumen, tanto Jordi como Oriol nos damos por más que satisfechos con el trabajo realizado y, en nuestra opinión, creemos que las pautas indicadas en la Introducción se han llevado a cabo en su totalidad.

Bibliografía y Webgrafía

- Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow, Aurélien Géron (Book)
- Deep Reinforcement Learning: An Overview, Yuxi Li (Paper): https://arxiv.org/pdf/1701.07274.pdf
- Deep-Q learning from Demonstrations, Team of Google DeepMind (Paper)
- Playing Atari with Deep Reinforcement Learning, DeepMind Technologies (Paper)
- Proyecto de referencia: https://github.com/GiannisMitr/DQN-Atari-Breakout
- Introducción a las redes RL y Deep Q: https://www.tensorflow.org/agents/tutorials/0_intro_rl
- 10 Real-Life Applications of Reinforcement Learning: https://neptune.ai/blog/reinforcement-learning-applications
- The Applications of Deep Reinforcement Learning: https://www.getsmarter.com/blog/market-trends/the-applications-of-deep-reinforcement-learning/
- https://www.xataka.com/videojuegos/2020-ano-jugar-4k-a-60-fps-nos-paramos-a-pensarlo-hace-no-pantalla-game-boy-era