Práctica

- Objetivo: Implementar una solución, usando *keras-rl* y basada en el algoritmo de *DQN* visto en clase, para que un agente aprenda una estrategia ganadora en el juego del *Pong*.
- La puntuación de este bloque es de 6 puntos sobre la nota final.

El entorno sobre el que trabajaremos será *PongDeterministic-v0* y el algoritmo que usaremos será *DQN*.

Para evaluar cómo lo está haciendo el agente, la recompensa en el *Pong* oscila, aproximadamente, en el rango de valores **[-20, 20]**. La estrategia óptima de un agente estaría alrededor de una media de recompensa de 20.

- **NOTA IMPORTANTE**: Si el agente no llegara a aprender una estrategia ganadora, responder sobre la mejor estrategia obtenida.

· Librerias

In [1]:

```
# Librerias
import numpy as np
import gym

from PIL import Image

from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten, Convolution2D, Permute, BatchNormaliza
from keras.layers.advanced_activations import LeakyReLU
from keras.optimizers import Adam, RMSprop, SGD

from rl.agents.dqn import DQNAgent
from rl.policy import LinearAnnealedPolicy, EpsGreedyQPolicy,BoltzmannQPolicy
from rl.memory import SequentialMemory
from rl.callbacks import FileLogger, ModelIntervalCheckpoint, Callback, TestLogger
from rl.core import Processor

import matplotlib.pyplot as plt
from IPython import display
%matplotlib inline

# seaborn を入れてない人は以下をコメントアウト
import seaborn as sns
sns.set_style('darkgrid')
```

```
Using TensorFlow backend.
```

· Entorno

In [2]:

```
# Environment
ENV_NAME = 'PongDeterministic-v0'

# Get the environment and extract the number of actions.
env = gym.make(ENV_NAME)
nb_actions = env.action_space.n

# random seed
np.random.seed(123)
env.seed(123)
```

Out[2]:

```
[123, 151010689]
```

· Preprocesamiento de las observaciones (imágenes) y Asignación Recompensas

In [3]:

```
# Define the input shape to resize the screen
INPUT_SHAPE = (84, 84)
WINDOW_LENGTH = 4

# This processor will be similar to the Atari processor
class PongProcessor(Processor):
    def process_observation(self, observation):
        assert observation.ndim == 3  # (height, width, channel)

        img = Image.fromarray(observation)
        # resize and convert to grayscale
        img = img.resize(INPUT_SHAPE).convert('L')
        processed_observation = np.array(img)

        assert processed_observation.shape == INPUT_SHAPE
        return processed_observation.astype('uint8')  # saves storage in experience memory

    def process_state_batch(self, batch):
        processed_batch = batch.astype('float32') / 255.
        return processed_batch

    def process_reward(self, reward):
        return np.clip(reward, -1., 1.)
```

Modelo de la red neuronal

1) Definir la arquitectura del modelo que se usará en la solución.

In [4]:

```
# Modelo - red convolucional
input_shape = (WINDOW_LENGTH,) + INPUT_SHAPE

model = Sequential()

model.add(Permute((2, 3, 1), input_shape=input_shape))

model.add(Convolution2D(32, (8, 8), strides=(4, 4)))
model.add(Activation('relu'))
model.add(Convolution2D(64, (4, 4), strides=(2, 2)))
model.add(Activation('relu'))
model.add(Convolution2D(64, (3, 3), strides=(1, 1)))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
print(model.summary())
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
permute_1 (Permute)          (None, 84, 84, 4)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 20, 20, 32)        8224
_________________________________________________________________
activation_1 (Activation)    (None, 20, 20, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 9, 9, 64)          32832
_________________________________________________________________
activation_2 (Activation)    (None, 9, 9, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
activation_3 (Activation)    (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1606144
_________________________________________________________________
activation_4 (Activation)    (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 3078
_________________________________________________________________
activation_5 (Activation)    (None, 6)                 0
=================================================================
Total params: 1,687,206
Trainable params: 1,687,206
Non-trainable params: 0
_________________________________________________________________
None
```

Algoritmo DQN

2) Implementación del algoritmo ***DQN*** con ***keras-rl***.

In [5]:

```
# Memory & Processor
memory = SequentialMemory(limit=1000000, window_length=WINDOW_LENGTH)
processor = PongProcessor()
```

In [6]:

```
# Policy
policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=.1, v
```

In [7]:

```
# DQN
dqn = DQNAgent(model=model, nb_actions=nb_actions, policy=policy, memory=memory, processor=
               nb_steps_warmup=10000, gamma=.99, target_model_update=10000, train_interval=
# Optimizer
optimizer = Adam(lr=.0001, epsilon=0.0001)
# Compile
dqn.compile(optimizer, metrics=['mae'])
```

```
WARNING:tensorflow:From C:\Users\Ocin\Anaconda3\lib\site-packages\rl\util.p
y:79: add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.arra
y_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

- Entrenamiento

In [8]:

```
# Callbacks & PlotReward
class PlotReward(Callback):
    def on_train_begin(self, episode, logs={}):
        self.episode_reward = []
        self.fig = plt.figure(0)

    def on_episode_end(self, episode, logs={}):
        self.episode_reward.append(logs['episode_reward'])
        self.show_result()

    def show_result(self):
        display.clear_output(wait=True)
        display.display(plt.gcf())
        plt.clf()
        plt.plot(self.episode_reward, 'r')
        plt.xlabel('Episode')
        plt.ylabel('Total Reward')
        plt.pause(0.001)

callbacks = [PlotReward(), ModelIntervalCheckpoint(filepath='./weight_now.h5f', interval=10
```

In [9]:

```
# Cargar pesos de un modelo entrenado (o parcialmente entrenado)
load_weights = False # True para cargar pesos

if load_weights:
    dqn.model.load_weights('trained_weights/trained_weights_6.h5f')
    print('Pesos cargados')
```

In [10]:

```
# Training
train = False # True para entrenar

if train:
    dqn.fit(env, verbose=2, visualize=True, callbacks=callbacks, nb_steps=10000000)
    dqn.save_weights('weight_now.h5f', overwrite=True)
```

- Test

In [11]:

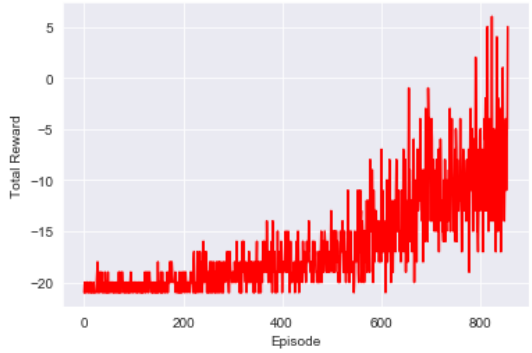
```
# Test
test = False # True para testear
eps_test = 3 # nº episodios test

if test:
    # Cargamos los pesos entrenados
    dqn.model.load_weights('trained_weights/trained_weights_6.h5f')
    # Resultados del test
    results = dqn.test(env, nb_episodes=eps_test, visualize=True, verbose = 2).history['epi
    print('-> Reward mean: ', np.mean(results),'\n')
```

Resultados Entrenamiento

A continuación se muestran las gráficas del entrenamiento que se ha realizado por partes.

· Entrenamiento 1

Policy steps: 100.000, **Target model update:** 10.000, **Train interval:** 20, **Steps warmup:** 50.000



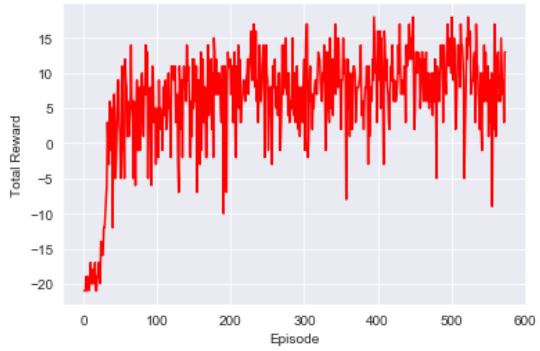

```
 1749308/1750000: episode: 857, duration: 100.706s, episode steps: 4885, steps per second: 49, episode reward: 5.000, mean rewa
rd: 0.001 [-1.000, 1.000], mean action: 2.649 [0.000, 5.000], mean observation: 105.623 [87.000, 236.000], loss: 0.003949, mae:
0.428657, mean_q: 0.506645, mean_eps: 0.100000
done, took 33007.657 seconds
```

> En el 1er entrenamiento a medida que se van completando los episodios la recompensa va augmentando progresivamente hasta llegar a picos de +5.

Nota: entre el entranamiento 1 y 2, hay un entrenamiento en el medio de 1 millón de steps y 280 episodios, llegando a picos de +10. Los parámetros són los mismos que en el 1er entrenamiento con la diferencia que los steps de la policy són 1.000 en vez de 100.000.

· Entrenamiento 2

Policy steps: 1.000, **Target model update:** 1.000, **Train interval:** 4, **Steps warmup:** 1.000

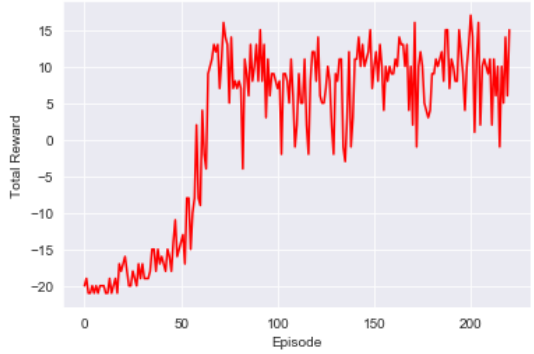



```
 1749948/1750000: episode: 575, duration: 21.845s, episode steps: 2781, steps per second: 127, episode reward: 13.000, mean rew
ard: 0.005 [-1.000, 1.000], mean action: 2.740 [0.000, 5.000], mean observation: 105.543 [87.000, 236.000], loss: 0.003779, ma
e: 0.629542, mean_q: 0.776519, mean_eps: 0.100000
done, took 12272.538 seconds
```

> Al reducir los steps de la policy 1.000 (que son los steps aproximadamente de una partida), el modelo no hace una primera etapa de exploración, por lo tanto el modelo empieza entrenando desde el principio con lo que aprendido en el anterior entrenamiento, hasta llegar a picos de +15.

· Entrenamiento 3

Policy steps: 50.000, **Target model update:** 10.000, **Train interval:** 20, **Steps warmup:** 10.000

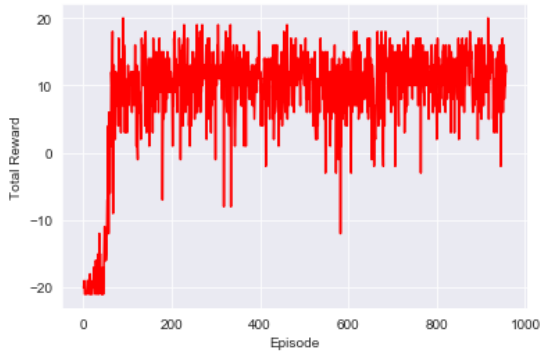



```
  555358/1750000: episode: 221, duration: 10.983s, episode steps: 2447, steps per second: 223, episode reward: 15.000, mean rew
ard: 0.006 [-1.000, 1.000], mean action: 2.049 [0.000, 5.000], mean observation: 105.519 [87.000, 236.000], loss: 0.001604, ma
e: 0.632533, mean_q: 0.744465, mean_eps: 0.100000
done, took 2263.109 seconds
```

En el 3er entrenamiento aumentamos los steps de la policy y el target model update lo ponemos como en el 1er entrenamiento. Con 221 episodios no sobrepasamos el pico anterior de +15.

· Entrenamiento 4

Policy steps: 100.000, **Target model update:** 25.000, **Train interval:** 1.000, **Steps warmup:** 10.000

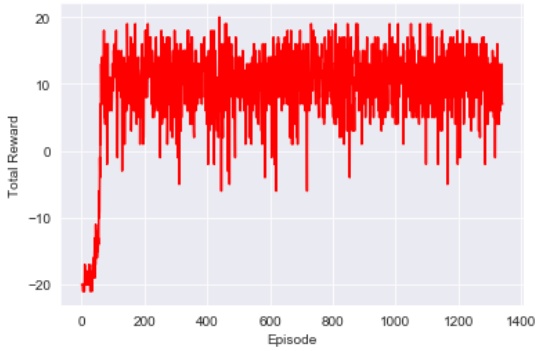



```
 2611272/10000000: episode: 958, duration: 24.419s, episode steps: 2472, steps per second: 101, episode reward: 12.000, mean re
ward: 0.005 [-1.000, 1.000], mean action: 1.659 [0.000, 5.000], mean observation: 105.556 [87.000, 236.000], loss: 0.003955, ma
e: 0.770560, mean_q: 0.951973, mean_eps: 0.100000
done, took 22101.674 seconds
```

Ponemos los steps de la policy como en el 1er entrenamiento, aumentamos los steps de target model update y train interval. Y llegamos a picos de +21.

· Entrenamiento 5

Policy steps: 100.000, **Target model update:** 10.000, **Train interval:** 4, **Steps warmup:** 10.000

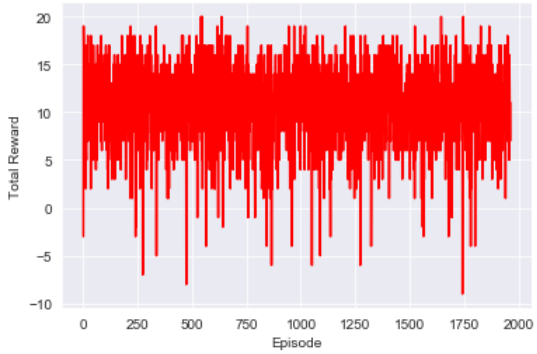



```
 3657014/10000000: episode: 1342, duration: 17.335s, episode steps: 2808, steps per second: 162, episode reward: 7.000, mean re
ward: 0.002 [-1.000, 1.000], mean action: 2.509 [0.000, 5.000], mean observation: 105.589 [87.000, 236.000], loss: 0.003835, ma
e: 0.792339, mean_q: 0.973161, mean_eps: 0.100000
done, took 21158.486 seconds
```

En el entrenamiento 5 reducimos los steps del target model update y el train interval. Se llegan a picos de +21 pero los resultados en general son similares al entrenamiento 4.

· Entrenamiento 6

Policy steps: 1.000, **Target model update:** 10.000, **Train interval:** 100, **Steps warmup:** 10.000

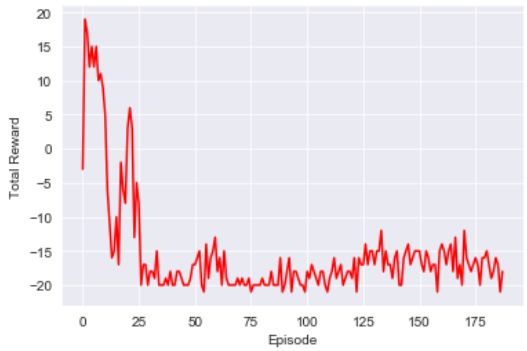



```
 5551150/10000000: episode: 1964, duration: 20.165s, episode steps: 2813, steps per second: 139, episode reward: 11.000, mean r
eward: 0.004 [-1.000, 1.000], mean action: 2.550 [0.000, 5.000], mean observation: 105.578 [87.000, 236.000], loss: 0.001469, m
ae: 0.786783, mean_q: 0.969877, mean_eps: 0.100000
done, took 28930.535 seconds
```

> Reducimos los steps de la policy, para evitar una primera etapa de exploración y augmentamos ligeramente los steps del train interval. Como se puede observar los resultados siguen siendo muy similares a los entrenamientos 4 y 5, sin llegar a estabilizarse a una recompensa fija.

· Entrenamiento 7

Policy steps: 1.000, **Target model update:** 50.000, **Train interval:** 1.000, **Steps warmup:** 10.000, **Gamma:** 0.5




```
  267893/10000000: episode: 188, duration: 4.959s, episode steps: 1264, steps per second: 255, episode reward: -18.000, mean re
ward: -0.014 [-1.000, 1.000], mean action: 3.523 [0.000, 5.000], mean observation: 105.500 [87.000, 236.000], loss: 0.000559, m
ae: 0.034496, mean_q: 0.003444, mean_eps: 0.100000
done, took 1039.646 seconds
```

> Si reducimos el valor de gamma, como se puede apreciar, el modelo en vez de aprender, desaprende y empeora drásticamente los resultados en pocos episodios.

Resultados Test

```
Testing for 3 episodes ...
Episode 1: reward: 20.000, steps: 1980
Episode 2: reward: 19.000, steps: 1992
Episode 3: reward: 19.000, steps: 2142
-> Reward mean:  19.333333333333332
```

```
Testing for 3 episodes ...
Episode 1: reward: 17.000, steps: 2030
Episode 2: reward: 21.000, steps: 1722
Episode 3: reward: 18.000, steps: 2030
-> Reward mean:  18.666666666666668
```

```
Testing for 3 episodes ...
Episode 1: reward: 20.000, steps: 2118
Episode 2: reward: 21.000, steps: 1703
Episode 3: reward: 17.000, steps: 2352
-> Reward mean:  19.333333333333332
```

Conclusiones

No se llega a obtener una recompensa fija de +21 en todas las partidas. Visualizando las partidas de test se puede apreciar que el contrincante (naranja) el primer lanzamiento siempre lo hace de la misma manera (lanza la pelota siempre en el mismo lugar) y nuestra IA termina aprendiendo la técnica de hacer punto devolviendo la pelota una sola vez, después del 1er lanzamiento del naranja. Aunque no siempre lo consigue, como se ha comentado y se aprecia en los resultados.