CENTRO DE CIENCIAS BÁSICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

INTELIGENCIA ARTIFICIAL

3° "A"

PROYECTO/EXAMEN: IMPLEMENTACIÓN DE UNA RED NEURONAL EN VIDEOJUEGOS

Profesor: Miguel Ángel Meza de Luna

Alumnos:

Espinoza Sánchez Joel Alejandro

González Arenas Fernando Francisco

Pardo Tinoco Jonathan David

Pérez Jaime Julio César

Fecha de Entrega: Aguascalientes, Ags.**, 6** de noviembre de 2019

Índice

Antecedentes y Contexto Teórico

“La inteligencia artificial se empeña en buscar algoritmos que permitan a un ordenador tomar decisiones y mejorar sus procesos a partir del éxito o fracaso de estas decisiones” (Russell, 2004). Han sido arduos los intentos por hacer que las máquinas tengan esta habilidad y los algoritmos creados van desde simples ecuaciones hasta estructuras complejas que se asemejan al modelo analítico neurológico del ser humano.

Haciendo énfasis en las analogías que se hacen de las estructuras similares al cerebro humano, los analistas informáticos han desarrollado algoritmos como las redes neuronales que permiten a una máquina procesar información que recibe y transformarla en algún valor que le permita tomar una decisión.

Las Redes Neuronales

Las redes neuronales las define Ponce (2010) como “un modelo computacional vagamente inspirado en el comportamiento observado en su homólogo biológico. Consiste en un conjunto de unidades, llamadas neuronas artificiales, conectadas entre sí para transmitirse señales”. Igualmente él comenta en su obra que la información de entrada atraviesa la red neuronal (donde se somete a diversas operaciones) produciendo unos valores de salida.

Cada neurona está conectada con otras a través de unos enlaces. En estos enlaces el valor de salida de la neurona anterior es multiplicado por un valor de peso. Estos pesos en los enlaces pueden incrementar o inhibir el estado de activación de las neuronas adyacentes. Del mismo modo, a la salida de la neurona, puede existir una función limitadora o umbral, que modifica el valor resultado o impone un límite que se debe sobrepasar antes de propagarse a otra neurona. Esta función se conoce como función de activación.

Su aplicación en videojuegos permite el entrenamiento y desarrollo de agentes inteligentes, así como de algoritmos por lo práctico que resulta, tanto ser un medio no real, como la facilidad de explorar un determinado terreno. Es por eso que la presente investiga las posibilidades de implementación de algoritmos de redes neuronales en entornos de videojuegos a partir del lenguaje de programación de Python. Guiados por el curso de Inteligencia Artificial con Python de Juan Gabriel Gomila (2018), se busca que agentes inteligentes aprendan a resolver distintos laberintos dinámicos y entretenidos, con algoritmos de aprendizaje automático implementados.

Curso de Inteligencia Artificial con Python: Sección 8

Video 1

El video comienza con una introducción en la que Gomila (2019) apoya la teoría usada en secciones pasadas como el Q-Learning. Comenta que particularmente se usó para el problema de la montaña rusa resuelto en la Sección 7. Sin embargo, se mejorará esa versión con el Deep Q-Learning que se utiliza para resolver problemas de control discretos. Estos son problemas donde se discretizan las acciones del actual problema.

En la sección, Gomila comenta que busca mejorar el algoritmo de Q-Learning con métodos muy modernos hasta lograr un Q-Learning muy estable con redes neuronales de aproximación muy útiles. Se tocarán temas como redes neuronales, experiencia de Replay, programación exploratoria, redes neuronales profundas con Pytorch y se comparará el uso con Python TensorBord.

Video 2

Este video es abordado a partir del añadido de la clase Q-Learner al algoritmo de Q-Learning en la sección pasada para entrenar y resolver el problema de la montaña rusa. Esto causó que se discretizara el número de acciones tomando una gama de acciones muy grande a un catálogo muy general de acciones, es decir, que se han restringido los movimientos. Se puede observar en la siguiente imagen esta clase implementada en el código.

```
# %% La clase Q-Learner

class QLearner(object):
    def __init__(self, environment):
        self.obs_shape = environment.observation_space.shape
        self.obs_high = environment.observation_space.high
        self.obs_low = environment.observation_space.low
        self.obs_bins = NUM_DISCRETE_BINS
        self.bin_width = (self.obs_high-self.obs_low)/self.obs_bins

        self.action_shape = environment.action_space.n
        self.Q = np.zeros((self.obs_bins+1, self.obs_bins+1, self.action_shape)) # Matriz de 31 x 31 x 3
        self.alpha = ALPHA
        self.gamma = GAMMA
        self.epsilon = 1.0

    def discretize(self, obs):
        return tuple(((obs-self.obs_low)/self.bin_width).astype(int))

    def get_action(self, obs):
        discrete_obs = self.discretize(obs)
        # Se escoge la acción con base en Epsilon-Greedy
        if self.epsilon > EPSILON_MIN:
            self.epsilon -= EPSILON_DECAY
        if np.random.random() > self.epsilon:
            return np.argmax(self.Q[discrete_obs]) # Con p = 1-epsilon, se toma la mejor posible
        else:
            return np.random.choice([a for a in range(self.action_shape)])# Con p = epsilon, se toma una al azar

    def learn(self, obs, action, reward, next_obs):
        discrete_obs = self.discretize(obs)
        discrete_next_obs = self.discretize(next_obs)
        self.Q[discrete_obs][action] += self.alpha*(reward + self.gamma * np.max(self.Q[discrete_next_obs]) - self.Q[discrete_obs][action])
```

En el video, Gomila (2019) también comenta que las redes neuronales son muy efectivas como funciones universales aproximadoras. Hoy en día, las redes neuronales se encuentran en desarrollo, por lo que no permiten tener una gran aplicación cotidiana, pero la investigación o el mismo sistema del celular, tiene algunos sistemas potenciados por las redes neuronales. A pesar de ser técnicas muy potentes, tienen poco desarrollo pero que de igual manera se busca que en el mismo archivo multimedia, se invite a conocer la implementación de ellas (véase anexo 2.01).

Videos 3 y 4

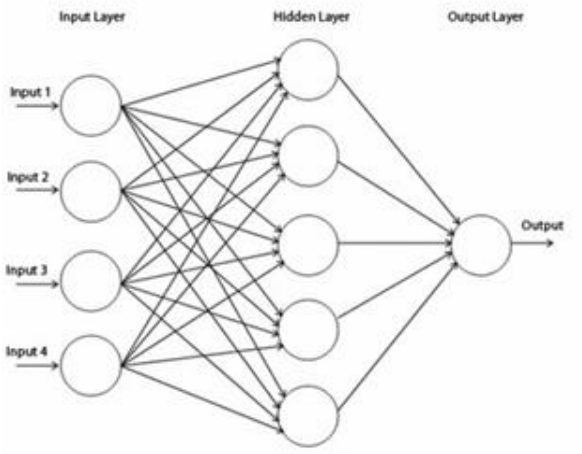
En este video se comenzó a implementar una red neuronal muy sencilla usando el módulo de PyTorch. Esta red neuronal sencilla se trata de un perceptrón, que es una red neuronal de una sola capa. Este fichero de Python se trata de un archivo genérico de tipo librería.

Este archivo se compone de una clase con dos funciones como se puede observar a continuación.

```python
# %% La clase SLP (Single Layer Perceptron)
class SLP(torch.nn.Module):

    def __init__(self, input_shape, output_shape, device = torch.device("cpu")):
        """
        :param input_shape: Longitud o forma de los datos de entrada
        :param output_shape: Longitud o forma de los datos de salida
        :param device: El dispositivo ('cpu' o 'cuda') que la SLP debe utilizar para almacenar los inputs a cada iteracion
        """
        super(SLP, self).__init__()
        self.device = device
        self.input_shape = input_shape[0]
        self.hidden_shape = 40
        self.linear1 = torch.nn.Linear(self.input_shape, self.hidden_shape)
        self.out = torch.nn.Linear(self.hidden_shape, output_shape)


    def forward(self, x):
        x = torch.from_numpy(x).float().to(self.device)
        x = torch.nn.functional.relu(self.linear1(x))##Funcion de activacion RELU
        x = self.out(x)
        return x
```

En este archivo multimedia, Gomila (2019) explica cada proposición escrita en el código, desde la construcción de la función de activación hasta el uso del Álgebra Lineal como el significado de las Transformaciones Lineales para que la neurona comience su trabajo, pues existe como función de activación la función RELU en la función `forward`, misma que hace que la neurona trabaje para valores positivos de las transformaciones lineales de cada entrada (véase anexo 2.02).

Con base en el código realizado en el video anterior, se comenzó ahora a analizar de forma teórica la explicación del perceptrón. Donde “los datos de entrada son mapeados a capas intermedias llamadas capas ocultas, es decir, caja negra. Sí que sabemos que se ponderan los datos a la capa interna.” (Gomila, 2019)



Explica nuevamente conceptos que se trataron en la sección 7 como las funciones de activación que pueden usarse, como la función sigmoide, tangente

hiperbólica o la función ReLu. Finalmente se decide por usar la función ReLu para que despierte al dato de entrada. A su vez, se explica que existe el Algoritmo Optimizador Adam que incrementa el radio de aprendizaje a partir de una combinación lineal convexa con parámetros dados para verificar lo que el agente aprende.

Finalmente, se concluye con la creación de una variante del Q-Learner copiando los cuatro métodos de la clase Q-Learner, pero la esencia de éste es un fichero muy primitivo, pues su complemento definitivo para terminarlo se da en los próximos videos.

Videos 5 y 6

Una vez que se tienen los elementos básicos de código para realizar un nuevo código de Q-Learning, se procedió a tomar el optimizador, por lo que se deberán cambiar muchos elementos del código. Ahora tendremos más elementos como librerías (véase anexo 1.01) y la inicialización de la función Q será distinto (véase anexo 1.02). Posteriormente se utilizará el optimizador Adam de la librería PyTorch.

La continuación de la función se vuelve más compleja puesto que el algoritmo de la ecuación de Bellman se eliminará para tomar elementos de la librería Torch, de modo que la función `learn` se construye de la siguiente forma.

```
    def learn(self, obs, action, reward, next_obs):
        td_target = reward + self.gamma * torch.max(self.Q(next_obs))
        td_error = torch.nn.functional.mse_loss(self.Q(obs)[action], td_target)
        self.Q_optimizer.zero_grad()
        td_error.backward()
        self.Q_optimizer.step()
```

Esta es la red neuronal más sencilla para poder enfocarse en conceptos más profundos de la inteligencia artificial y así conocer en este caso el algoritmo, la propagación hacia atrás y se usa el código implementado en el Q-Learner para continuar con este algoritmo.

Ahora, el pensar por qué se decrementa épsilon es necesario porque se entrelaza este proceso con el de elegir una acción. Para que esto se vea más claro, se agregan estas líneas al código

```
        self.epsilon_max = 1.0
        self.epsilon_min = 0.05
        self.epsilon_decay = LinearDecaySchedule(initial_value = self.epsilon_max,
                                                 final_value = self.epsilon_min,
                                                 max_steps = 0.5 * MAX_NUM_EPISODES * STEPS_PER_EPISODE)
        self.step_num = 0
        self.policy = self.epsilon_greedy_Q
```

Pues esto, en términos de iteración del algoritmo, trata la política de actuación del algoritmo para desglosar la acción y no sólo aplicar una política, sino que, tomando la observación actual, el algoritmo mantendrá su curso y así conocer el paso dado. Así, también se añadirán más librerías y se perfeccionan algunos métodos. Igualmente se tienen todas las librerías necesarias para el funcionamiento óptimo del algoritmo.

Puede observarse que la clase se construye ahora de la siguiente manera.

```python
# %% La clase Swallow Q-Learner o Agente de Q-Learning Ligero
class SwallowQLearner(object):
    def __init__(self, environment, learning_rate = 0.005, gamma = 0.98):
        self.obs_shape = environment.observation_space.shape

        self.action_shape = environment.action_space.n
        self.Q = SLP(self.obs_shape, self.action_shape)
        self.Q_optimizer = torch.optim.Adam(self.Q.parameters(), lr = learning_rate)

        self.gamma = gamma

        self.epsilon_max = 1.0
        self.epsilon_min = 0.05
        self.epsilon_decay = LinearDecaySchedule(initial_value = self.epsilon_max,
                                                 final_value = self.epsilon_min,
                                                 max_steps = 0.5 * MAX_NUM_EPISODES * STEPS_PER_EPISODE)
        self.step_num = 0
        self.policy = self.epsilon_greedy_Q
        #Exploracion paso a paso del algoritmo para mejor entendimiento de epsilon

        self.memory = ExperienceMemory(capacity = int(1e5))
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    def get_action(self, obs):
        return self.policy(obs)

    def epsilon_greedy_Q(self, obs):
        if random.random() < self.epsilon_decay(self.step_num):
            action = random.choice([a for a in range(self.action_shape)])
        else:
            action = np.argmax(self.Q(obs).data.to(torch.device('cpu')).numpy())
        return action
```

En comparación con el Q-Learner anterior, muchos parámetros dejan de ser útiles debido al cuidado que se tiene con la épsilon más especificado e incluso con la optimización que se realizó gracias a este algoritmo Adam, permite que muchos de ellos puedan ser eliminados. Sin embargo, épsilon seguirá perfeccionándose en videos posteriores para implementar el entorno y finalmente llegar al producto final (véase anexo 2.03).

Video 7

En este video se nos plantea la idea de una función para la calibración o disminución de épsilon, con el cual vamos a poder calibrar de mejor manera nuestro agente para que nuestra red neuronal pueda aprender de manera controlada. Aquí mismo planteamos condiciones para que épsilon solo decrezca, de ahí el decaimiento de la función en su ciclo. Como en toda función de programación se tienen que ir mandando los datos para que pueda trabajar con ellos.

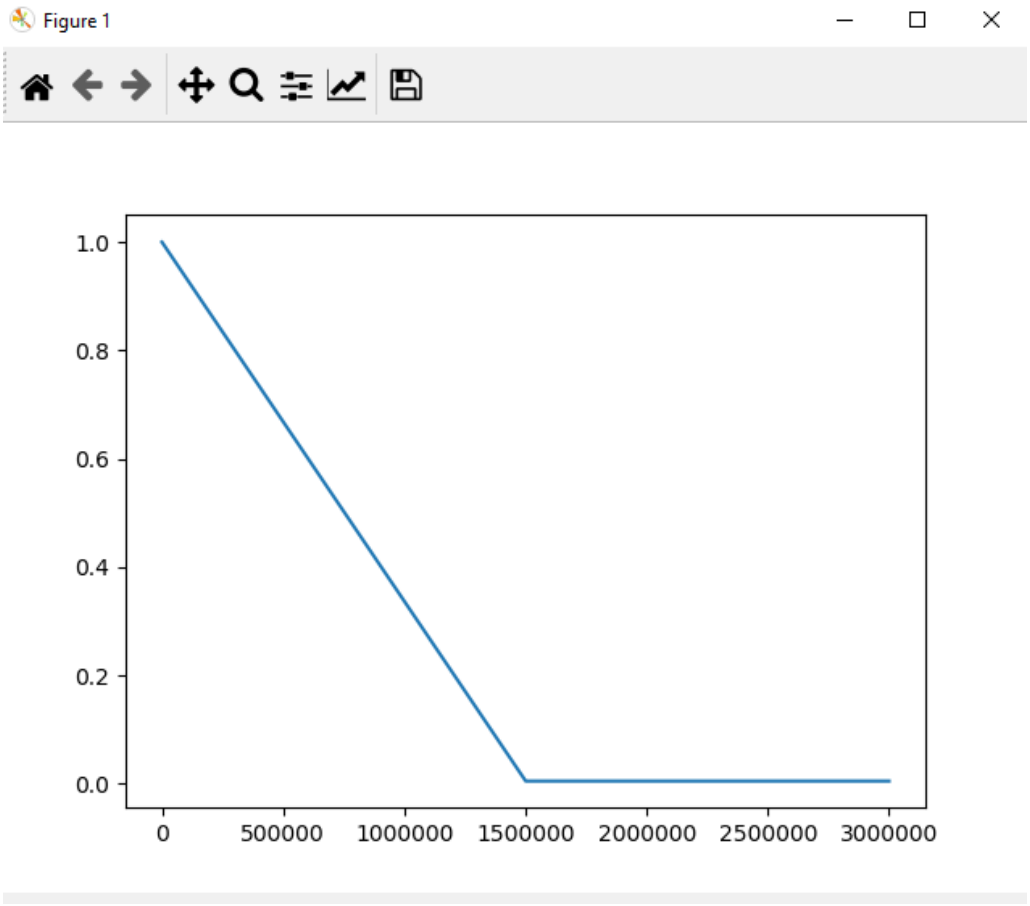
```python
class LinearDecaySchedule(object):
    def __init__(self, initial_value, final_value, max_steps):
        assert initial_value > final_value, "El valor inicial debe ser estrictamente mayor que el valor final."
        self.initial_value = initial_value
        self.final_value = final_value
        self.decay_factor = (initial_value - final_value)/max_steps

    def __call__(self, step_num): #Con esta funcion se va restando con el factor de decarimiento a cada nuevo est
        current_value = self.initial_value - step_num * self.decay_factor
        if current_value < self.final_value:
            current_value = self.final_value
        return current_value
```

Además, se le ponen condiciones a épsilon para que no rebase el valor inicial ni sea más bajo que el valor final. Asi, queda terminada la función de decaimiento de épsilon, pero se nos da la opción de poder comprobar el funcionamiento de

esta. Se puede observar cómo es que funciona en el siguiente grafico generado por la misma función con una nueva función.

```
22
23 if __name__ == "__main__":
24     import matplotlib.pyplot as plt
25     epsilon_initial = 1.0
26     epsilon_final = 0.005
27     MAX_NUM_EPISODES = 10000
28     STEPS_PER_EPISODE = 300
29
30     linear_schedule = LinearDecaySchedule(initial_value = epsilon_initial,
31                                           final_value = epsilon_final,
32                                           max_steps = 0.5 * MAX_NUM_EPISODES * STEPS_PER_EPISODE)
33     epsilons = [linear_schedule(step) for step in range(MAX_NUM_EPISODES * STEPS_PER_EPISODE)]
34     plt.plot(epsilons)
35     plt.show()
```

Con esta función podemos observar lo siguiente:



Se puede observar que justo en medio las decisiones del agente son aleatorias, en el video se explica que el agente utiliza el 50% de los intentos para aprender y el otro 50% lo utiliza para aplicar el conocimiento adquirido para llegar a una solución óptima (véase anexo 2.04).

Video 8

Después de haber creado nuestra función de decaimiento de épsilon, podemos ver en este video como es implementado el entorno y el agente se reinicia en cada intento para no tener alteraciones en los datos de su aprendizaje.

```
104
105 if __name__ == "__main__":
106     environment = gym.make("CartPole-v0")
107     agent = SwallowQLearner(environment)
108     first_episode = True
109     episode_rewards = list()
110     for episode in range(MAX_NUM_EPISODES):
111         obs = environment.reset()
112         total_reward = 0.0
113
```

Acto seguido se nos revela las acciones que toma la red neuronal del entorno observable por el agente y de cómo es que se llevan los valores al procesamiento de la red neuronal. Esto por cada vez que el agente prueba el entorno aprendiendo de su estado actual para poder actuar en el siguiente intento. También se actualiza el número máximo de recompensa al cual ha podido acceder nuestro agente.

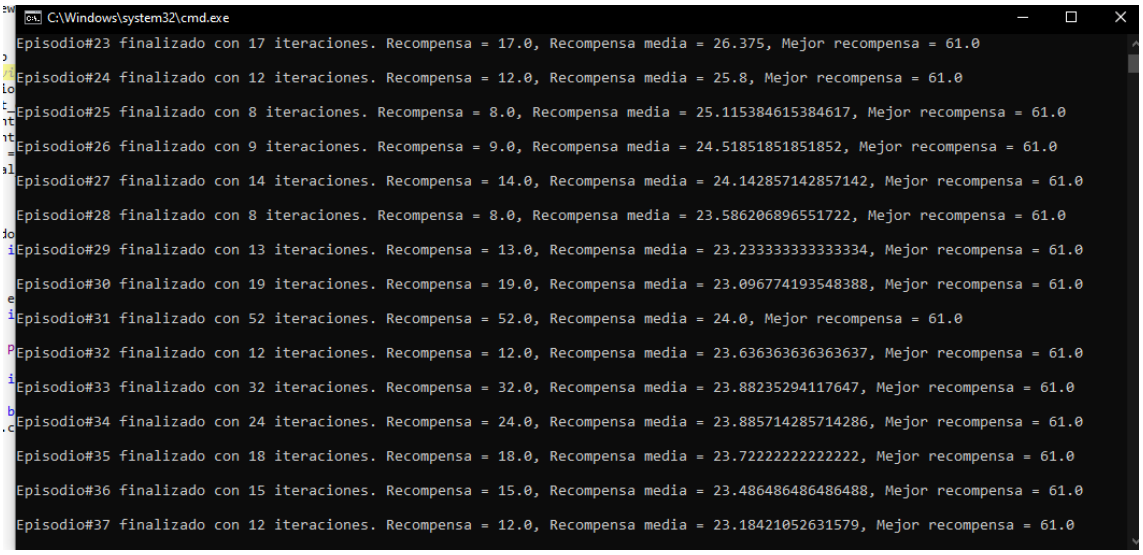
```
114
115         for step in range(STEPS_PER_EPISODE):
116             environment.render()
117             action = agent.get_action(obs)
118             next_obs, reward, done, info = environment.step(action)
119             agent.memory.store(Experience(obs, action, reward, next_obs, done))
120             agent.learn(obs, action, reward, next_obs)
121             obs = next_obs
122             total_reward += reward
123
124             
125
```

Por último, se implementan acciones para poder ver los datos recopilados por cada intento que el agente hace, sus pasos, el número de intento y también las recompensas obtenidas, tanto como por el agente actual como por el más que logró la mejor recompensa (véase anexo 2.04).

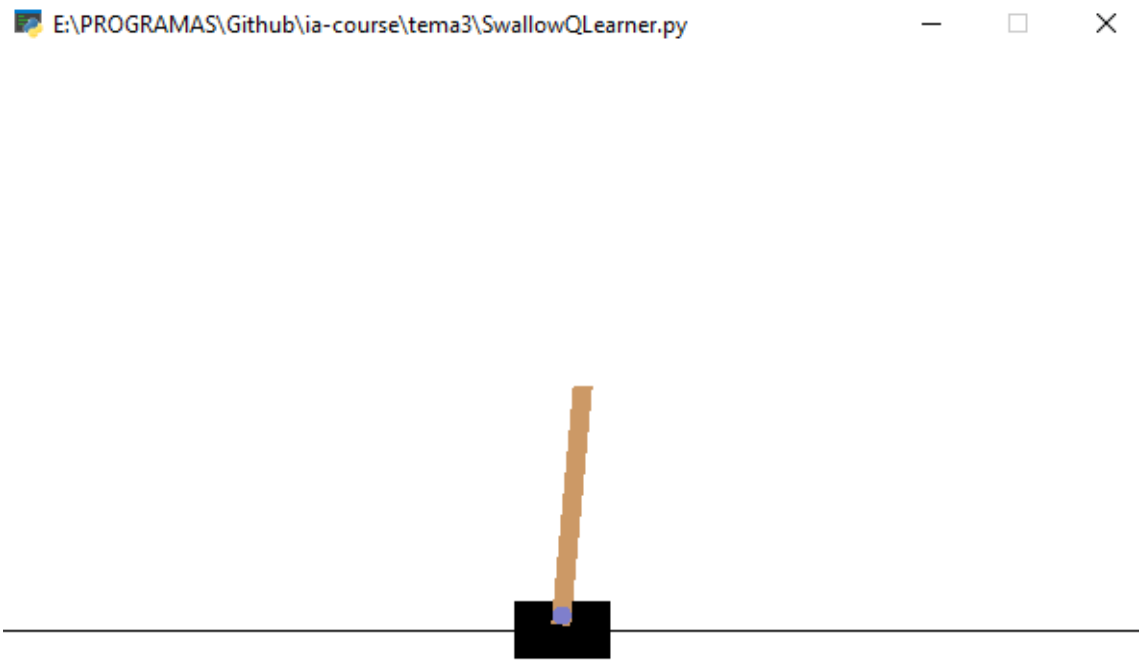
```
125
126             if done is True:
127                 if first_episode:
128                     max_reward = total_reward
129                     first_episode = False
130                 episode_rewards.append(total_reward)
131                 if total_reward > max_reward:
132                     max_reward = total_reward
133                 print("\nEpisodio#{} finalizado con {} iteraciones. Recompensa = {}, Recompensa media = {}, Mejor r
134                       format(episode, step+1, total_reward, np.mean(episode_rewards), max_reward))
135                 if agent.memory.get_size()>1000:
136                     agent.replay_experience(32)
137                 break
138     environment.close()
```

Video 9

Ya como último paso solo queda probar el programa que acabamos de terminar y en este video se nos muestra la ejecución del mismo.



En el video no se muestra la parte grafica porque, debido al número de episodios sería bastante extenso. El resultado de ver el entorno de manera gráfica es el siguiente:



El agente intenta balancear el palo sobre él, cabe aclarar que primero empieza a realizar acciones aleatorias, se requiere más de la mitad de los episodios, para que el agente se vuelva más inteligente (véase anexo 2.04).

En el video, mientras se sigue ejecutando el programa se nos da un poco más de ejemplos de la inteligencia artificial en el mundo de los videojuegos, además de que se muestra un video de una red neuronal que aprende a jugar el mítico juego de Mario Bros.

Video 10

Durante el transcurso del video se comenta que la información puede ser independiente e idéntica, lo cual puede explicar la distribución de los datos, aunque esto no aplica en nuestra red neuronal, ya que esta converge los datos más rápido, tomando en cuenta los estados anteriores y futuros, aunque esto ayuda a llegar más rápido a resultado y/o a una predicción.

Se plantea también que creemos otro programa para implementar una memoria para la red neuronal y que pueda adquirir experiencia de varias de sus ejecuciones.

Video 11

Así como se plantea en el video anterior de la sección, elaboramos el método de experience (véase anexo 2.05) para simular una memoria para el agente, para esto lo tomamos como un tipo tuple, es decir que pueda recuperar distintos tipos de datos, el código es el siguiente:

```python
from collections import namedtuple
import random

Experience = namedtuple("Experience", ['obs', 'action', 'reward', 'next_obs', 'done'])

class ExperienceMemory(object):
    """
    Un buffer que simula la memoria, experiencia del agente
    """
    def __init__(self, capacity = int(1e6)):
        """
        :param capacity: Capacidad total de la memoria cíclica (número máximo de experiencias almacenables)
        :return:
        """
        self.capacity = capacity
        self.memory_idx = 0 #identificador que sabe la experiencia actual
        self.memory = []

    def sample(self, batch_size):
        """
        :param batch_size: Tamaño de la memoria a recuperar
        :return: Una muestra aleatoria del tamaño batch_size de experiencias de la memoria
        """
        assert batch_size <= self.get_size(), "El tamaño de la muestra es superior a la memoria disponible"
        return random.sample(self.memory, batch_size)

    def get_size(self):
        """
        :return: Número de experiencias almacenadas en memoria
        """
        return len(self.memory)

    def store(self, exp):
        """
        :param experience: Objeto experiencia a ser almacenado en memoria
        :return:
        """
        self.memory.insert(self.memory_idx % self.capacity, exp)
        self.memory_idx += 1
```

Video 12

Para la aplicación del método de experience dentro de nuestro programa principal lo llamaremos de la forma: *from experience_memory import ExperienceMemory, Experience*; por lo cuál llamaremos al método ExperienceMemory y al objeto Experience, el cuál contiene el resultado de la experiencia del agente.

Una vez que conectamos al agente con la simulación de memoria o experiencia agregamos un método para utilizarla, el cuál es:

```
    def replay_experience(self, batch_size):
        """
        Vuelve a jugar usando la experiencia aleatoria almacenada
        :param batch_size: Tamaño de la muestra a tomar de la memoria
        :return:
        """
        experience_batch = self.memory.sample(batch_size)
        self.learn_from_batch_experience(experience_batch)

    def learn_from_batch_experience(self, experiences):
        """
        Actualiza la red neuronal profunda en base a lo aprendido en el conjunto de experiencias anteriores
        :param experiences: fragmento de recuerdos anteriores
        :return:
        """
        batch_xp = Experience(*zip(*experiences))
        obs_batch = np.array(batch_xp.obs)
        action_batch = np.array(batch_xp.action)
        reward_batch = np.array(batch_xp.reward)
        next_obs_batch = np.array(batch_xp.next_obs)
        done_batch = np.array(batch_xp.done)

        td_target = reward_batch + ~done_batch * \
        np.tile(self.gamma, len(next_obs_batch)) * \
        self.Q(next_obs_batch).detach().max(1)[0].data

        td_target = td_target.to(self.device)
        action_idx = torch.from_numpy(action_batch).to(self.device)
        td_error = torch.nn.functional.mse_loss(self.Q(obs_batch).gather(1, action_idx.view(-1,1)), td_target.float().unsqueeze(1))

        self.Q_optimizer.zero_grad()
        td_error.mean().backward()
        self.Q_optimizer.step()
```

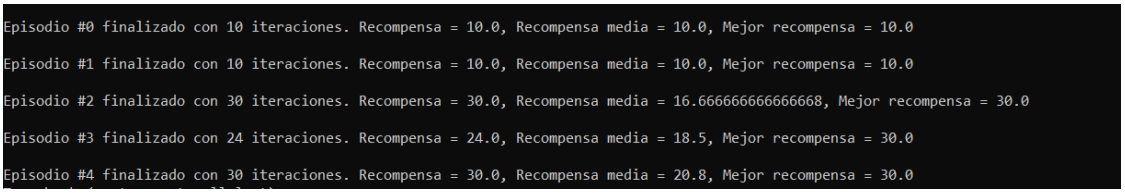
Una vez terminando una ejecución del agente se comenzará a ejecutar a partir de estos métodos que utilizarán su *memoria*, a la vez, en el método learn_from_batch_experience unimos los datos que obtuvimos de las ejecuciones anteriores y lo convertimos en vectores o matrices de datos (véase anexo 2.06).

Video 13

Comenzamos a escribir los últimos detalles del programa, en los métodos que utilizamos para implementar la memoria en el agente utilizamos una variable memory que debe de estar dentro de la clase, para complementar esto la declaramos e igualamos al resultado que suelta el método del programa experience_memory, usándolo como si fuera una librería. En la función principal vamos a agregar el guardado en la memoria con el siguiente comando: *agent.memory.store(Experience(obs, action, reward, next_obs, done))*. Siendo así que, de manera general, el agente puede aprender de cada tirada, generando un tipo de *experiencia*, en la cual se basa para sus siguientes ejecuciones, mejorando considerablemente más en cada intento. El código final se muestra en la siguiente imagen, aunque cabe destacar que son necesarios los programas adicionales que se crean durante esta sección.

```python
import torch
import numpy as np
from perceptron import SLP
from decay_schedule import LinearDecaySchedule
import random
import gym
from experience_memory import ExperienceMemory, Experience

#Definiciones
MAX_NUM_EPISODES = 100000
STEPS_PER_EPISODE = 300

#Clase de los métodos
class SwallowQLearner(object):
    def __init__(self,environment,learning_rate = 0.005,gamma = 0.98):
        self.obs_shape = environment.observation_space.shape

        self.action_shape = environment.action_space.n
        self.Q = SLP(self.obs_shape,self.action_shape)
        self.Q_optimizer = torch.optim.Adam(self.Q.parameters(),lr = learning_rate)
        self.gamma = gamma

        self.epsilon_max = 1.0
        self.epsilon_min=0.05
        self.epsilon_decay = LinearDecaySchedule(initial_value = self.epsilon_max,
                                                 final_value = self.epsilon_min,
                                                 max_steps = 0.5 * MAX_NUM_EPISODES * STEPS_PER_EPISODE)

        self.step_num = 0
        self.policy = self.epsilon_greedy_Q


        self.memory = ExperienceMemory(capacity = int(1e5))
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    def get_action(self,obs):
        return self.policy(obs)
    def epsilon_greedy_Q(self,obs):
        if random.random() < self.epsilon_decay(self.step_num):
            action = random.choice([a for a in range(self.action_shape)])
        else:
            action = np.argmax(self.Q(obs).data.to(torch.device('cpu')).numpy())
        return action

    def learn(self, obs, action, reward, next_obs): #Método de aprendizaje
        td_target = reward + self.gamma * torch.max(self.Q(next_obs))
        td_error = torch.nn.functional.mse_loss(self.Q(obs)[action], td_target)
        self.Q_optimizer.zero_grad()
        td_error.backward()
        self.Q_optimizer.step()

    def replay_experience(self, batch_size):
        """
        Vuelve a jugar usando la experiencia aleatoria almacenada
        :param batch_size: Tamaño de la muestra a tomar de la memoria
        :return:
        """
        experience_batch = self.memory.sample(batch_size)
        self.learn_from_batch_experience(experience_batch)

    def learn_from_batch_experience(self, experiences):
        """
        Actualiza la red neuronal profunda en base a lo aprendido en el conjunto de experiencias anteriores
        :param experiences: fragmento de recuerdos anteriores
        :return:
        """
        batch_xp = Experience(*zip(*experiences))
        obs_batch = np.array(batch_xp.obs)
        action_batch = np.array(batch_xp.action)
        reward_batch = np.array(batch_xp.reward)
        next_obs_batch = np.array(batch_xp.next_obs)
        done_batch = np.array(batch_xp.done)

        td_target = reward_batch + ~done_batch * \
        np.tile(self.gamma, len(next_obs_batch)) * \
        self.Q(next_obs_batch).detach().max(1)[0].data

        td_target = td_target.to(self.device)
        action_idx = torch.from_numpy(action_batch).to(self.device)
        td_error = torch.nn.functional.mse_loss(self.Q(obs_batch).gather(1, action_idx.view(-1,1)), td_target.float().unsqueeze(1))

        self.Q_optimizer.zero_grad()
        td_error.mean().backward()
        self.Q_optimizer.step()


#Función principal
if __name__ == "__main__":
    environment = gym.make("CartPole-v0") #Llamamos el ambiente que se va a intentar resolver
    agent = SwallowQLearner(environment)
    first_episode = True
    episode_rewards = list()
    for episode in range(MAX_NUM_EPISODES):
        obs = environment.reset()
        total_reward = 0.0
        for step in range(STEPS_PER_EPISODE):
            #environment.render()
            action = agent.get_action(obs)
            next_obs, reward, done, info = environment.step(action)
            agent.memory.store(Experience(obs, action, reward, next_obs, done)) #guardamos la experiencia del agente
            agent.learn(obs, action, reward, next_obs)#Llamamos al aprendizaje

            obs = next_obs
            total_reward += reward

            if done is True:
                if first_episode:
                    max_reward = total_reward
                    first_episode = False
                episode_rewards.append(total_reward)
                if total_reward > max_reward:
                    max_reward = total_reward
                print("\nEpisodio #{} finalizado con {} iteraciones. Recompensa = {}, Recompensa media = {}, Mejor recompensa = {}". #Imprime la información obte
                      format(episode, step+1, total_reward, np.mean(episode_rewards), max_reward))
                if agent.memory.get_size()>100:
                    agent.replay_experience(32)
                break
    environment.close()
```

Un pequeño ejemplo de su ejecución es el siguiente:



En donde se puede observar el intento o episodio que se ejecutó, cuantas acciones realizó, su recompensa, la cual es igual a las acciones o movimientos porque su objetivo es realizar la máxima cantidad de acciones posibles, seguido de el promedio de recompensas que ha conseguido contando ese intento y la mejor recompensa que ha obtenido en toda su historia (véase anexo 2.07).

Antes de finalizar el video se nos da a conocer que es casi la mejor optimización que podemos realizar a este programa, al menos hasta el momento de finalización de la sección, pero conocemos que, con los métodos de aprendizaje profundo, los cuales aprenderemos en la siguiente sección, se puede mejorar aún más que con redes neuronales y la simulación de memoria.

Para concluir con esta sección podemos destacar el llamado de otros programas como librerías para optimizar nuestro código y la simulación de una memoria para el agente, siendo que pueda utilizar algo similar a la experiencia, mejorando considerablemente los tiempos para obtener el mejor resultado. También retomamos conocimientos previamente aprendidos y expandimos nuestro conocimiento para utilizar de mejor manera los agentes y además mejorar nuestra lógica de programación basada en las redes neuronales, pertenecientes a la rama de inteligencia artificial, contando como un paso más hacia el aprendizaje profundo y distintos métodos para mejorar los sistemas digitales.

Bibliografía

- Gomila, J. (2019). *Curso de Inteligencia Artificial con Python.* Octubre 31, 2019, de Udemy Sitio web: https://www.udemy.com/course/curso-completo-de-inteligencia-artificial/
- Ponce, P. (2010). *Inteligencia Artificial con Aplicaciones en la Ingeniería.* México: Alfaomega.
- Russell, S. (2004). *Inteligencia Artificial: Un Enfoque Moderno.* España: Prentice Hall.

Anexos

Primera Parte: Anexos Internos al Documento

Anexo 1.01: Las Librerías para el Swallow Q-Learner.

```python
# %% Las librerias
import torch
import numpy as np
from libs.Programa2 import SLP
from utils.decay_schedule import LinearDecaySchedule
import random
import gym
from utils.experience_memory import ExperienceMemory, Experience
```

Anexo 1.02: La nueva inicialización de la función Q.

```python
# %% La clase Swallow Q-Learner o Agente de Q-Learning Ligero
class SwallowQLearner(object):
    def __init__(self, environment, learning_rate = 0.005, gamma = 0.98):
        self.obs_shape = environment.observation_space.shape

        self.action_shape = environment.action_space.n
        self.Q = SLP(self.obs_shape, self.action_shape)
        self.Q_optimizer = torch.optim.Adam(self.Q.parameters(), lr = learning_rate)

        self.gamma = gamma

        self.epsilon_max = 1.0
        self.epsilon_min = 0.05
        self.epsilon_decay = LinearDecaySchedule(initial_value = self.epsilon_max,
                                                 final_value = self.epsilon_min,
                                                 max_steps = 0.5 * MAX_NUM_EPISODES * STEPS_PER_EPISODE)
        self.step_num = 0
        self.policy = self.epsilon_greedy_Q

        self.memory = ExperienceMemory(capacity = int(1e5))
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Anexo 1.03:

Segunda Parte: Anexos Externos al Documento

Esta sección de anexos no pudo ser incluida en el presente, debido a que son los archivos de código de los videos.

Se anexan en caso de que se quiera realizar una prueba de ejecución o revisar el código de forma completa, puesto que los extractos más importantes del código se encuentran en los anexos internos al documento.

Estos anexos se adjuntaron al mismo documento en su entrega con el mismo nombre que se describen a continuación.

Anexo 2.01: Programa 1

Anexo 2.02: Programa 2

Anexo 2.03: Programa 3

Anexo 2.04: Programa 4

Anexo 2.05: Programa 5

Anexo 2.06: Programa 6

Anexo 2.07: Programa 7

Es necesario comentar que en este anexo, este programa se ejecuta normalmente pero no puede completar totalmente su ejecución. Se programó distintas ocasiones e incluso se tomó el código del instructor de los videos y el archivo no pudo ejecutarse. Se cree que por una característica de Windows no fue posible su ejecución óptima, sin embargo, el código es el mismo que el del video.