# Taller de inteligencia artificial

Grupo:

Este resumen se centra en el proyecto desarrollado en el marco del curso "Taller de Inteligencia Artificial". El objetivo principal de este proyecto fue diseñar y construir un agente de inteligencia artificial capaz de aprender a jugar el icónico videojuego Mario Bros utilizando la técnica de aprendizaje por refuerzo y la implementación de los algoritmos Deep Q-Network (DQN) y Double DQN. Para alcanzar este objetivo, se utilizaron diversas tecnologías, entre las cuales se destacan Gym, PyTorch y Python, aprovechando los principios de programación orientada a objetos.

El aprendizaje de refuerzo es una rama de la inteligencia artificial que se basa en el concepto de que un agente interactúa con un entorno, realizando acciones y recibiendo retroalimentación como recompensas (positivas) o castigos (negativas) dependiendo de su desempeño. El objetivo final es que el agente aprenda a tomar las mejores decisiones para maximizar las recompensas a lo largo del tiempo.

En el caso de nuestro proyecto, comenzamos probando el algoritmo DNQ proporcionado por Alexander Zai y Brandon Brown en el libro “Deep Reinforcement Learning in Action” donde el ambiente es un Gridworld o mundo de celdas donde un agente representado por una letra “P” se mueve por una cuadrícula bidimensional compuesta por celdas. Cada celda puede tener diferentes propiedades, como ser un estado objetivo, un obstáculo o un espacio vacío. El objetivo del agente es navegar por la cuadrícula y llegar al estado objetivo mientras evita los obstáculos.

| *# Así se puede mostrar gráficamente el ambiente del juego*  game**.**display() |
| --- |

Esto imprime lo siguiente:

| array([['+', '-', ' ', 'P'],  [' ', 'W', ' ', ' '],  [' ', ' ', ' ', ' '],  [' ', ' ', ' ', ' ']], dtype='<U2') |
| --- |

Donde ”P” es el jugador o player, “W” es un muro, “-” es un pozo y “+” es el objetivo a llegar.

Resolvimos este juego con el siguiente código proporcionado por el libro de forma que pudimos generar una explicación a cada línea de código resultando en la imagen a continuación. Con lo cual los conceptos básicos del aprendizaje por refuerzo ya estaban integrados en nuestras mentes.



Una vez aprendido esto y con una revisión de las clases grabadas del curso seguimos con el problema de la “replay memory” es decir, un mecanismo que consiste en almacenar las experiencias pasadas (estado, acción, recompensa, estado siguiente) del agente para entrenar al modelo. Este mecanismo tiene las siguientes ventajas: rompe la correlación temporal, mejora la eficiencia de aprendizaje, facilita el aprendizaje fuera de distribución y estabiliza el entrenamiento. Esta idea ejemplificada en el código nos resulta de la siguiente forma:

| from collections import deque epochs = 7000 losses = [] mem\_size = 2000 batch\_size = 300 replay = deque(maxlen=mem\_size) max\_moves = 50  h = 0  for i in range(epochs):  game = Gridworld(size=4, mode='random')  state1\_ = game.board.render\_np().reshape(1, 64) \  + np.random.rand(1, 64) / 100.0  state1 = torch.from\_numpy(state1\_).float()  status = 1  mov = 0  while (status == 1):  mov += 1  qval = model(state1)  qval\_ = qval.data.numpy()  if (random.random() < epsilon):  action\_ = np.random.randint(0, 4)  else:  action\_ = np.argmax(qval\_)  action = action\_set[action\_]  game.makeMove(action)  state2\_ = game.board.render\_np().reshape(1, 64) \  + np.random.rand(1, 64) / 100  state2 = torch.from\_numpy(state2\_).float()  reward = game.reward()  done = True if reward > 0 else False  exp = (state1, action\_, reward, state2, done)  replay.append(exp)  state1 = state2   if len(replay) > batch\_size:  minibatch = random.sample(replay, batch\_size)  state1\_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch])  action\_batch = torch.Tensor([a for (s1,a,r,s2,d) in minibatch])  reward\_batch = torch.Tensor([r for (s1,a,r,s2,d) in minibatch])  state2\_batch = torch.cat([s2 for (s1,a,r,s2,d) in minibatch])  done\_batch = torch.Tensor([d for (s1,a,r,s2,d) in minibatch])   Q1 = model(state1\_batch)  with torch.no\_grad():  Q2 = model(state2\_batch)    Y = reward\_batch + gamma \* (  (1 - done\_batch) \* torch.max(Q2, dim=1)[0])  X = Q1.gather(  dim=1, index=action\_batch.long().unsqueeze(dim=1)).squeeze()  loss = loss\_fn(X, Y.detach())  optimizer.zero\_grad()  loss.backward()  losses.append(loss.item())  optimizer.step()  if reward != -1 or mov > max\_moves:  status = 0  mov = 0 losses = np.array(losses) |
| --- |

(compartimos el código completo en el siguiente repositorio:

<https://github.com/crisdavid3335/Master_Ort/blob/main/Taller_IA/grid_/Grid_word.ipynb>)

Esto, una vez más nos aclaró las ideas básicas de cómo el algoritmo funciona y por tanto sólo nos hacía falta aplicarlo al código del obligatorio, con lo cual una de las primeras iteraciones fue simplemente aplicar el bucle de entrenamiento al ambiente de mario proporcionado para el obligatorio.

| import torch from torch import optim import torch.nn as nn from replay\_memory import ReplayMemory from abc import ABC, abstractmethod from tqdm import tqdm from torch.utils.tensorboard import SummaryWriter from mario\_utils import show\_video, wrap\_env import random import numpy as np  class Agent:  *# class Agent(ABC):*  def \_\_init\_\_( self, gym\_env, model, obs\_processing\_func,  memory\_buffer\_size, batch\_size, learning\_rate, gamma,  epsilon\_i, epsilon\_f, epsilon\_anneal\_time, epsilon\_decay, episode\_block):  self.device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")  *# Función phi para procesar los estados.*  self.model = model.to(self.device)  self.obs\_processing\_func = obs\_processing\_func  *# Asignar memoria al agente*  self.memory = ReplayMemory(memory\_buffer\_size)  self.env = gym\_env   *# Hyperparameters*  self.batch\_size = batch\_size  self.learning\_rate = learning\_rate  self.gamma = gamma  self.loss\_fn = nn.MSELoss().to(self.device)   self.epsilon\_i = epsilon\_i  self.epsilon\_f = epsilon\_f  self.epsilon\_anneal = epsilon\_anneal\_time  self.epsilon\_decay = epsilon\_decay  self.episode\_block = episode\_block   self.total\_steps = 0   def train(self, number\_episodes=50000, max\_steps\_episode=10000,  max\_steps=1000000, writer\_name="default\_writer\_name"):  rewards = []  total\_steps = 0  writer = SummaryWriter(comment="-" + writer\_name)  optimizer = optim.Adam(self.model.parameters(), lr=self.learning\_rate)  losses = []  self.number\_episodes = number\_episodes   for ep in tqdm(range(number\_episodes), unit=" episodes"):  if total\_steps > max\_steps:  break   *# Observar estado inicial como indica el algoritmo*  state = self.env.reset()  current\_episode\_reward = 0.0   for s in range(max\_steps):  self.qval\_ = self.model(  torch.from\_numpy(np.array(self.obs\_processing\_func(state)))  .unsqueeze(0)  .to(self.device)  )  *# Seleccionar acción usando una política epsilon-greedy.*  self.epsilon = self.compute\_epsilon(ep)  if random.random() < self.epsilon:  action = torch.randint(0, self.env.action\_space.n, (1,)).item()  else:  action = torch.argmax(self.qval\_).item()  *# Ejecutar la acción, observar resultado y procesarlo como indica el algoritmo.*  next\_state, reward, done, info = self.env.step(action)  current\_episode\_reward += reward  total\_steps += 1   *# Guardar la transición en la memoria*  self.memory.add(state, action, reward, done, next\_state)   *# Actualizar el estado*  state = next\_state  *# Actualizar el modelo*  if len(self.memory) > self.batch\_size:  self.minibatch = self.memory.sample(self.batch\_size)  self.state\_batch = torch.cat(  [  torch.from\_numpy(np.array(self.obs\_processing\_func(s1)))  .unsqueeze(0)  .to(self.device)  for (s1, a, r, d, s2) in self.minibatch  ]  ).to(self.device)  self.action\_batch = torch.Tensor(  [a for (s1, a, r, d, s2) in self.minibatch]  ).to(self.device)  self.reward\_batch = torch.Tensor(  [r for (s1, a, r, d, s2) in self.minibatch]  ).to(self.device)  self.done\_batch = torch.Tensor(  [d for (s1, a, r, d, s2) in self.minibatch]  ).to(self.device)  self.next\_state\_batch = torch.cat(  [  torch.from\_numpy(np.array(self.obs\_processing\_func(s2)))  .unsqueeze(0)  .to(self.device)  for (s1, a, r, d, s2) in self.minibatch  ]  ).to(self.device)   Q1 = self.model(self.state\_batch).to(self.device)  with torch.no\_grad():  Q2 = self.model(self.next\_state\_batch).to(self.device)   Y = self.reward\_batch + self.gamma \* (  (1 - self.done\_batch) \* torch.max(Q2, dim=1)[0]  ).to(self.device)  X = (  Q1.gather(  dim=1, index=self.action\_batch.long().unsqueeze(dim=1)  )  .squeeze()  .to(self.device)  )  loss = self.loss\_fn(X, Y.detach())  optimizer.zero\_grad()  loss.backward()  losses.append(loss.item())  optimizer.step()   if done:  break  if s == max\_steps\_episode:  break   rewards.append(current\_episode\_reward)  mean\_reward = np.mean(rewards[-100:])  writer.add\_scalar("epsilon", self.epsilon, total\_steps)  writer.add\_scalar("reward\_100", mean\_reward, total\_steps)  writer.add\_scalar("reward", current\_episode\_reward, total\_steps)   *# Report on the traning rewards every EPISODE BLOCK episodes*  if ep % self.episode\_block == 0:  print(  f"Episode {ep} - Avg. Reward over the last {self.episode\_block} episodes {np.mean(rewards[-self.episode\_block:])} epsilon {self.epsilon} total steps {total\_steps}"  )   print(  f"Episode {ep + 1} - Avg. Reward over the last {self.episode\_block} episodes {np.mean(rewards[-self.episode\_block:])} epsilon {self.epsilon} total steps {total\_steps}"  )   torch.save(  self.model.state\_dict(),  "/mnt/c/Users/crisd/OneDrive/Escritorio/modelo\_pesos.pth",  )  writer.close()   return rewards   def compute\_epsilon(self, steps\_so\_far):  return 1 - steps\_so\_far / self.number\_episodes   def record\_test\_episode(self, env):  done = False   env = wrap\_env(env)  state = env.reset()  *# Observar estado inicial como indica el algoritmo*   while not done:  *# env.render() # Queremos hacer render para obtener un video al final.*  self.qval\_ = self.model(torch.from\_numpy(np.array(state)).unsqueeze(0))  action = torch.argmax(self.qval\_).item()  state, reward, done, info = env.step(action)  if done:  break   *# Actualizar el estado*  env.close()  show\_video()  *# @abstractmethod* *# def select\_action(self, state, current\_steps, train=True):* *# pass* *#* *# @abstractmethod* *# def update\_weights(self):* *# pass* |
| --- |

Este código, si bien ya optimizaba los pesos y nos permitió empezar a obtener los primeros resultados en los videos del aprendizaje aún carecía del uso de la estructura de clases que se requieren. Además de tampoco usar GPU.

Al entrenar a nuestro agente utilizando DNQ en el entorno MountainCar-v0, esperábamos que aprendiera a tomar las acciones necesarias para superar los desafíos y alcanzar el éxito en el juego. El algoritmo DNQ utiliza una red neuronal profunda para aproximar la función Q, que representa la calidad de una acción en un estado determinado. A través del entrenamiento y la exploración del entorno, el agente aprende a tomar decisiones óptimas basadas en su conocimiento acumulado y las recompensas recibidas.

Utilizando MountainCar-v0 como entorno de prueba, nos permitió familiarizarnos con el algoritmo DNQ y evaluar su rendimiento en un escenario relativamente sencillo. Una vez que obtuvimos resultados satisfactorios en este entorno, nos propusimos el desafío de aplicar el algoritmo DNQ al juego de Mario Bros, que es considerablemente más complejo. Además también nos enfocamos en implementar la estructura de clases propuesta en nuestro proyecto. La programación orientada a objetos nos permitió organizar y modularizar nuestro código de manera eficiente, lo cual es especialmente importante cuando trabajamos en proyectos complejos.

Al expandir nuestro proyecto al juego de Mario Bros, tuvimos que enfrentar nuevos desafíos, como la alta dimensionalidad de los estados y encontrar cuales eran las dimensiones (shape’s) de las matrices y tensores que hacen que nuestro modelo entrene; esta fue la parte más difícil. Sin embargo, estábamos seguros de que el algoritmo DNQ y la metodología de imprimir las dimensiones de todo e ir ajustando, nos permitirá hacer frente a estos obstáculos y lograr que nuestro agente aprendiera a jugar Mario Bros de manera efectiva.

En resumen, nuestro proyecto de semestre se centró en el diseño y construcción de un agente de inteligencia artificial capaz de aprender a jugar Mario Bros mediante el uso de la técnica de aprendizaje por refuerzo y la implementación del algoritmo DNQ. Comenzamos probando el algoritmo en el entorno MountainCar-v0 de Gym y luego nos enfrentamos al desafío de aplicarlo al juego de Mario Bros. A través de la combinación de Gym, PyTorch y Python, esperábamos lograr que nuestro agente aprendiera a tomar decisiones inteligentes y a jugar el juego de manera exitosa.

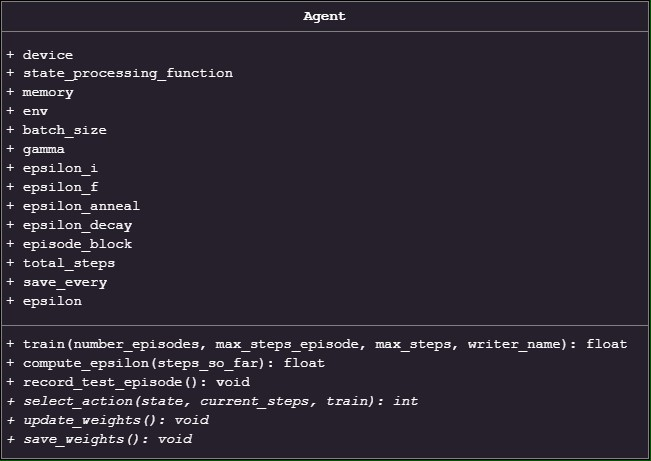
Descripción del proyecto: Una vez se han expuesto las dificultades sorteadas en el proyecto es preciso describir el estado al que se llegó. El proyecto ocupa el entorno de desarrollo Gym, una herramienta ampliamente utilizada en el ámbito de la inteligencia artificial para la implementación y evaluación de algoritmos de aprendizaje por refuerzo. Gym proporciona una amplia variedad de entornos de juego, incluido el clásico Mario Bros, que permiten a los investigadores y desarrolladores probar y comparar diferentes enfoques de aprendizaje de refuerzo.

El algoritmo clave utilizado en este proyecto fue el Deep Q-Network (DNQ), una técnica de aprendizaje por refuerzo que combina redes neuronales profundas con el algoritmo Q-Learning. DNQ se ha demostrado eficaz en la resolución de problemas complejos en entornos de juego, aprendiendo directamente de los datos de entrada y mejorando su desempeño a medida que acumula experiencia.

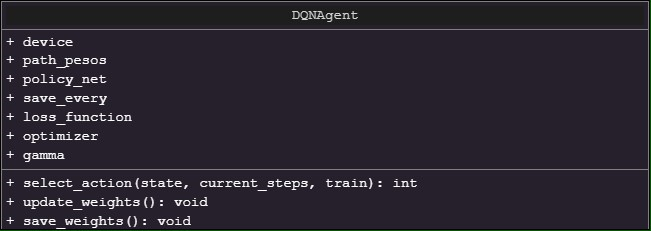
Además del DQN, también se implementó el algoritmo Double DQN. Este algoritmo es una mejora del DQN que aborda la sobreestimación de valores Q, un desafío común en el aprendizaje por refuerzo. El Double DQN utiliza dos redes neuronales, una para seleccionar las acciones y otra para evaluar el valor de esas acciones. Esto ayuda a reducir la sobreestimación de los valores Q y mejora el desempeño del agente.

Para implementar el proyecto, se utilizó el framework PyTorch, una biblioteca de código abierto ampliamente utilizada para la creación y entrenamiento de redes neuronales. PyTorch proporciona una amplia gama de herramientas y funciones que facilitan el proceso de construcción y entrenamiento de modelos de aprendizaje profundo.

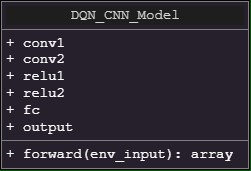
La clase inicial del proyecto es Agent, que es una clase abstracta que posee la lógica del bucle que hemos definido según los conceptos y definiciones tratados en el curso. Esta clase posee los siguientes atributos que son públicos: device, que es para enviar a GPU los tensores de pytorch, state\_processing\_function que es para procesar los estados recibidos por gym y convertirlo a tensores de pytorch con las dimensiones deseadas, memory que es una instancia de la clase ReplayMemory, env que es una instancia de gym con el ambiente de Mario, batch\_size que es la cantidad de experiencias a tener en cuenta en el entrenamiento del modelo (tamaño del batch), gamma que es la preferencia temporal por la recompensa del agente, epsilon\_i que es el epsilon con el cual el agente inicia en el entrenamiento, epsilon\_f que es el epsilon con el cual el agente finaliza el entrenamiento, epsilon\_anneal es la cantidad de episodios para que epsilon\_i llegue a ser epsilon\_f, epsilon\_decay que es un decimal que marca cuanto reducirá epsilon en el paso siguiente con respecto al actual, epsilon\_block no lo ocupamos, pero decidimos dejarlo, total\_steps es el máximo número de pasos de tiempo en el total de episodios de entrenamiento antes de que se corte, save\_every es el intervalo de pasos de tiempo en el que se generará una copia de los pesos del modelo en un archivo .pth , finalmente epsilon que es el atributo de epsilon para el agente en ese momento de tiempo. La clase abstracta tiene también diferentes métodos que son los siguientes; train que define la lógica secuencial del entrenamiento, define paso a paso que hace el agente con el ambiente y como maneja las observaciones obtenidas de él y retorna una lista de recompensas, compute\_epsilon que recibe como parámetro el episodio actual de entrenamiento y retorna el epsilon para ese episodio, record\_test\_episode es un método que ejecuta un episodio con la red sin la estrategia epsilon greedy y que guarda el episodio en video para mostrarlo después, los métodos select\_action, update\_weights, save\_weights son métodos abstractos sin definir.



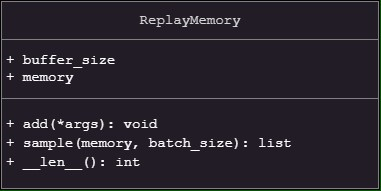
La siguiente clase es DQNAgent que posee los siguientes atributos; device que sirve para enviar los tensores a el dispositivo de GPU, path\_pesos que es un string de la ruta al archivo de pesos del modelo, policy\_net que es una instancia de la clase DQN\_CNN\_Model que es un modelo definido en pytorch según las especificaciones del paper propuesto para el obligatorio, save\_every qué es un intervalo de tiempo en el cual los pesos del modelo se van a guardar en el archivo de la ruta de pesos, loss\_function que es una métrica para medir la distancia entre la predicción del modelo y un target, optimizer que es el optimizador de pesos del modelo que se va a usar en la retropropagación, gamma que es la preferencia temporal del agente por una recompensa futura sobre una actual. Como métodos de la función esta clase posee; select\_action que retorna un entero (que representa una acción), update\_weights que posee la lógica en la cual el modelo (o modelos) actualizarán sus pesos.



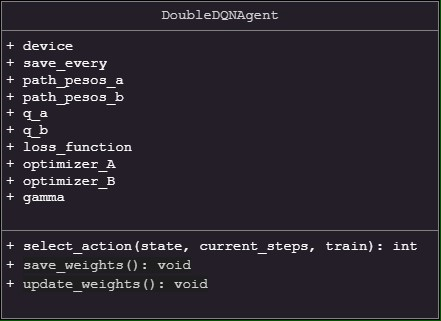
La clase del modelo convolucional DQN\_CNN\_Model posee los siguientes atributos de los cuales todos son públicos; conv1 que es una instancia del objeto Conv2d del módulo nn de pytorch, conv2 que es otra instancia de un objeto Conv2d de pytorch, relu1 que es una instancia de la clase ReLU como función de activación, relu2 que es otra instancia del objeto ReLU de pytorch, fc que es una instancia de una capa lineal de pytorch, output que es otra instancia de la clase Linear del módulo nn de pytorch. Esta capa posee un método llamado forward que recibe como parámetro “env\_input” que es una observación del ambiente y la pasa por las capas en el siguiente orden: conv1, relu1, conv2, relu2, luego lo aplana o lo convierte en un solo vector de dimensión (2592,) que es la entrada para la capa lineal fc que tiene de salida un vector de (256,), se procesa esta salida por una capa de activación relu y finalmente por una capa lineal que recibe el vector (256,) y devuelve un vector de salida n\_actions que depende del ambiente.



La clase ReplayMemory posee los atributos buffer\_size que es la cantidad de memorias que va a almacenar y memory que es una instancia de un objeto deque de la librería estándar de python. Esta clase posee tres métodos que son; add que acepta como parámetro varios argumentos nombrados, este método toma esos argumentos dentro de una tupla nombrada Transition y la añade a la memoria, sample que recibe batch\_size por parámetro que es un número entero y retorna una cantidad batch\_size de tuplas de Transition tomadas aleatoriamente del total de la memoria. Método \_\_len\_\_ modifica el atributo propio de longitud de la clase y lo hace retornar la longitud de la memoria (el objeto definido previamente).



Otra clase creada es DoubleDQNAgent que contiene la lógica del algoritmo doble Q learning y posee los siguientes atributos; device que es el dispositivo al que enviamos los tensores para usar GPU, save\_every qué es el intervalo de tiempo en el que los pesos se copiaran a dos archivo de pesos (en este caso se ocupan dos diferentes tipos de archivos ya que el agente usa dos modelos), path\_pesos\_a que es la ruta de pesos a la cual se enviaran los pesos de la primera red, path\_pesos\_b es la ruta a en la cual se copiaran los pesos de la segunda red, q\_a que es la primera red, q\_b que es la segunda red, loss\_function que es la métrica de error elegida (en este caso una MSE), optimizer\_A y optimizer\_B que son los optimizadores de las redes A y B correspondientemente, gamma que es la preferencia temporal de las rewards. Esta clase posee los siguientes métodos; select\_action que devuelve una acción codificada en un número entero y recibe una observación del ambiente, el número de episodio actual y un identificador de si el agente se encuentra en un entrenamiento o no (esto se utiliza para activar o desactivar la estrategia epsilon greedy), save\_weights que guarda los pesos de ambas redes en archivos previamente definidos. Por último update\_weights que es el método responsable de actualizar los pesos de las redes según el algoritmo; en el caso de doble Q learning se aplica la actualización de pesos a una red u otra aleatoriamente esto ayuda a controlar la sobreestimación.



Descripción del algoritmo:

A continuación, se describen los pasos principales del algoritmo DQN:

El algoritmo implementado sigue una metodología en tres etapas para el entrenamiento de un agente de aprendizaje por refuerzo utilizando la variante del algoritmo DQN. En la etapa de inicialización, se definen y se asignan los parámetros y estructuras necesarios para el entrenamiento, como la memoria de replay y los hiperparámetros que controlan el proceso de aprendizaje.

En la etapa del ciclo de entrenamiento, el agente interactúa con el entorno a través de episodios. En cada episodio, se establece el estado inicial y se ejecutan pasos dentro del episodio. El agente selecciona una acción utilizando una política epsilon-greedy, y después de cada acción se obtiene la siguiente observación, la recompensa y se verifica si se alcanzó un estado terminal. Estos datos se almacenan en la memoria de replay, y los pesos del modelo se actualizan mediante la optimización de la función de pérdida.

En la etapa de actualización y evaluación, se realizan acciones posteriores a cada episodio. Se calcula la recompensa promedio de los últimos 100 episodios para evaluar el desempeño del agente y se registran los valores en TensorBoard para un análisis más detallado. Además, los pesos del modelo se guardan periódicamente para mantener un respaldo de los avances alcanzados. El algoritmo continúa ejecutando episodios hasta que se alcance un número máximo de episodios o un número máximo de pasos totales, lo que permite al agente mejorar su desempeño a medida que aprende a tomar decisiones óptimas en el entorno de forma autónoma.

A continuación, se describen los pasos principales del algoritmo doble DQN:

El algoritmo Double DQN sigue un proceso de aprendizaje similar a DQN, pero con algunas diferencias clave en el paso de actualización de pesos. A continuación se presenta el pseudocódigo del algoritmo Double DQN:

Se inicializan la red Q\_a y la red Q\_b con pesos aleatorios.

Inicializar la memoria de repetición.

Para cada episodio:

a. Reiniciar el entorno y obtener el estado inicial.

b. Mientras el episodio no haya terminado:

i. Seleccionar una acción epsilon-greedy utilizando la red Q principal.

ii. Ejecutar la acción en el entorno y observar el siguiente estado, la recompensa y si el episodio ha terminado.

iii. Almacenar la transición en la memoria de repetición.

iv. Muestrear un minibatch de transiciones de la memoria de repetición.

v. Para cada transición en el minibatch:

1. Calcular el valor Q actual utilizando la red Q\_a o Q\_b.

2. Calcular el valor Q siguiente utilizando la red Q\_a o Q\_b.

3. Seleccionar la acción que maximice el valor Q siguiente utilizando la red Q\_a o Q\_b.

4. Calcular el target de DQN utilizando la ecuación:

Y = recompensa + gamma \* (1 - episodio\_terminado) \* Q\_siguiente[action\_maximizadora]

5. Calcular la pérdida cuadrática media entre el valor Q actual y el target de DQN.

6. Retropropagar el error y actualizar los pesos de la red Q seleccionada.

Repetir los pasos 3-6 hasta que se cumpla el criterio de finalización.

Conclusión: El proyecto ha permitido explorar y aplicar los conocimientos adquiridos en la materia de Taller de Inteligencia Artificial en la creación de un agente de aprendizaje por refuerzo capaz de jugar el videojuego Mario Bros. Mediante la combinación de técnicas como DNQ, Gym, PyTorch y Python, se ha logrado entrenar al agente para que aprenda a tomar decisiones inteligentes y optimizar su rendimiento en el juego.

Este proyecto no solo ha brindado una experiencia práctica en el campo de la inteligencia artificial, sino que también ha permitido comprender los desafíos y las posibilidades que ofrece el aprendizaje por refuerzo en el desarrollo de agentes inteligentes capaces de interactuar con su entorno y aprender de él. El conocimiento y las habilidades adquiridas a lo largo del proyecto seguramente.