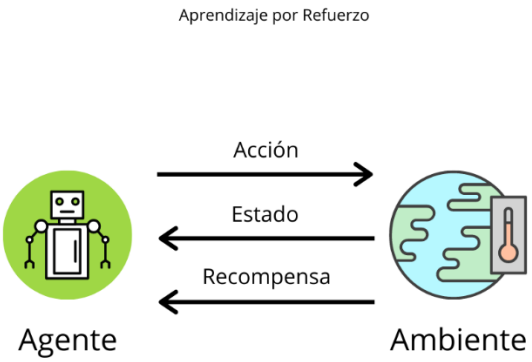
Centro Universitario de Ciencias Exactas e Ingenierías

Universidad de Guadalajara

Actividad 11: Algoritmos de aprendizaje por refuerzo

Aprendizaje Máquina



Alumno: Samuel David Pérez Brambila

Código: 222966286

Profesora: Karla Ávila Cárdenas

Sección: D01

Fecha de Entrega: 29 de Octubre de 2024

Introducción

El aprendizaje por refuerzo es “una técnica de machine learning (ML) que entrena al software para que tome decisiones y logre los mejores resultados. Imita el proceso de aprendizaje por ensayo y error que los humanos utilizan para lograr sus objetivos. Las acciones de software que trabajan para alcanzar su objetivo se refuerzan, mientras que las que se apartan del objetivo se ignoran.” (AWS, s.f.)

Los algoritmos de aprendizaje por refuerzo utilizan un paradigma de recompensa y castigo al procesar los datos. Aprenden de los comentarios de cada acción y descubren por sí mismos las mejores rutas de procesamiento para lograr los resultados finales. Los algoritmos también son capaces de funcionar con gratificación aplazada. La mejor estrategia general puede requerir sacrificios a corto plazo, por lo que el mejor enfoque descubierto puede incluir algunos castigos o dar marcha atrás en el camino. El aprendizaje por refuerzo es un potente método que ayuda a los sistemas de inteligencia artificial a lograr resultados óptimos en entornos invisibles.

De acuerdo con AWS (s.f.), algunas ventajas del aprendizaje por refuerzo son:

- Sobresale en entornos complejos: Los algoritmos de aprendizaje por refuerzo se pueden utilizar en entornos complejos con muchas reglas y dependencias. En el mismo entorno, es posible que un ser humano no sea capaz de determinar el mejor camino a seguir, incluso con un conocimiento superior del entorno. En cambio, estos algoritmos sin modelo se adaptan rápidamente a entornos que cambian continuamente y encuentran nuevas estrategias para optimizar los resultados.
- Requiere menos interacción humana: En el caso de los algoritmos de machine learning tradicionales, se necesitan personas que etiqueten pares de datos para dirigir el algoritmo. Cuando se utiliza un algoritmo de aprendizaje por refuerzo, esto no es necesario. El algoritmo aprende por sí mismo. Al mismo tiempo, ofrece mecanismos para integrar la retroalimentación humana, lo que permite crear sistemas que se adapten a las preferencias, la experiencia y las correcciones humanas.
- Optimiza de acuerdo con objetivos a largo plazo: El aprendizaje por refuerzo se centra intrínsecamente en la maximización de las recompensas a largo plazo, lo que lo hace apto para escenarios en los que las acciones tienen consecuencias prolongadas. Es especialmente adecuado para situaciones del mundo real en las que no hay retroalimentación disponible de inmediato para cada paso, ya que puede aprender de las recompensas retrasadas.

El aprendizaje por refuerzo se puede aplicar a una amplia gama de casos de uso del mundo real. Según AWS (s.f.), encontramos:

- Personalización de marketing: En aplicaciones como los sistemas de recomendación, el aprendizaje por refuerzo puede personalizar las sugerencias para los usuarios individuales en función de sus interacciones. Esto lleva a experiencias más personalizadas. Por ejemplo, una aplicación puede mostrar anuncios a un usuario en función de cierta información demográfica. Con cada interacción publicitaria, la aplicación aprende qué anuncios mostrar al usuario para optimizar las ventas de productos.
- Desafíos de optimización: Los métodos de optimización tradicionales resuelven los problemas mediante la evaluación y comparación de las posibles soluciones en función de ciertos criterios. Por el contrario, el aprendizaje por refuerzo (RL) introduce el aprendizaje a partir de las interacciones para encontrar las mejores soluciones (o las más cercanas a las mejores) a lo largo del tiempo. Por ejemplo, un sistema de optimización del gasto en la nube utiliza el RL para adaptarse a las necesidades de recursos fluctuantes y elegir los tipos, cantidades y configuraciones de instancias óptimos. Toma decisiones en función de factores como la infraestructura de nube actual y disponible, los gastos y la utilización.
- Predicciones financieras: La dinámica de los mercados financieros es compleja, con propiedades estadísticas que cambian con el tiempo. Los algoritmos de RL pueden optimizar los rendimientos a largo plazo al considerar los costos de transacción y adaptarse a los cambios del mercado. Por ejemplo, un algoritmo podría observar las reglas y los patrones del mercado de valores antes de probar las acciones y registrar las recompensas asociadas. El algoritmo crea una función de valor y desarrolla una estrategia para maximizar las ganancias.

Para profundizar en este tema, a continuación se presenta un resumen de un artículo de una página web que ofrece una explicación detallada sobre el aprendizaje por refuerzo.

Contenido de la Actividad

¿Qué es el aprendizaje por refuerzo?

Seguramente ya se conocerán las 2 grandes áreas de aprendizaje tradicional del Machine Learning, el aprendizaje supervisado y el aprendizaje no supervisado. Parece difícil que aquí hubiera espacio para otras opciones; sin embargo, sí la hay y es el aprendizaje por refuerzo. En aprendizaje por refuerzo (o Reinforcement Learning en inglés) no tenemos una “etiqueta de salida”, por lo que no es de tipo supervisado y si bien estos algoritmos aprenden por sí mismos, tampoco son de tipo no supervisado, en donde se intenta clasificar grupos teniendo en cuenta alguna distancia entre muestras.

Los problemas de machine learning supervisados y no supervisados son específicos de un caso de negocio en particular, sea de clasificación o predicción, están muy delimitados, por ejemplo, clasificar “perros o gatos“, o agrupar “k=5” clústeres. En contraste, en el mundo real contamos con múltiples variables que por lo general se interrelacionan y que dependen de otros casos de negocio y dan lugar a escenarios más grandes en donde tomar decisiones. Una solución sería tener múltiples máquinas de ML supervisadas y que interactúan entre si -y esto no estaría mal- o podemos cambiar el enfoque. Y ahí aparece el Reinforcement Learning (RL) como una alternativa, tal vez de las más ambiciosas en las que se intenta integrar el Machine Learning en el mundo real, sobre todo aplicado a robots y maquinaria industrial. El Reinforcement Learning entonces, intentará hacer aprender a la máquina basándose en un esquema de “premios y castigos” en un entorno en donde hay que tomar acciones y que está afectado por múltiples variables que cambian con el tiempo.

Diferencias con los clásicos

En los modelos de aprendizaje supervisado (o no supervisado) como redes neuronales, árboles, KNN, etc. Se intenta “minimizar la función coste”, reducir el error. En cambio, en el aprendizaje por refuerzo se intenta “maximizar la recompensa“. Y esto puede ser, a pesar de a veces cometer errores o de no ser óptimos.

Componentes del aprendizaje por refuerzo (Reinforcement Learning)

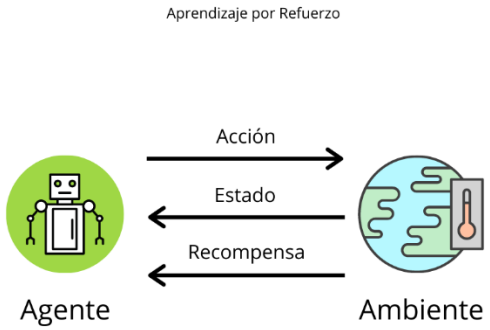
El Reinforcement Learning propone un nuevo enfoque para hacer que la máquina aprenda, para ello, postula los siguientes 2 componentes:

- El agente: será el modelo que queremos entrenar y que aprenda a tomar decisiones.
- Ambiente: será el entorno en donde interactúa y “se mueve” el agente. El ambiente contiene las limitaciones y reglas posibles a cada momento.

Entre ellos hay una relación que se retroalimenta y cuenta con los siguientes nexos:

- Acción: las posibles acciones que puede tomar en un momento determinado el agente.
- Estado (del ambiente): son los indicadores del ambiente de cómo están los diversos elementos que lo componen en ese momento.
- Recompensas (o castigos): a raíz de cada acción tomada por el agente, podremos obtener un premio o una penalización que orientarán al agente en si lo está haciendo bien o mal.

Entonces, la imagen final queda así:



En un primer momento, el agente recibe un estado inicial y toma una acción con lo cual influye é interviene en el ambiente. Y esa decisión tendrá sus consecuencias: en la siguiente iteración el ambiente devolverá al agente el nuevo estado y la recompensa obtenida. Si la recompensa es positiva estaremos reforzando ese comportamiento para el futuro. En cambio, si la recompensa es negativa lo estaremos penalizando, para que ante la misma situación el agente actúe de manera distinta. El esquema en el que se apoya el Reinforcement Learning es en el de Proceso de Decisión de Markov.

Casos de uso del aprendizaje por refuerzo

El aprendizaje por refuerzo puede ser usado en robots, por ejemplo, en brazos mecánicos en donde en vez de enseñar instrucción por instrucción a moverse, podemos dejar que haga intentos a ciegas e ir recompensando cuando lo hace bien.

También puede usarse en ambientes que interactúan con el mundo real, como en otro tipo de maquinaria industrial y para el mantenimiento predictivo, pero también en el ambiente financiero, por ejemplo, para decidir cómo conformar una cartera de inversión sin intervención humana. Otro caso de uso que está ganando terreno es el de usar RL para crear webs personalizadas para cada internauta.

Los videojuegos suelen ser ejemplos del uso de RL porque los videojuegos son un entorno ya programado en el que se está simulando un ambiente y en el que ocurren eventos a la vez. Por lo general el jugador es el agente que debe decidir qué movimientos hacer.

¿Cómo funciona el aprendizaje por refuerzo?

Premios y castigos

Al principio de todo, nuestro agente está en blanco, es decir, no sabe nada de nada de lo que tiene que hacer ni de cómo comportarse. Entonces podemos pensar en que tomará una de las posibles acciones aleatoriamente e irá recibiendo pistas de si lo está haciendo bien o mal en base a las recompensas. Entonces irá “tomando nota”, esto bien, esto mal. Una recompensa para un humano es algún estímulo que le de placer. Podría ser un aumento de sueldo, chocolate, una buena noticia. Para nuestro modelo de ML la recompensa es sencillamente un score: un valor numérico. Supongamos que la acción “A” recompensa con 100 puntos. El agente podría pensar “genial, voy a elegir A nuevamente para obtener 100 puntos” y puede que el algoritmo se estanque en una única acción y nunca logre concretar el objetivo global que queremos lograr. Es decir que tenemos que lograr un equilibrio entre “explorar lo desconocido y explotar los recursos” en el ambiente. Eso es conocido como el dilema de exploración/explotación. El agente explorará el ambiente e irá aprendiendo “cómo moverse” y cómo ganar recompensas (y evitar las penalizaciones). Al final almacenará el conocimiento en unas normas también llamadas “políticas“. Pero… es posible decir que es probable que el agente “muera” o pierda la partida las primeras… ¿mil veces? Con esto nos referimos a que deberemos entrenar miles y miles de veces al agente para que cometa errores y aciertos y pueda crear sus políticas hasta ser un buen agente.

Fuerza bruta

La realidad es que, para hacerle aprender a un coche autónomo a conducir, debemos hacerlo chocar, acelerar, conducir contramano y cometer todo tipo de infracciones para decirle “eso está mal, te quito los puntos” y para ello, hay que hacer que ejecute miles y miles de veces en un entorno de simulado. Esto tiene un lado bueno y uno malo. El malo ya lo vemos; tenemos que usar la fuerza bruta para que aprenda. Lo bueno es que contamos con equipos muy potentes que nos posibilitan realizar esto. Por otra parte, recordemos que estamos apuntando a un caso de uso mucho más grande y ambicioso que el de “sólo distinguir entre perritos y gatitos”.

Q-Learning, el algoritmo más usado

Ahora comentando uno de los modelos usados en Reinforcement Learning para poder concretar un ejemplo de su implementación. Es el llamado “Q-Learning”.

Repasemos los elementos que tenemos:

- Políticas: Es una tabla (aunque puede tener n-dimensiones) que le indicará al modelo “como actuar” en cada estado.
- Acciones: las diversas elecciones que puede hacer el agente en cada estado.
- Recompensas: si sumamos o restamos puntaje con la acción tomada.
- Comportamiento “avaro” (greedy en inglés) del agente. Es decir, si se dejará llevar por grandes recompensas inmediatas, o irá explorando y valorando las riquezas a largo plazo.

El objetivo principal al entrenar nuestro modelo a través de las simulaciones será ir “rellenando” la tabla de políticas de manera que las decisiones que vaya tomando nuestro agente obtengan “la mayor recompensa” a la vez que avanzamos y no nos quedamos estancados, es decir, pudiendo cumplir el objetivo global (o final) que deseamos alcanzar.

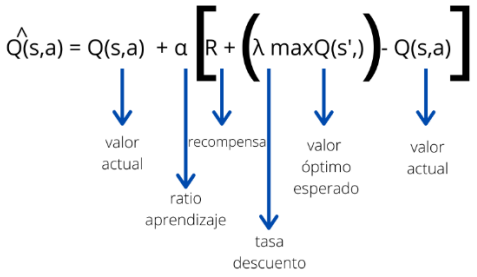
A la política la llamaremos “Q” por lo que:

Q(estado, acción) nos indicará el valor de la política para un estado y una acción determinados.

Y para saber cómo ir completando la tabla de políticas nos valemos de la ecuación de Bellman.

Ecuación de Bellman

La ecuación matemática es:



En resumen, lo que explica la ecuación es cómo ir actualizando las políticas Q^(s,a) en base al valor actual más una futura recompensa que se recibirá, en caso de tomar dicha acción. Hay dos ratios que afectan a la manera en que influye esa recompensa: el ratio de aprendizaje, que regula la velocidad en la que se aprende, y la tasa de descuento que tendrá en cuenta la recompensa a corto o largo plazo.

Ejercicio Python de aprendizaje por refuerzo: Pong con Matplotlib

El agente será el “player 1” y sus acciones posible son 2:

1. mover hacia arriba
2. mover hacia abajo

Y las reglas del juego:

- El agente tiene 3 vidas.
- Si pierde… castigo, restamos 10 puntos.
- Cada vez que le demos a la bola, recompensa, sumamos 10.
- Para que no quede jugando por siempre, limitaremos el juego a
    - 3000 iteraciones máximo o alcanzar 1000 puntos y habremos ganado.

```
import numpy as np
import matplotlib.pyplot as plt
from random import randint
from time import sleep
from IPython.display import clear_output
from math import ceil,floor

%matplotlib inline
```

<u>La clase Agente</u>

Dentro de la clase Agente encontraremos la tabla donde iremos almacenando las políticas. En nuestro caso la tabla cuenta de 3 coordenadas:

1. La posición actual del jugador.
2. La posición “y” de la pelota.
3. La posición en el eje “x” de la pelota.

Además, en esta clase, definiremos el factor de descuento, el learning rate y el ratio de exploración.

Los métodos más importantes:

- get_next_step() decide la siguiente acción a tomar en base al ratio de exploración si tomar “el mejor paso” que tuviéramos almacenado ó tomar un paso al azar, dando posibilidad a explorar el ambiente.
- update() aquí se actualizan las políticas mediante la ecuación de Bellman que vimos anteriormente. Es su implementación en python.

```python
class PongAgent:

    def __init__(self, game, policy=None, discount_factor = 0.1, learning_rate = 0.1, ratio_explotacion = 0.9):

        # Creamos la tabla de politicas
        if policy is not None:
            self._q_table = policy
        else:
            position = list(game.positions_space.shape)
            position.append(len(game.action_space))
            self._q_table = np.zeros(position)

        self.discount_factor = discount_factor
        self.learning_rate = learning_rate
        self.ratio_explotacion = ratio_explotacion

    def get_next_step(self, state, game):

        # Damos un paso aleatorio...
        next_step = np.random.choice(list(game.action_space))

        # o tomaremos el mejor paso...
        if np.random.uniform() <= self.ratio_explotacion:
            # tomar el maximo
            idx_action = np.random.choice(np.flatnonzero(
                    self._q_table[state[0],state[1],state[2]] == self._q_table[state[0],state[1],state[2]].max()
                ))
            next_step = list(game.action_space)[idx_action]

        return next_step

    # actualizamos las politicas con las recompensas obtenidas
    def update(self, game, old_state, action_taken, reward_action_taken, new_state, reached_end):
        idx_action_taken =list(game.action_space).index(action_taken)

        actual_q_value_options = self._q_table[old_state[0], old_state[1], old_state[2]]
        actual_q_value = actual_q_value_options[idx_action_taken]

        future_q_value_options = self._q_table[new_state[0], new_state[1], new_state[2]]
        future_max_q_value = reward_action_taken  +  self.discount_factor*future_q_value_options.max()
        if reached_end:
            future_max_q_value = reward_action_taken #maximum reward

        self._q_table[old_state[0], old_state[1], old_state[2], idx_action_taken] = actual_q_value + \
                                            self.learning_rate*(future_max_q_value -actual_q_value)

    def print_policy(self):
        for row in np.round(self._q_table,1):
            for column in row:
                print('[', end='')
                for value in column:
                    print(str(value).zfill(5), end=' ')
                print('] ', end='')
            print('')

    def get_policy(self):
        return self._q_table
```

La clase Environment

En la clase de Ambiente encontramos implementada la lógica y control del juego del pong. Se controla que la pelotita rebote, que no se salga de la pantalla y se encuentran los métodos para graficar y animar en matplotlib. Por defecto se define una pantalla de 40 pixeles x 50 pixeles de alto y si utilizamos la variable “movimiento_px = 5” nos quedará definida nuestra tabla de políticas en 8 de alto y 10 de ancho (por hacer 40/5=8 y 50/5=10). Estos valores se pueden modificar a gusto.

Además, muy importante, tenemos el control de cuándo dar las recompensas y penalizaciones, al perder cada vida y detectar si el juego a terminado.

```
class PongEnvironment:

    def __init__(self, max_life=3, height_px = 40, width_px = 50, movimiento_px = 3):

        self.action_space = ['Arriba','Abajo']

        self._step_penalization = 0

        self.state = [0,0,0]

        self.total_reward = 0

        self.dx = movimiento_px
        self.dy = movimiento_px

        filas = ceil(height_px/movimiento_px)
        columnas = ceil(width_px/movimiento_px)

        self.positions_space = np.array([[[0 for z in range(columnas)]
                                                for y in range(filas)]
                                                   for x in range(filas)])

        self.lives = max_life
        self.max_life=max_life

        self.x = randint(int(width_px/2), width_px)
        self.y = randint(0, height_px-10)

        self.player_alto = int(height_px/4)

        self.player1 = self.player_alto  # posic. inicial del player

        self.score = 0

        self.width_px = width_px
        self.height_px = height_px
        self.radio = 2.5

    def reset(self):
        self.total_reward = 0
        self.state = [0,0,0]
        self.lives = self.max_life
        self.score = 0
        self.x = randint(int(self.width_px/2), self.width_px)
        self.y = randint(0, self.height_px-10)
        return self.state

    def step(self, action, animate=False):
        self._apply_action(action, animate)
        done = self.lives <=0 # final
        reward = self.score
        reward += self._step_penalization
        self.total_reward += reward
        return self.state, reward , done

    def _apply_action(self, action, animate=False):

        if action == "Arriba":
            self.player1 += abs(self.dy)

        elif action == "Abajo":
            self.player1 -= abs(self.dy)

        self.avanza_player()

        self.avanza_frame()

        if animate:
            clear_output(wait=True);
            fig = self.dibujar_frame()
            plt.show()

        self.state = (floor(self.player1/abs(self.dy))-2, floor(self.y/abs(self.dy))-2, floor(self.x/abs(self.dx))-2)

    def detectaColision(self, ball_y, player_y):
        if (player_y+self.player_alto >= (ball_y-self.radio)) and (player_y <= (ball_y+self.radio)):
            return True
        else:
            return False

    def avanza_player(self):
        if self.player1 + self.player_alto >= self.height_px:
            self.player1 = self.height_px - self.player_alto
        elif self.player1 <= -abs(self.dy):
            self.player1 = -abs(self.dy)

    def avanza_frame(self):
        self.x += self.dx
        self.y += self.dy
        if self.x <= 3 or self.x > self.width_px:
            self.dx = -self.dx
            if self.x <= 3:
                ret = self.detectaColision(self.y, self.player1)

                if ret:
                    self.score = 10
                else:
                    self.score = -10
                    self.lives -= 1
                    if self.lives>0:
                        self.x = randint(int(self.width_px/2), self.width_px)
                        self.y = randint(0, self.height_px-10)
                        self.dx = abs(self.dx)
                        self.dy = abs(self.dy)
        else:
            self.score = 0

        if self.y < 0 or self.y > self.height_px:
            self.dy = -self.dy

    def dibujar_frame(self):
        fig = plt.figure(figsize=(5, 4))
        a1 = plt.gca()
        circle = plt.Circle((self.x, self.y), self.radio, fc='slategray', ec="black")
        a1.set_ylim(-5, self.height_px+5)
        a1.set_xlim(-5, self.width_px+5)

        rectangle = plt.Rectangle((-5, self.player1), 5, self.player_alto, fc='gold', ec="none")

        a1.add_patch(circle);
        a1.add_patch(rectangle)
        #a1.set_yticklabels([]);a1.set_xticklabels([]);
        plt.text(4, self.height_px, "SCORE:"+str(self.total_reward)+"  LIFE:"+str(self.lives), fontsize=12)
        if self.lives <=0:
            plt.text(10, self.height_px-14, "GAME OVER", fontsize=16)
        elif self.total_reward >= 1000:
            plt.text(10, self.height_px-14, "YOU WIN!", fontsize=16)
        return fig
```

El juego: Simular miles de veces para enseñar

Finalmente definimos una función para jugar, donde indicamos la cantidad de veces que queremos iterar la simulación del juego e iremos almacenando algunas estadísticas sobre el comportamiento del agente, si mejora el puntaje con las iteraciones y el máximo puntaje alcanzado.

```
def play(rounds=5000, max_life=3, discount_factor = 0.1, learning_rate = 0.1,
         ratio_explotacion=0.9,learner=None, game=None, animate=False):

    if game is None:
        game = PongEnvironment(max_life=max_life, movimiento_px = 3)

    if learner is None:
        print("Begin new Train!")
        learner = PongAgent(game, discount_factor = discount_factor,learning_rate = learning_rate, ratio_explotacion= ratio_explotacion)

    max_points= -9999
    first_max_reached = 0
    total_rw=0
    steps=[]

    for played_games in range(0, rounds):
        state = game.reset()
        reward, done = None, None

        itera=0
        while (done != True) and (itera < 3000 and game.total_reward<=1000):
            old_state = np.array(state)
            next_action = learner.get_next_step(state, game)
            state, reward, done = game.step(next_action, animate=animate)
            if rounds > 1:
                learner.update(game, old_state, next_action, reward, state, done)
            itera+=1

        steps.append(itera)

        total_rw+=game.total_reward
        if game.total_reward > max_points:
            max_points=game.total_reward
            first_max_reached = played_games

        if played_games %500==0 and played_games >1 and not animate:
            print("-- Partidas[", played_games, "] Avg.Puntos[", int(total_rw/played_games),"]  AVG Steps[", int(np.array(steps).mean()), "] Max Score[", max_points,"]")

    if played_games>1:
        print('Partidas[',played_games,'] Avg.Puntos[',int(total_rw/played_games),'] Max score[', max_points,'] en partida[',first_max_reached,']')

    #learner.print_policy()

    return learner, game
```

Para entrenar ejecutamos la función con los siguientes parámetros:

- 6000 partidas jugará
- ratio de explotación: el 85% de las veces será avaro, pero el 15% elige acciones aleatorias, dando lugar a la exploración.
- learning rate = se suele dejar en el 10 por ciento como un valor razonable, dando lugar a las recompensas y permitiendo actualizar la importancia de cada acción poco a poco. Tras más iteraciones, mayor importancia tendrá esa acción.
- discount_factor = También se suele empezar con valor de 0.1 pero aquí utilizamos un valor del 0.2 para intentar indicar al algoritmo que nos interesa las recompensas a más largo plazo.

```
learner, game = play(rounds=6000, discount_factor = 0.2, learning_rate = 0.1, ratio_explotacion=0.85)
```

Y vemos la salida del entreno, luego de unos 2 minutos:

```
Begin new Train!
-- Partidas[ 500 ] Avg.Puntos[ -234 ]  AVG Steps[ 116 ] Max Score[ 10 ]
-- Partidas[ 1000 ] Avg.Puntos[ -224 ]  AVG Steps[ 133 ] Max Score[ 100 ]
-- Partidas[ 1500 ] Avg.Puntos[ -225 ]  AVG Steps[ 134 ] Max Score[ 230 ]
-- Partidas[ 2000 ] Avg.Puntos[ -223 ]  AVG Steps[ 138 ] Max Score[ 230 ]
-- Partidas[ 2500 ] Avg.Puntos[ -220 ]  AVG Steps[ 143 ] Max Score[ 230 ]
-- Partidas[ 3000 ] Avg.Puntos[ -220 ]  AVG Steps[ 145 ] Max Score[ 350 ]
-- Partidas[ 3500 ] Avg.Puntos[ -220 ]  AVG Steps[ 144 ] Max Score[ 350 ]
-- Partidas[ 4000 ] Avg.Puntos[ -217 ]  AVG Steps[ 150 ] Max Score[ 350 ]
-- Partidas[ 4500 ] Avg.Puntos[ -217 ]  AVG Steps[ 151 ] Max Score[ 410 ]
-- Partidas[ 5000 ] Avg.Puntos[ -216 ]  AVG Steps[ 153 ] Max Score[ 510 ]
-- Partidas[ 5500 ] Avg.Puntos[ -214 ]  AVG Steps[ 156 ] Max Score[ 510 ]
Partidas[ 5999 ] Avg.Puntos[ -214 ] Max score[ 510 ] en partida[ 5050 ]
```

En las salidas vemos sobre todo cómo va mejorando en la cantidad de “steps” que da el agente antes de perder la partida.

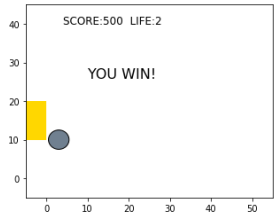
Observando el resultado

Ya contamos con nuestro agente entrenado, ahora veamos qué tal se comporta en una partida de pong, y lo podemos ver jugar, pasando el parámetro animate=True.

Antes de jugar, instanciamos un nuevo agente “learner2” que utilizará las políticas que creamos anteriormente. A este agente le seteamos el valor de explotación en 1, para evitar que tome pasos aleatorios.

```
learner2 = PongAgent(game, policy=learner.get_policy())
learner2.ratio_explotacion = 1.0  # con esto quitamos las elecciones aleatorias al jugar
player = play(rounds=1, learner=learner2, game=game, animate=True)
```

En este caso, con las 6 mil iteraciones de entrenamiento fue suficiente alcanzar los 500 puntos y ganar (se puede ir variando el objetivo a 500 puntos o a 1000, la cantidad de vidas, etc.)



La tabla de políticas resultante

En este ejemplo, se muestra una tabla de 3 coordenadas. La primera toma valores del 0 al 7 (posición del jugador), la segunda también 8 valores (altura de la bola de pong) y la tercera va del 0 al 9 con el desplazamiento horizontal de la pelota. Supongamos que el player está situado en la posición “de abajo de todo”, es decir, en la posición cero.

Dentro de esa posición queda conformada la siguiente tabla:

	x0		x1		x2		x3		x4		x5		x6		x7		x8		x9	
	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar
y7	-2.4	-0.2	-0.3	-0.	-0.1	-0.	-0.	-0.	-0.	-0.	-0.1	-0.1	-0.3	-0.2	-39.6	-27.3	-16.1	-24.4	-21.2	-16.6
y6	-4.	-0.9	-0.6	-0.1	-0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.1	-0.3	-9.7	-0.7	-15.7	-14.	-20.7	-45.9
y5	-5.4	-3.4	-0.4	-0.1	-0.	-0.	-0.	-0.	-0.	-0.	-0.1	-0.	-0.3	-0.	-0.7	-0.7	-17.4	-1.1	-53.2	-28.5
y4	-3.4	-1.4	-0.1	-0.4	-0.	0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.1	-0.	-1.9	-0.6	-8.5	-19.5	-46.	-16.7
y3	-0.4	-3.4	-0.	-0.1	-0.	-0.	-0.	0.	-0.	-0.	-0.	-0.	-0.	-0.1	-23.3	-5.3	-12.4	-4.1	-2.4	-28.6
y2	0.4	-1.1	0.	-0.1	-0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.1	-0.4	-2.7	-14.4	1.9	-9.	3.8	-16.6
y1	-1.8	-2.2	-0.3	-0.1	-0.	-0.	-0.	0.	-0.	-0.	-0.	-0.1	-0.	-0.1	-17.6	-29.	-9.8	-4.1	-39.3	5.7
y0	-2.8	-0.7	-0.3	-0.1	-0.	0.	-0.	-0.	-0.	-0.	-0.1	-0.	-0.	-0.2	-5.7	-14.4	-16.5	0.1	-35.9	2.7

Si nos fijamos en la coordenada de la bola (x8, y1) vemos los valores 1.9 para subir y -9 para bajar. Claramente la recompensa mayor está en la acción de subir. Pero si la pelotita estuviera en (x9,y4) la mejor acción será Bajar, aunque tenga un puntaje negativo de -16,7 será mejor que restar 46.

Conclusiones

En conclusión, comprender el funcionamiento del aprendizaje por refuerzo es fundamental para aprovechar su capacidad de optimización y toma de decisiones en machine learning. Esta metodología ofrece ventajas específicas para resolver problemas complejos y la correcta aplicación de estos algoritmos no solo permite descubrir rutas óptimas para alcanzar objetivos a largo plazo, sino que también resulta clave en áreas como la personalización de marketing, la optimización de recursos y la predicción financiera que se hablaba al inicio de este trabajo.

Un resumen de los conceptos esenciales ayuda a estructurar y clarificar los fundamentos del aprendizaje por refuerzo, facilitando una comprensión más profunda de temas como el paradigma de recompensas y castigos, el enfoque de fuerza bruta, el Q-Learning, las diferencias entre este tipo de algoritmos con los clásicos de machine learning, cómo se aplica este tipo de aprendizaje en los videojuegos, entre otros. Al simplificar el tema en puntos clave, se destacan los aspectos más relevantes y se facilita su aplicación efectiva en escenarios prácticos.

Por tanto, este análisis subraya la relevancia del aprendizaje por refuerzo en el análisis de datos y la toma de decisiones, resaltando la importancia de haber realizado un resumen para organizar la información de manera clara. Esto facilita una comprensión precisa de los principios fundamentales y permite enfrentar problemas complejos en distintos contextos con una base sólida.

Referencias

- AWS. (s.f.). *¿Qué es el aprendizaje mediante refuerzo?*. Amazon Web Services. Recuperado el 28 de Octubre de 2024 de: https://aws.amazon.com/es/what-is/reinforcement-learning/
- Na8. (2020, 24 de diciembre). *Aprendizaje por Refuerzo*. Aprende Machine Learning en Español. Recuperado el 28 de Octubre de 2024 de: https://www.aprendemachinelearning.com/aprendizaje-por-refuerzo/?authuser=0