Reinforcement Learning: Combates de Pokémon

Antecedentes, implementación y resultados

Alejandro Jiménez, Alejandro Gómez, Luis Crespo y Antonio Cabrera

Trabajo para el doble grado de
Ingeniería del Software y Matemática Computacional

Asignatura de inteligencia artifial
U-tad
España
Enero 2024

Contents

List of Figures

Chapter 1

Introducción

1.1 Contexto

Desde que Deep Blue gana a Kasparov en 1997, las máquinas han ido superando a los humanos en la mayoría de juegos clásicos. Con los avances en computación y en Deep Learning la diferencia entre los grandes maestros y los ordenadores es más grande que nunca. La nueva generación de bots para jugar a juegos hace uso del Reinforcement Learning, que consiste en entrenar a una red neuronal en base a lo que aprende jugando contra ella misma.

Pokémon es la franquicia de entretenimiento más grande del mundo, con más de 300 millones de juegos vendidos. El juego consiste en capturar criaturas llamadas Pokémon y entrenarlos para que luchen contra otros Pokémon. A pesar de que el juego fue pensado para niños, Pokémon cuenta con una escena competitiva muy profesional (el torneo mundial de 2024 contará con 2.000.000 de euros en premios), sin embargo el uso de inteligencias artificiales aun no se ha extendido como en otros juegos.

1.2 Objetivos

El objetivo de este trabajo es crear un bot que sea capaz de jugar a Pokémon de forma autónoma. Para ello se ha utilizado una versión simplificada del juego programada en python, en la que se eliminan ciertos elementos como el azar en la partida. El bot se ha entrenado utilizando el algoritmo de Reinforcement Q-Learning , haciendo uso de la librería de PyTorch.

Chapter 2

Explicación del juego

A continuación vamos a explicar de forma muy resumida las mecánicas del juego:

2.1 Turnos

El juego se basa en batallas entre dos jugadores, cada uno cuenta con un equipo de 6 pokémon, de los cuales solo puede tener uno en el campo de batalla.

Las batallas se dividen en turnos, en cada turno el jugador puede realizar una acción, ya sea atacar, cambiar de pokémon. Al comienzo de cada turno ambos jugadores selecciona una acción, en el caso de que ambos decidan atacar atacará primero el pokémon con más velocidad.

Figure 2.1: Jugador eligiendo el ataque de su pokémon.

2.2 Pokémons

Las batallas se realizan entre pokémons, en la actualidad existen 1025 especies de pokémons. Nosotros trabajaremos con una muestra reducida para facilitar el entrenamiento de la red neuronal.

Figure 2.2: Algunos de los pokémons.

2.3 Tipos

Cada pokémon tiene un tipo, existen 18 tipos diferentes, cada tipo tiene una serie de debilidades y fortalezas contra otros tipos. Además del tipo los pokémons tienen otras estadísticas como los puntos de vida, ataque, defensa o velocidad.

Figure 2.3: Tabla de tipos.

2.4 Movimientos

Cada pokémon tiene una serie de movimientos, los cuales realizan una acción en la batalla, ya sea atacar, curar, aumentar la defensa, etc. Cada movimiento tiene un tipo, una potencia y una precisión. El tipo determina contra que pokémons es eficaz el ataque, la potencia influye en el daño del ataque y la precisión determina la probabilidad de acierto (que en nuestro caso siempre será 100%). A día de hoy existen 934 movimientos, la mayoría consisten en atacar pero hay algunas excepciones.

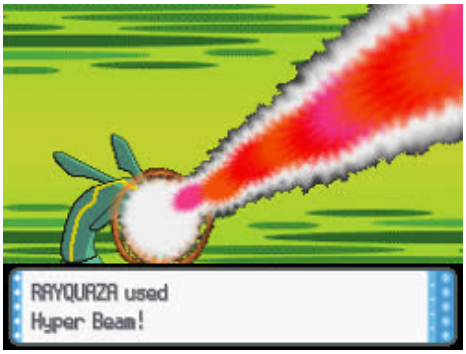

Figure 2.4: Un pokémon usando el movimiento hiperrayo.

Chapter 3

Reinforcement Learning

A continuación, se va a explicar brevemente en qué consiste el Reinforcement Learning (a partir de ahora RL), además de explicar cómo se aplicará este area del aprendizaje automático a nuestro trabajo.

El RL es un paradigma de aprendizaje automático que se centra en enseñar a un agente a tomar decisiones secuenciales para maximizar una recompensa acumulativa a lo largo del tiempo. En este enfoque de aprendizaje automático, el agente interactúa con un entorno dinámico y aprende a base de las recompensas y penalizaciones que conllevan sus decisiones.

3.1 Reinforcement Learning aplicado a Pokémon

Como en este trabajo estamos llevando a cabo batallas pokemon, el agente tiene un rango de 4 opciones a la hora de realizar una accion. Estas opciones son los 4 movimientos que puede efectuar el pokemon que está en ese momento luchando. La forma en la que se entregarán recompensas y penalizaciones al agente será en base de si estos movimientos inflingen daño (quitan puntos de vida del pokemon al que se enfrenta), o si gana el combate. De esta forma el agente aprenderá qué movimientos son los óptimos para los distintos pokemons.

El agente también posee un estado, en relación con el entorno de este trabajo. Este estado incluirá valores como, por ejemplo, el tipo o tipos de Pokémon que el agente maneja y el Pokémon al que se enfrenta, las puntuaciones de vida de los Pokémon y los movimientos disponibles de los Pokémon.

Para elegir la accion que realizará el Pokémon, el modelo que se creará tendrá que recibir como inputs, todos los valores del estado comentados previamente y devolvera un output de 4 valores correspondientes a los 4 posibles movimientos del Pokémon controlado por el agente, de los cuales se elegirá el que tenga un mayor valor.

3.2 Deep Q-Learning

Este trabajo tambien va a estar centrado en Deep Q-Learning, un enfoque en el campo del deep learning aplicado a la toma de decisiones en entornos dinámicos como el que estamos usando. Q-Learning es un algoritmo que buscan aprender una función Q con la que evaluar la calidad de las acciones en un estado dado.

Para esto vamos a tener que implementar una red neuronal profunda para evitar utilizar tablas con todas las posibles combinaciones de acciones y estados, que en este caso serían tablas de proporciones demasiado grandes teniendo en cuenta todos los movimientos, pokemons y tipos que se han comentado anteriormente.

La función Q es una función que asigna un valor para cada par estado-acción y se denota como Q(s,a), donde s respresenta el estado y a la acción. Esta función representa el valor esperado de tomar una acción en un estado especfíco, es decir, mide cuánto valor acumulativo (de recompensas) se espera obtener de realizar dicha acción es ese estado, y de esta forma saber cuál es la estrategia que seguir de ahí en adelante.

El Q-Learning presenta un algoritmo para entrenar al agente en el entorno propuesto, y sigue los siguientes pasos:

1. Inicialización
   - Inicializa la función Q(s,a) de forma arbitraria para todos los pares estado- acción.
   - Establece los parámetros del algoritmo: tasa de aprendizaje, factor de descuento y otros que se explicarán más adelante.
2. Exploración/ Explotación (Realizar acciones)
   - En cada frame o etapa del proceso, se elige una acción a realizar en el estado actual. Esta decisión puede ser determinista o estocástica, la primera se basa en elegir la acción que tiene el mayor valor Q conocido (explotacion), mientras que la segunda busca realizar acciones nuievas para descubrir como de efectivas o útiles son (exploración).
3. Interaccion con el entorno
   - Una vez la acción ha sido realizada, se observa la recompensa o penalización recibida y el próximo estado.
4. Actualización de la función Q
   - Se usa la ecuación de Bellman para actualizar Q(s,a):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \Big[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a) \Big] \tag{3.1}$$

   Donde:

- (alpha) es la tasa de aprendizaje que controla la magnitud de la actualización.
- r es la recompensa obtenida
- gamma como el factor de descuento que penaliza las recompensas a largo plazo.
- max_a'Q(s',a') representa el valor máximo esperado en el proximo estado s'

5. Repetición
    - Se realizan los pasos del 2 al 4 durante un número de frames o iteraciones.
6. Convergencia
    - La función Q converge a la función de valor óptimo a medida que el agente explora y aprende más sobre el entorno.
7. Política óptima
    - La política óptima se obtiene a partir de la función Q resultante, la acción óptima es aquella que maximiza Q(s,a).

Este algoritmo es un proceso iterativo en el que el agente interactúa con el entorno y cambia sus estimaciones de Q, de esta forma creando una política o estrategia para maximizar las recompensas. Un equilibrio entre la exploración y la explotación es de gran importancia para asegurar que el agente conoce las mejores acciones a realizar sin atascarse en comportamientos subóptimos.

Chapter 4

IA Pokemon

Habiendo establecido el funcionamiento del RL y del Deep Q-Learning, se explicará como se aplican al entrenamiento de una IA capaz de realizar batallas pokemon.

4.1 Entorno

El entorno del que se ha hablado anteriormente tiene un rol de suma importancia pues al programar este en la clase "environment.py", se le ha añadido la función "step()". Esta funcion es la encargada de dirigir las batallas pokemon pues elige el movimiento del pokemon, ejecuta el turno y calcula las recompensas a impartir en los agentes, que son los dos contrincantes. Una vez se han calculado las recompensas respectivas al turno recién realizado, se ha de comprobar si el combate se ha terminado, es decir, si uno de los pokemons ha sido derrotado, en caso positivo se reparten las recompensas acordes a ganar el combate y finalmente se devuelven las recompesnsas de ambos jugadores, si el combate ha terminado, y el ganador de en caso de haberlo hecho.

4.2 Agente

El agente es la clase que comunica el entorno con el modelo de dos formas: con memoria corta, encargada del entrenamiento entre turno, y la memoria larga, la encargada del entrenamiento en base a todos los datos que se han ido recopilando a lo largo del entrenamiento. Para esto se usa una estructura de datos "deque" o "double ended queue", en español, cola doblemente terminada.

En este trabajo, este tipo de cola resulta mas eficiente ya que permite realizar cambios a ambos lados de la cola mientras que las colas convencionales solo son manipulables por un extremo, y en este trabajo estamos constantemente accediendo a datos de ambos lados de la cola, uno cuando borramos elementos de la cola y otro para añadir nuevos. Con deque es posible hacer esto con una

complejidad temporal de solo O(1) mientras que con una cola normal sería de O(n).

En esta clase tambien tenemos una funcion vital para el funcionamiento de esta inteligencia artifial, get_state(env). Esta funcion traduce el estado del juego actual a numeros naturales para mandarselo a la red neuronal y que esta interprete la información. En nuestro caso el estado es un array de numpy de enteros de 16 valores en los que se guardan el id de cada pokémon, los puntos de vida de cada pokemon y los 4 movimientos de cada uno en forma de id numerica. Como se ha utilizado una muestra reducida de los pokémon, con el id la IA puede hacerse a la idea de como es cada pokémon, pero con muestras más grandes habría que añadir más datos como pueden ser la velocidad, el tipo o la defensa de cada pokémon. Hemos implementado dos versiones del agente:

4.2.1 Agente local

El agente local está diseñado para entrenar un combate especifico, en los que los pokémons siempre son los mismos. De esta forma, conseguimos resultados óptimos en periodos de entrenamiento más cortos. La desventaja de este agente es que no se adapta bien a otros combates.

4.2.2 Agente global

El agente global está diseñado para entrenar un combate genérico, en los que los pokémons pueden ser cualquiera de los disponibles en nuestra muestra. Al contrario que el agente local, la duración del entrenamiento es mayor ya que para que aprenda tiene que pasar por muchas más épocas y además los resultados son inferiores.

4.3 Modelo

El modelo es la red neuronal que se encarga de interpretar los datos que le llegan del agente y devolver una acción. En nuestro caso, el modelo es una red neuronal con una capa oculta lineal. El modelo tiene una capa de input de 12, que es igual al tamaño del estado del juego y la capa de salida es igual a 4 que son las acciones que cada jugador puede realizar por turno. Hemos decidido utilizar la libreria de pytorch en vez de keras, ya que hemos visto que entrena más rápido.

4.4 Entrenamiento

El entrenamiento de la IA se ha realizado en dos fases, la primera de ellas es el entrenamiento de la memoria corta, en la que se entrena el modelo con los datos del último turno, y la segunda es el entrenamiento de la memoria larga, en la que se entrena el modelo con todos los datos recopilados hasta el momento. En

nuestra implementación, guardamos el modelo de cada jugador en el punto que mejor media de accurac y tiene en las últimas 100 partidas.

```
    def train(pokemon1:str, pokemon2:str, epochs:int) -> None:
    agent = Agent()
    env = Environment(pokemon1, pokemon2)

    # variables para el record
    p1_wins = []
    p2_wins = []
    last_victories_p1 = 0
    last_victories_p2 = 0

    record_win_p1 = 0
    record_win_p2 = 0

    while True:
        # obtener estado antiguo
        state_old = agent.get_state(env)

        # obtener movimiento
        final_moves = agent.get_action(state_old)

        # traducir el movimiento [0, 0, 0, 1] -> 3
        rewards, done, winner  = env.step(np.argmax(final_moves[0]), np
    .argmax(final_moves[1]))
        state_new = agent.get_state(env)

        # entrenar al agente con el nuevo estado (short memory)
        agent.train_short_memory(state=state_old, actions=final_moves,
    rewards=rewards, next_state=state_new, done=done)

        # guardar en la memoria del agente el estado, la accion, la
    recompensa, el siguiente estado y si el juego ha terminado
        agent.remember(state_old, final_moves, rewards, state_new, done
    )

        if done:
            # entrenar al agente con todos los estados (long memory),
    resetear el juego y actualizar el record
            env = Environment(pokemon1, pokemon2)
            agent.numero_partidas += 1
            agent.train_long_memory()

            if winner == 0:
                last_victories_p1 += 1
            elif winner == 1:
                last_victories_p2 += 1

            if(agent.numero_partidas % 100 == 0):
                p1_wins.append(last_victories_p1 / 100)
                p2_wins.append(last_victories_p2 / 100)

                if record_win_p1 < p1_wins[-1]:
                    record_win_p1 = p1_wins[-1]
                    agent.model_p1.save(file_name=f"[{pokemon1}]-vs-{
    pokemon2}.pth")
```

```
49
                if record_win_p2 < p2_wins[-1]:
51                  record_win_p2 = p2_wins[-1]
                    agent.model_p2.save(file_name=f"{pokemon2}-vs-[{
    pokemon1}].pth")
53
                last_victories_p1 = 0
55              last_victories_p2 = 0

57          if agent.numero_partidas == epochs:
                break
```

Listing 4.1: Entrenamiento de la IA

En la primera implementación del entrenamiento, el modelo entrenaba contra un bot aleatorio, pero en las versiones actuales del proyecto el agente tiene dos modelos de tal forma que ambos van aprendiendo a luchar entre ellos. En cuanto a los agentes, la diferencia entre el global y el local es que en el global los pokemons se cambian cada 100 partidas, para que así aprenda a luchar con diferentes oponentes.

Chapter 5

Conclusiones

Al realizar los dos entrenamientos, uno con el agente local y otro con el global, se puede ver que el entrenamiento del local ha sido más preciso que su contraparte.

A continuacion, se pueden ver dos gráficas del agente local:

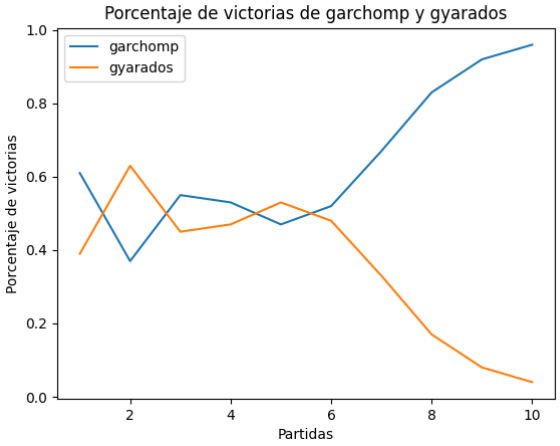

Figure 5.1: Combate entre Garchomp y Gyarados.

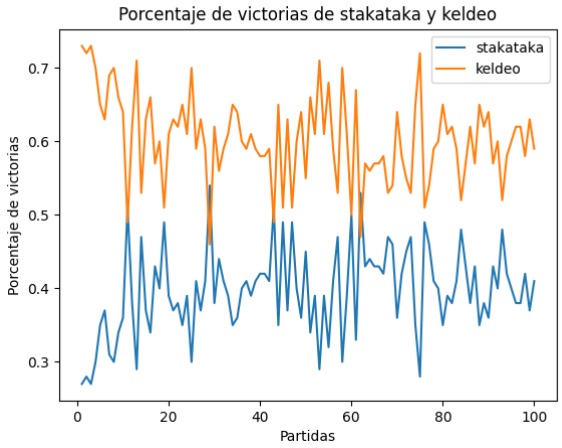

Figure 5.2: Combate entre Stakataka y Keldeo.

En ambas figuras se puede observar que la ia comienza eligiendo ataques de forma aleatoria hasta que encuentra uno que le permite ganar todas las partidas. De esta forma, se crea un balance entre ambos jugadores en el que uno gana siempre.

Hemos observado que al realizar el entrenamiento del agente global, existe un error que invierte el valor de las recompensas. Esto hace que el modelo tenga como objetivo perder. En la siguiente figura, la inteligencia artifial está jugando como el jugador 2, y vemos que solo tiene un porcentaje de victorias de aproximadamente 35

```
P1 wins: 64227, P1 losses: 35773 winrate: 0.64227
P2 wins: 35773, P2 losses: 64227 winrate: 0.35773
```

Figure 5.3: Combate entre Stakataka y Keldeo.

5.1 Posibles ampliaciones

Además de solucionar el error presentado anteriormente, otras de las posibles mejoras para el proyecto incluyen utilizar más datos en el estado del juego, como por ejemplo las estadísticas de defensa y velocidad del pokémon.

Una funcionalidad que no ha sido posible implementar es utilizar el motor gráfico del Pokémon Showdown para poder visualizar las batallas, e incluso probar los modelos contra jugadores reales online.