

Proyecto de Simulación e IA

Integrantes:

Hivan Cañizares Diaz
Franco Hernández Piloto
Carlos Mauricio Reyes Escudero

Universidad de La Habana
Curso 2024

Resumen

Este proyecto se centra en la simulación de combates Pokémon utilizando inteligencia artificial (IA) para evolucionar Pokémon y crear agentes de entrenadores. La primera fase utiliza algoritmos genéticos para evolucionar a los Pokémon a través de múltiples generaciones, seleccionando y reproduciendo los más exitosos. En la segunda fase, se desarrollan agentes de entrenadores basados en dos enfoques: el modelo BDI (Creencias, Deseos, Intenciones) y Modelos de Lenguaje de Gran Tamaño (LLMs). Estos agentes son evaluados en un entorno simulado para analizar su rendimiento en combate.

1. Introducción

Este proyecto tiene como objetivo simular combates Pokémon, utilizando técnicas de inteligencia artificial para analizar el rendimiento de diferentes estrategias y agentes en este entorno.

Para ello, desarrollamos dos partes:

1.1. Parte 1: Evolución mediante Algoritmos Genéticos

Utilizamos algoritmos genéticos para evolucionar a los diferentes Pokémon. Este proceso genera múltiples generaciones de Pokémon, seleccionando y reproduciendo los más exitosos. A través de este enfoque, se espera que los Pokémon evolucionados presenten características más fuertes y efectivas en combate.

1.2. Parte 2: Agentes de Entrenadores

En la segunda fase del proyecto, se desarrollan agentes de entrenadores utilizando dos enfoques distintos: el modelo BDI (Creencias, Deseos, Intenciones) y Modelos de Lenguaje de Gran Tamaño (LLMs).

2. Simulación de Combates

La simulación de combates se basa en una implementación existente de las mecánicas de batalla Pokémon [1], que ofrece una base sólida y precisa para las interacciones entre Pokémon durante el combate. La interacción con el simulador comienza cuando se inicia una batalla entre dos entrenadores, cada uno controlando un equipo de Pokémon.

Durante cada turno, se solicita a cada jugador que elija un movimiento para su Pokémon. Una vez que ambos jugadores han seleccionado sus acciones, se llama a la función `do_turn()`, que ejecuta los cálculos necesarios para resolver el turno. Esta función maneja los efectos de los movimientos, calcula el daño infligido y actualiza los estados de los Pokémon involucrados en la batalla. En caso de alinearse con los objetivos de la simulación, se calcula la experiencia obtenida y se asignan los niveles.

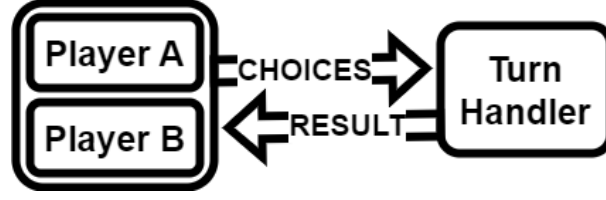


Figura 1: Flujo del combate.

A continuación, se presenta el pseudocódigo que describe el proceso de la simulación de combates:

Algorithm 1 Simulación de Combates

```

1: INICIALIZAR batalla CON entrenador1 Y entrenador2
2: while la batalla NO haya finalizado do
3:   ACCION1 ← solicitar_accion(entrenador1)
4:   ACCION2 ← solicitar_accion(entrenador2)
5:   do_turn()
6: end while
7: GANADOR ← obtener_ganador(entrenador1, entrenador2)
8: if GANADOR es entrenador1 then
9:   ASIGNAR experiencia(entrenador1)
10: else
11:   ASIGNAR experiencia(entrenador2)
12: end if

```

El sistema de experiencia y niveles se basa en datos actualizados de fuentes especializadas en Pokémon[3][4]. La fórmula para el cálculo de experiencia es:

$$\Delta EXP = \left(\frac{b \times L}{5} \times \frac{1}{s} \times \left(\frac{2L + 10}{L + L_p + 10} \right)^{2.5} + 1 \right) \times t \times e \times v \times f \times p$$

Figura 2: Fórmula de cálculo de experiencia en Pokémon

3. Evolución mediante Algoritmos Genéticos

En esta parte, se lleva a cabo un proceso de evolución en paralelo para los Pokémon y sus cerebros, que son redes neuronales evolucionadas utilizando ES-HyperNEAT (Evolvable-Substrate HyperNEAT). Para la evolución de las redes neuronales, se utiliza la biblioteca PUREPLES [2], que proporciona una implementación eficiente del algoritmo ES-HyperNEAT.

Se define la función de fitness (o aptitud) que se comparte entre un individuo Pokémon y su cerebro (red neuronal):

$$\text{fitness} = k_1 \left(\frac{\text{lvl}}{100} \right) + k_2 \left(\frac{\text{won_battles}}{\text{total_battles}} \right) \quad (1)$$

Donde:

- **lvl** es el nivel del Pokémon.
- **won_battles** es el número de batallas ganadas por el Pokémon.
- **total_battles** es el número total de batallas en las que ha participado.
- $k_i \in [0, 1]$, $\sum(k_i) = 1$

Al optimizar el fitness compartido, se asegura que tanto el Pokémon como su cerebro evolucionan de manera coordinada, lo que lleva a la formación de individuos altamente adaptados y capaces de tomar decisiones estratégicas en los combates.

3.1. Evolución de Especies Pokémon

Los Pokémon se representan como entidades que evolucionan a lo largo de generaciones, manteniendo información sobre los siguientes aspectos:

- **Especie:** Cada Pokémon pertenece a una especie específica que determina sus estadísticas base, como ataque, defensa y velocidad.
- **Nivel:** El nivel de un Pokémon varía entre 1 y 100. A medida que gana experiencia en combates, su nivel aumenta, lo que generalmente mejora sus estadísticas.
- **Experiencia:** La experiencia se acumula a través de combates y actividades, y es fundamental para el aumento de nivel y el desarrollo de habilidades del Pokémon.
- **Habilidad:** Cada especie de Pokémon puede tener una o varias habilidades que ofrecen ventajas únicas en combate, afectando su desempeño y estrategia.
- **Naturaleza:** La naturaleza de un Pokémon influye en el crecimiento de sus características, potenciando una estadística en un 10 % y reduciendo otra en un 10 %, excepto los Puntos de Salud (PS).
- **Movimientos:** Los Pokémon pueden aprender hasta cuatro movimientos a la vez, los cuales son esenciales para su desempeño en combate y pueden ser de ataque, defensa o apoyo.
- **Red Neuronal:** Red implementada utilizando ES-HyperNEAT.

A nivel de entidad, la evolución involucra los procesos:

- **Reproducción:** Se utiliza el entrecruzamiento uniforme (uniform crossover) para combinar los atributos de dos Pokémon progenitores.

Algorithm 2 Reproducción de dos Pokémon

```
1: function REPRODUCIR(padre, madre)
2:   nuevo_pokemon ← crear_nuevo_pokemon()
3:   movimientos_padre ← padre.movimientos
4:   movimientos_madre ← madre.movimientos
5:   nuevo_pokemon.movimientos ← []
6:   for movimiento en zip(movimientos_padre, movimientos_madre) do
7:     seleccionado ← seleccionar_aleatoriamente(movimiento[0], movimiento[1])
8:     nuevo_pokemon.movimientos.agregar(seleccionado)
9:   end for
10:  nuevo_pokemon.habilidad ← random_choose(padre.habilidad, madre.habilidad)
11:  nuevo_pokemon.naturaleza ← random_choose(padre.naturaleza, madre.naturaleza)
12:  nuevo_pokemon.neural_net ← random_choose(padre.neural_net, madre.neural_net)
   return nuevo_pokemon
13: end function
```

Mantenemos la posición de los movimientos puesto que la red neuronal retorna la posición del movimiento a efectuar, cambiarlo produciría inconsistencias.

- **Mutación:** Los movimientos, habilidades y naturalezas de los Pokémon pueden mutar. La tasa de mutación está controlada por un índice de temperatura que disminuye a medida que aumenta el fitness del Pokémon, siguiendo la fórmula:

$$\text{temperatura} = (1 - \text{fitness})^{\text{COOLER}}$$

donde COOLER es una constante que controla la velocidad de enfriamiento.

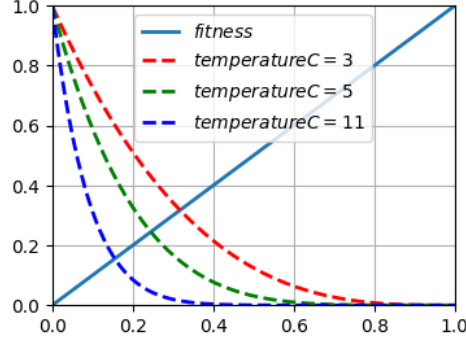


Figura 3: Relación entre el fitness y la temperatura para distintos valores de COOLER.

Algorithm 3 Mutación de un Pokémon

```

1: function MUTAR(pokemon)
2:    $U \leftarrow \text{generar\_aleatorio\_uniforme}()$ 
3:   while  $U < \text{pokemon.temperatura}$  do
4:      $\text{tipo\_mutacion} \leftarrow \text{random\_choose}([\text{movimientos}, \text{habilidades}, \text{naturaleza}], p=[0.5, 0.4, 0.1])$ 
5:      $A \leftarrow \text{obtener cambios disponibles según selección}$ 
6:      $\text{aplicar\_mutacion}(\text{pokemon}, \text{tipo\_mutacion}, A)$ 
7:      $U \leftarrow \text{generar\_aleatorio\_uniforme}()$ 
8:   end while
9: end function

```

3.2. Cerebros y ES-HyperNEAT

ES HyperNEAT es una extensión del algoritmo HyperNEAT, que a su vez se basa en NEAT (Neuro-Evolution of Augmenting Topologies). Este enfoque permite la evolución de redes neuronales artificiales a gran escala mediante la utilización de patrones geométricos y una codificación indirecta.

El funcionamiento de ES HyperNEAT se centra en la generación de redes neuronales complejas a partir de redes de patrones composicionales (CPPN). Estas CPPNs son funciones que generan patrones de pesos entre nodos, facilitando la creación de estructuras neuronales densas y organizadas. A diferencia de HyperNEAT, donde los nodos deben ser colocados manualmente, ES HyperNEAT permite que la posición de los nodos ocultos evolucione durante el entrenamiento, optimizando así la topología geométrica del sustrato.

La inicialización en ES HyperNEAT implica crear una población inicial de CPPNs que definen la estructura básica de las redes neuronales las cuales se utilizan para generar conexiones entre los nodos de la red.

Para nuestro proyecto se utiliza PUREPLES, que permite especificar diversos parámetros esenciales para la evolución de redes neuronales. A continuación, se detallan los aspectos clave:

- Profundidad máxima de la red: Se establece un límite en la cantidad de capas ocultas que puede tener la red neuronal, lo que influye en su capacidad para aprender patrones complejos.
- Funciones de activación: Se definen las funciones de activación que cada neurona utilizará, lo que afecta cómo se procesan las señales dentro de la red.
- Variaciones: Se especifican las variaciones en los parámetros de la red, permitiendo explorar diferentes configuraciones durante el proceso evolutivo.
- Rango de aceptación de un individuo poderoso: Se establece un umbral que determina qué tan aptos deben ser los individuos para ser considerados exitosos dentro de la población.
- Coordenadas de entrada y salida: Las entradas y salidas del sistema se especifican en coordenadas cartesianas, lo que facilita la representación y el procesamiento de datos en un espacio multidimensional.

- Conjunto de parámetros del algoritmo: Todos estos parámetros están incluidos en el archivo ‘config-cppn’, donde se configuran las características específicas del algoritmo.

La biblioteca utilizada lleva a cabo la mutación, reproducción y selección de las redes más aptas. Este proceso de selección funciona como una selección global tanto para las entidades como para las redes neuronales que constituyen el cerebro. Esto asegura que las mejores configuraciones sean preservadas y mejoradas a lo largo de las generaciones, optimizando así el rendimiento del sistema.

La IA que guía a cada Pokémon tiene entonces la siguiente estructura:

- **Entrada:** Estado del terreno y características de los Pokémon en combate.
- **Salida:** Movimiento a efectuar en el turno actual.
- **Evolución:** La topología de la red evoluciona a lo largo de las generaciones, adaptándose para mejorar la toma de decisiones en combate.

3.3. Descripción del Algoritmo Genético

El algoritmo general consta de dos partes principales, cada una diseñada para gestionar la evolución de los Pokémon de manera eficiente y efectiva.

- **Parte 1: Definición de una Función de Evaluación**

En esta primera parte, se define una función que evalúa una generación de Pokémon. Esta función recibe un conjunto de Pokémon que pueden ser nuevos o tener información acumulada de generaciones anteriores. Se utiliza un diccionario para determinar el estado de cada Pokémon y se toman decisiones basadas en su caso específico. A continuación, se presenta el pseudocódigo correspondiente:

Algorithm 4 Función de evaluación

```

1: function EVALUAR_GENERACION(pokemons, diccionario)
2:   for pokemon in pokemons do
3:     caso ← determinar_caso(pokemon, diccionario)
4:     if caso == nuevo then
5:       pokemon ← new Pokemon()
6:       pokemon.fitness ← 0.0
7:       diccionario[id] = pokemon
8:     else if caso == reproduccion then
9:       pokemon ← reproduce(parents)
10:      pokemon.fitness ← get_fitness(parents)
11:      diccionario[id] = pokemon
12:     else if caso == veterano then
13:       // No aplica cambios al fitness
14:     end if
15:     pokemon.nivel ← 0
16:     pokemon.experiencia ← 0
17:     mutar(pokemon)
18:   end for
19:   enfrentar(pokemons) // Realiza múltiples enfrentamientos
20:   actualizar_fitness(pokemons)
21:   return list_fitnesses(pokemons)
22: end function

```

- **Parte 2: Iteración hasta el Éxito o por N Generaciones Definidas**

La segunda parte del algoritmo se encarga de iterar hasta que se alcance un nivel de éxito definido o hasta que se completen N generaciones. Durante este proceso, se inicializa una población y se realizan evaluaciones continuas para seleccionar, mutar y reproducir Pokémon. A continuación, se muestra el pseudocódigo que ilustra este proceso:

Algorithm 5 Iteración hasta el éxito (o no)

```
1: function ITERAR_HASTA_EXITO(fitness_esperado, N)
2:   poblacion  $\leftarrow$  generar_poblacion(tamaño_fijo)
3:   fitness_máximo  $\leftarrow$  0.0
4:   while fitness_máximo < fitness_esperado y generaciones < N do
5:     fitnesses  $\leftarrow$  evaluar_generación(poblacion)
6:     fitness_máximo  $\leftarrow$  max(fitness_máximo, [fitness for fitness in fitnesses])
7:     //USAR PUREPLES para seleccionar, mutar y reproducir una nueva generacion
8:     poblacion  $\leftarrow$  PUREPLES.get_pop(poblacion)
9:   end while
10:  if exito then
11:    return mejor_pokemon()
12:  else
13:    return 'Final triste: no se alcanzó el fitness esperado.'
14:  end if
15: end function
```

3.4. Analizando monopolio de ataques

Para verificar la hipótesis de que los Pokémon tienden a especializarse en un único ataque poderoso, se implementa el siguiente proceso:

1. Recopilación de datos de uso de ataques para cada Pokémon evolucionado.
2. Cálculo de la distribución acumulativa empírica de uso de ataques.
3. Definición de una distribución teórica de uso equitativo de ataques.
4. Aplicación de la prueba de Kolmogorov-Smirnov de una muestra para comparar las distribuciones empírica y teórica.
5. Análisis de los resultados para determinar si hay una desviación significativa hacia el uso predominante de un solo ataque.

Este enfoque permite cuantificar objetivamente la tendencia de los Pokémon evolucionados a especializarse en estrategias de ataque único o diversificadas.

3.5. Experimentos sobre variabilidad de movimientos

Se entrenó un conjunto de 30 Pokémon hasta alcanzar un nivel de fitness adecuado, con los que se desarrolló una serie de combates que incluyeron enfrentamientos entre especies diferentes. Durante estos combates, se midió la cantidad de movimientos utilizados por cada Pokémon.

Venusaur	Charizard	Blastoise	Alakazam	Machop	Gengar
Gyarados	Snorlax	Dragonite	Mewtwo	Tyranitar	Blaziken
Gardevoir	Aggron	Manectric	Salamence	Metagross	Rayquaza
Garchomp	Lucario	Magnezone	Gallade	Excadrill	Chandelure
Hydreigon	Aegislash	Hawlucha	Decidueye	Kommo-o	Kartana

Cuadro 1: Lista de Pokémon seleccionados para los experimentos

Se llevaron a cabo dos entrenamientos distintos para analizar la variedad de ataques:

1. **Experimento sin énfasis en la variedad de movimientos:** En este primer experimento, la función de fitness no considera la variedad de movimientos utilizados por el Pokémon, lo que permite que la evolución se centre únicamente en la efectividad de los Pokémon en términos de nivel alcanzado y tasa de victorias.

2. **Experimento con énfasis decreciente en la variedad de movimientos:** En este segundo experimento, se amplía la función de fitness completa descrita anteriormente:

$$fitness_1 = p \cdot \left(\frac{\text{moves_used}}{4}\right)^{2,9} + (1 - p) \cdot fitness$$

donde:

- $p = \frac{3}{\text{age}+2,7}$ es un factor que disminuye con la edad del Pokémon.
- moves_used es el número de movimientos diferentes utilizados por el Pokémon.

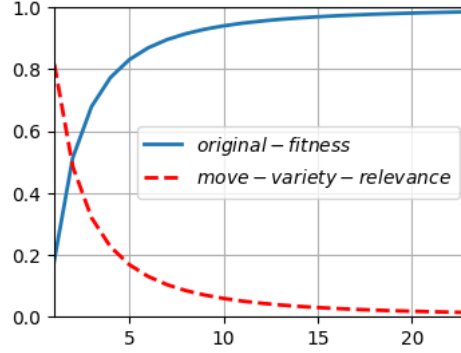


Figura 4: Relevancia de la variedad de movimientos y el fitness usual respecto a la edad.

Esta fórmula está diseñada para:

- Fomentar inicialmente la variedad de ataques usados.
- Reducir gradualmente la importancia de la variedad a medida que el Pokémon envejece.
- Balancear la importancia entre el nivel alcanzado y la tasa de victorias.

Estos dos experimentos nos permitirán comparar cómo la consideración de la variedad de movimientos afecta la evolución de las estrategias de los Pokémon. El primer experimento sirve como línea base, mientras que el segundo nos ayudará a entender si la presión inicial para usar una variedad de movimientos conduce eventualmente a estrategias más especializadas o si mantiene un conjunto más diverso de tácticas.

Los resultados de ambos experimentos se analizarán utilizando la prueba de Kolmogorov-Smirnov descrita anteriormente, lo que nos permitirá cuantificar y comparar la tendencia hacia la especialización en un único ataque poderoso en ambos escenarios.

3.6. Resultados

exp	dks	p-val	entropy	mean	variance
exp1	0.8200	0.0004	0.3251	1.10	0.09
exp2	0.8467	0.0002	0.8412	1.33	0.69

Cuadro 2: Resultados de los Experimentos de Uso de Movimientos

Los resultados obtenidos de ambos experimentos proporcionan información valiosa sobre el comportamiento de los Pokémon en combate. A continuación, se detalla el significado de cada estadística y su relevancia en el contexto del estudio:

- D de Kolmogorov-Smirnov: Los valores obtenidos (0.8200 para el Experimento 1 y 0.8467 para el Experimento 2) indican una diferencia significativa entre la distribución observada de uso de movimientos y una distribución uniforme. Esto sugiere que los Pokémon no utilizan sus movimientos de manera equitativa en ambos experimentos.
- p-valor: Los p-valores muy bajos (0.0004 y 0.0002) proporcionan evidencia estadística fuerte para rechazar la hipótesis nula de que los movimientos se utilizan de manera uniforme. Esto indica que los Pokémon tienden a especializarse en ciertos movimientos.

- Entropía de Shannon: La entropía de Shannon aumentó de 0.3251 en el Experimento 1 a 0.8412 en el Experimento 2, lo que sugiere una mayor diversidad en el uso de movimientos en el segundo experimento. Sin embargo, el valor de entropía sigue siendo relativamente bajo, indicando que, en general, los Pokémon aún utilizan un número limitado de movimientos.
- Promedio de movimientos usados: El promedio de movimientos utilizados aumentó de 1.10 a 1.33, lo que sugiere que al introducir un énfasis decreciente en la variedad de movimientos, los Pokémon comenzaron a usar un poco más de movimientos diferentes, aunque todavía se especializan en un solo ataque.
- Varianza de movimientos usados: La varianza también aumentó de 0.09 a 0.69, lo que indica una mayor dispersión en el uso de movimientos. Esto sugiere que algunos Pokémon están utilizando más movimientos que otros, lo que podría ser un resultado de la estrategia de combate más flexible implementada en el segundo experimento.

Estos hallazgos refuerzan la idea de que los Pokémon tienden a especializarse en ciertos movimientos, pero también indican que al permitir una mayor variedad de movimientos, existe la posibilidad de diversificación en sus estrategias de combate.

4. Implementación de Agentes para Entrenadores

En esta parte del proyecto, se implementan agentes para entrenadores que gestionan múltiples Pokémon, permitiendo tanto el uso de ataques como el cambio entre ellos. Estos agentes están diseñados para tomar decisiones estratégicas durante los combates, optimizando su rendimiento en función de las condiciones del entorno.

4.1. Implementación del Agente BDI

El agente BDI se implementa utilizando un motor de inferencia lógica llamado PyReason [6].

4.1.1. PyReason como Motor de Inferencia

PyReason opera como un motor de inferencia, utilizando grafos para modelar relaciones y propiedades. A continuación se detallan sus características fundamentales:

- Funcionamiento sobre Grafos: PyReason utiliza la biblioteca NetworkX para representar grafos. En este contexto, los arcos del grafo son modelados como predicados binarios, que representan relaciones entre pares de nodos, mientras que las propiedades de los nodos se definen como predicados unarios, que describen características individuales de cada nodo.
- Declaración de Hechos: Los hechos en PyReason pueden ser declarados explícitamente o pueden ser obtenidos a partir de la estructura del grafo. Esto permite una flexibilidad en la forma en que se introducen y manejan los datos dentro del sistema.
- Reglas de Inferencia: Las reglas en PyReason tienen la forma **head** \leftarrow **body**, donde 'head' representa un predicado que se desea establecer como verdadero, y 'body' consiste en un conjunto de cláusulas que deben cumplirse para que el predicado 'head' sea considerado verdadero. Esta estructura permite definir condiciones bajo las cuales se puede inferir nueva información a partir de los hechos existentes.

Dado un conjunto de hechos y reglas, PyReason expande el conjunto de hechos reales en el tiempo o momento actual. Esto significa que puede deducir nueva información y actualizar su conocimiento basado en las reglas definidas y los hechos disponibles, facilitando así un razonamiento dinámico y adaptativo dentro del sistema.

4.1.2. Grafo de Conocimiento

Al inicio del combate, se construye un grafo que contiene todas las posibles combinaciones de efectividad entre los Pokémon. Sin embargo, la creación de un grafo completo puede ser costosa: para optimizar este proceso, solo se pasan al grafo aquellas combinaciones necesarias para el combate específico que se

está llevando a cabo. Esto significa que, en lugar de construir un grafo exhaustivo que incluya todas las posibles interacciones, se limita a las interacciones relevantes para los Pokémon que están actualmente en combate.

Durante cada turno del combate, se pasa el estado actual de ambos Pokémon. Este estado incluye información crítica como la vida y otros atributos relevantes. Para representar estos valores numéricos, se utilizan predicados categóricos. Estos predicados permiten clasificar y evaluar el estado de los Pokémon de manera clara y estructurada.

4.1.3. Modelado de Creencias

Las creencias del agente se modelan como reglas dentro de PyReason. Estas reglas permiten al agente evaluar el estado actual del combate y tomar decisiones basadas en su conocimiento. Por ejemplo:

$$des(M, T) \leftarrow can(M, T) \ \& \ typ_m(M, t_1) \ \& \ typ_p(T, t_2) \ \& \ eff(t_1, t_2) \quad (2)$$

Esto indica que si el agente puede realizar un movimiento sobre un objetivo y el tipo del movimiento es efectivo contra el tipo del objetivo, entonces el agente tiene el deseo de llevar a cabo esa acción.

4.1.4. Modelado de Deseos

El proceso de pensamiento de un agente implica inferir todo aquello que puede a partir de los hechos y creencias (reglas) actuales. Dentro del conjunto de elementos inferidos, un subconjunto es mapeado directamente al espacio de los deseos del agente. Este mapeo asegura que las inferencias realizadas se traduzcan en deseos concretos que el agente puede perseguir.

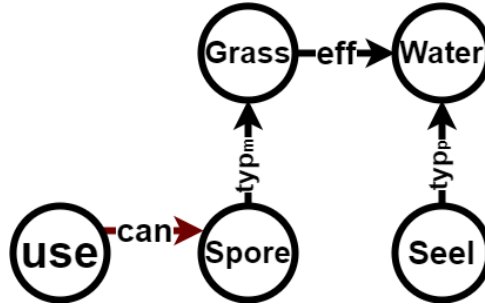


Figura 5: Conocimiento inicial.

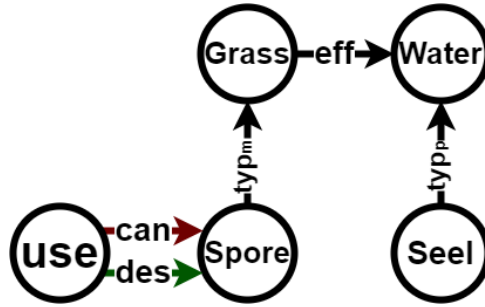


Figura 6: En verde un deseo inferido usando la regla (2). [se definen que predicados constituyen deseos, **des** es solo un predicado mas]

4.1.5. Intenciones

Las intenciones no están expresadas explícitamente; en su lugar, se utiliza un índice obtenido a partir de las reglas para determinar cuánto se desea ejecutar una acción. Este índice se calcula utilizando la función softmax, que permite seleccionar acciones en función de la intensidad del deseo. El índice se obtiene

declarando diversos predicados para diferentes intensidades de deseo y aprovechando las anotaciones de PyReason.

Se utiliza una constante llamada REGRET que verifica si algún plan anterior persiste en los deseos pasados del agente. Si un plan sigue siendo relevante, se mantiene, pero su valor se ajusta multiplicándolo por REGRET, que está entre 0 y 1. Esto permite al agente adaptar sus decisiones basándose en experiencias previas, optimizando así su comportamiento en función de la relevancia continua de sus deseos.

4.1.6. Acciones y Restricciones

Las acciones se toman a partir de un plan predefinido, y se imponen restricciones en el grafo de conocimiento para evitar acciones inválidas. Esto asegura que el agente tome decisiones coherentes y efectivas durante el combate.

4.2. Implementación del Agente Basado en LLM

Los agentes basados en LLM se desarrollaron utilizando de la API de Google [5], el modelo Gemini Flash 1.5. Este enfoque permite a los agentes tomar decisiones informadas durante los combates Pokémon, aprovechando el procesamiento del lenguaje natural para interpretar el estado del juego y generar respuestas adecuadas.

4.2.1. Instrucción de Sistema

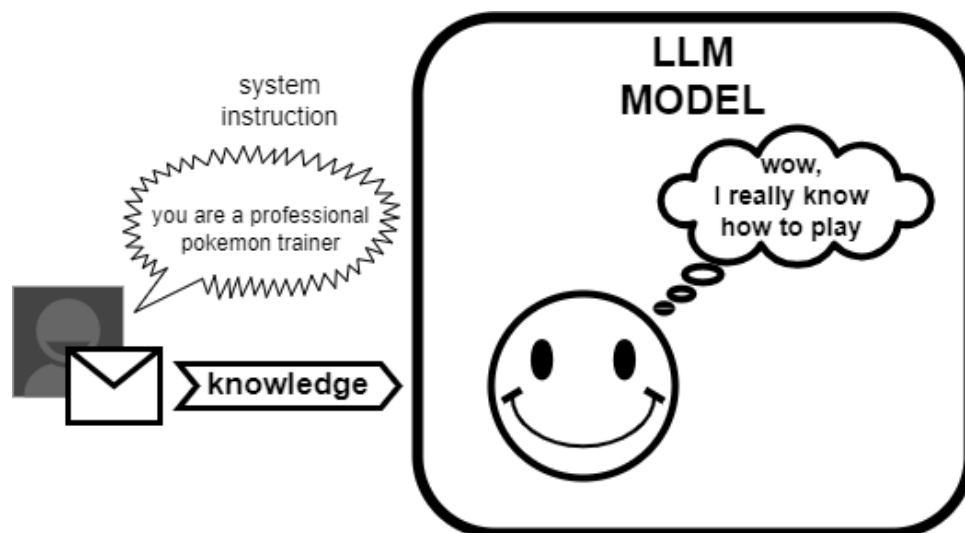


Figura 7: Dando instrucciones al modelo.

Se crea una instrucción de sistema a través de la cual se le pasa al modelo tanto el conocimiento relevante como las instrucciones de formato para las órdenes. Esta instrucción establece el contexto en el que el agente debe operar y define cómo debe estructurar sus respuestas. Entre las instrucciones mas relevantes estan:

- Razonamiento Paso a Paso: Se le indicó al agente que razonara paso a paso sobre su posible decisión.
- Categorización de Jugadas: Cada posible jugada se categoriza desde la A hasta la F. Este enfoque estructurado permite al agente evaluar diferentes estrategias.
- Especificacion de instruccion: Prefija el texto **INSTRUCTION:** a la accion seleccionada para el turno.

4.2.2. Información del Estado del Campo

Cada turno, se proporciona al agente información sobre el estado del campo y de su equipo. Algunos valores numéricos, como la vida de los Pokémon, se convierten a texto categórico para facilitar la

comprensión del estado actual. Por ejemplo, la vida puede ser descrita como 'muy baja', 'baja', 'relativamente alta' o 'perfecta' (Similar que con los agentes BDI). Se le pasa además el historial de turnos y sus consecuencias para evitar que cambie drásticamente su estrategia en situaciones de desventaja total. Esto ayuda a mantener una coherencia en la toma de decisiones y evita reacciones impulsivas que podrían resultar en una pérdida inmediata.

4.2.3. Extracción de Respuestas mediante Expresiones Regulares

Para extraer la respuesta del modelo, se utilizan expresiones regulares, que son herramientas poderosas para buscar y manipular patrones en cadenas de texto. En este caso, las expresiones regulares suelen tener la forma:

INSTRUCTION:(.|\s){action}(.|\s){target}

Desglose de la Expresión Regular

- INSTRUCTION: Esta parte de la expresión indica que se está buscando una instrucción específica que comienza con la palabra 'INSTRUCTION:', lo que proporciona un contexto claro para la acción que se desea realizar.
- (.|\s): Este fragmento permite capturar cualquier carácter (.) o un espacio en blanco (\s), lo que significa que puede haber cualquier tipo de texto entre 'INSTRUCTION:' y las siguientes partes de la expresión.
- {action}: Aquí se especifica el lugar donde debe ir la acción deseada. Este es un marcador que será reemplazado por la acción real que el agente puede ejecutar.
- (.|\s): Similar al primer uso, este fragmento permite capturar cualquier carácter o espacio en blanco, facilitando así la flexibilidad en el formato del texto.
- {target}: Al igual que {action}, este marcador representa el objetivo de la acción, que también será reemplazado por el valor específico correspondiente.

4.3. Sobre la flexibilidad de los agentes racionales

El agente no se limita a ser una simple entidad con una capacidad fija para la tarea para la que ha sido diseñado; en realidad, representa un framework completo que permite la construcción de diversos modelos de combate. Esta flexibilidad es fundamental, ya que los modelos pueden desempeñarse de manera variable, adaptándose a diferentes situaciones y contextos en el juego.

4.4. Análisis de Agentes con Diferentes Creencias

Se analizan tres agentes, cada uno con un conjunto distinto de creencias que influyen en su comportamiento y decisiones. A continuación, se presentan las reglas de inferencia de cada agente:

4.4.1. BASELINE

```
# reglas para la relevancia de deseos
desire_h(A,B) <-0 desire_hh(A,B)
desire_m(A,B) <-0 desire_h(A,B)
desire_l(A,B) <-0 desire_m(A,B)
desire_ll(A,B) <-0 desire_l(A,B)
desire_m(switch,P) <-0 is_switch(switch), is_foe(P, F), on_field_against(C, F), fainted(C)

# reglas para la seleccion de atacar segun efectividad
desire_l(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move.type(M, MT), pokemon.type(F, FT) , effectiveness(MT,FT):[0.2,1.0]

desire_m(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move.type(M, MT), pokemon.type(F, FT) , effectiveness(MT,FT):[0.4,1.0]

desire_hh(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move.type(M, MT), pokemon.type(F, FT) , effectiveness(MT,FT):[0.8,1.0]
# reglas para el switcheo a un pokemon con mayor ventaja
```

```

desire_ll(switch,P) <-0 is_foe(P, F), is_switch(switch), has_move(P, M),
move_type(M, MT), pokemon_type(F, FT), effectiveness(MT,FT):[0.2,1.0]

```

```

desire_l(switch,P) <-0 is_foe(P, F), is_switch(switch), has_move(P, M),
move_type(M, MT), pokemon_type(F, FT), effectiveness(MT,FT):[0.4,1.0]

```

```

desire_m(switch,P) <-0 is_foe(P, F), is_switch(switch), has_move(P, M),
move_type(M, MT), pokemon_type(F, FT), effectiveness(MT,FT):[0.8,1.0]

```

4.4.2. LESS_SWITCH

```

# reglas para la relevancia de deseos
desire_h(A,B) <-0 desire_hh(A,B)
desire_m(A,B) <-0 desire_h(A,B)
desire_l(A,B) <-0 desire_m(A,B)
desire_ll(A,B) <-0 desire_l(A,B)
desire_m(switch,P) <-0 is_switch(switch), is_foe(P, F), on_field_against(C, F), fainted(
    C)

```

```

# reglas para la seleccion de atacar segun efectividad
desire_ll(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move_type(M, MT), pokemon_type(F, FT) , effectiveness(MT,FT):[0.0,1.0]

```

```

desire_l(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move_type(M, MT), pokemon_type(F, FT) , effectiveness(MT,FT):[0.2,1.0]

```

```

desire_m(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move_type(M, MT), pokemon_type(F, FT) , effectiveness(MT,FT):[0.4,1.0]

```

```

desire_hh(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move_type(M, MT), pokemon_type(F, FT) , effectiveness(MT,FT):[0.8,1.0]

```

```

# reglas para el switcheo a un pokemon con mayor ventaja
desire_ll(switch,P) <-0 is_foe(P, F), is_switch(switch), has_move(P, M),
move_type(M, MT), pokemon_type(F, FT), effectiveness(MT,FT):[0.4,1.0]

```

```

desire_l(switch,P) <-0 is_foe(P, F), is_switch(switch), has_move(P, M),
move_type(M, MT), pokemon_type(F, FT), effectiveness(MT,FT):[0.8,1.0]

```

4.4.3. NO_SWITCH

```

# reglas para la relevancia de deseos
desire_h(A,B) <-0 desire_hh(A,B)
desire_m(A,B) <-0 desire_h(A,B)
desire_l(A,B) <-0 desire_m(A,B)
desire_ll(A,B) <-0 desire_l(A,B)
desire_m(switch,P) <-0 is_switch(switch), is_foe(P, F), on_field_against(C, F), fainted(
    C)

```

```

# reglas para la seleccion de atacar segun efectividad
desire_ll(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move_type(M, MT), pokemon_type(F, FT) , effectiveness(MT,FT):[0.0,1.0]

```

```

desire_l(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move_type(M, MT), pokemon_type(F, FT) , effectiveness(MT,FT):[0.2,1.0]

```

```

desire_m(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move_type(M, MT), pokemon_type(F, FT) , effectiveness(MT,FT):[0.4,1.0]

```

```

desire_hh(move,M) <-0 has_move(P,M), on_field_against(P, F), is_use(move),
move_type(M, MT), pokemon_type(F, FT) , effectiveness(MT,FT):[0.8,1.0]

```

4.4.4. Resultados de los Torneos

Se realizaron torneos en formato 1 vs 1 entre los agentes, y se obtuvieron los siguientes resultados:

Name	Won	Percentage (%)
BASELINE	57	51.82
LESS_SWITCH	53	48.18

Cuadro 3: Resultados del Torneo 1

Name	Won	Percentage (%)
BASELINE	59	53.64
NO_SWITCH	51	46.36

Cuadro 4: Resultados del Torneo 2

Name	Won	Percentage (%)
LESS_SWITCH	58	52.73
NO_SWITCH	52	47.27

Cuadro 5: Resultados del Torneo 3

4.5. Probando el Agente LLM

Para evaluar el rendimiento del agente de LLM, se enfrentó al agente BDI BASELINE en un total de 30 combates, debido a las restricciones del uso gratuito de la API.

4.5.1. Instrucción de sistema

- For each available action, you will make a step by step reasoning, assigning at the end a qualification from 'A'(best possible assignation) to 'F'(worst possible assignation).
- From the previous step set of reasonings, you will make a step by step reasoning to choose an option, based in reasonings and categories assigned.
- From the turn historial, you will try to retrieve information about what pokemons has the rival that are not in the field, so when a pokemon faints, you can choose to switch or prepare an attack using that information.
- Avoid switching panic (switching between pokemons again and again, use historial to check wether to that current adversary you switched to a specific pokemon, your reasoning took you to select it).
- You will reassignate categories based on past turn historial (a switch that ended in a bad outcome would be penalized) [examples below]

examples:

use thunderbolt (A) thunderbolt is very effective against POKEMON, so it would be a good option
switch to pikachu (D) pikachu is effective against rival's pokemon, but current pokemon is equally effective, change doesn't make a difference
switch to riolu (B) riolu is equally effective against rival's pokemon, but current pokemon hp is very low, and riolu is on a good state

4.5.2. Ejemplo de dialogo interno

LLM 's INNER DIALOGUE:

1. Use knockoff (A): Knockoff is super effective against Rock types, and it has a chance to remove the opponent's item. This would be a good option to weaken the opponent and potentially gain an advantage.
2. Use hyperbeam (C): Hyperbeam is super effective against Rock types, but it leaves the user vulnerable for a turn. This is a risky move, but it could potentially take out the opponent in one hit.
3. Use dive (B): Dive is super effective against Rock types, and it avoids all attacks except Surf and Whirlpool. This is a good option if the opponent is likely to use a move that Tentacruel is weak to.
4. Use reflecttype (F): Reflecttype is not effective against Rock types, and it could potentially make Tentacruel vulnerable to attacks. This is not a good option.
5. Switch to anorith (D): Anorith is super effective against Rock types, but Tentacruel is already super effective. Switching would be unnecessary.
6. Switch to baltoy (E): Baltoy is not effective against Rock types, and it would be a disadvantage to switch.
7. Switch to ivysaur (E): Ivysaur is not effective against Rock types, and it would be a disadvantage to switch.

8. Switch to genesect (D): Genesect is super effective against Rock types, but Tentacruel is already super effective. Switching would be unnecessary.
9. Switch to raticate (F): Raticate is not effective against Rock types, and it would be a disadvantage to switch.

Reasoning:

- Knockoff is the best option because it is super effective against Rock types and has a chance to remove the opponent’s item.
- Dive is a good option if the opponent is likely to use a move that Tentacruel is weak to.
- Hyperbeam is a risky option, but it could potentially take out the opponent in one hit.
- Switching is not necessary because Tentacruel is already super effective against Rock types.

INSTRUCTION: use knockoff

4.5.3. Resultados

Name	Won	Percentage (%)
BDI	17	56.67
LLM	13	43.33

Cuadro 6: Resultados del torneo

Referencias

- [1] **pokemon-python**: Simulador de batallas que proporcionó una base sólida para las mecánicas de combate y las interacciones entre Pokémon (Repositorio de GitHub).
- [2] **PUREPLES**: Proporcionó la implementación de ES-HyperNEAT (Evolvable-Substrate HyperNEAT) utilizada para evolucionar las topologías de redes neuronales (Repositorio de GitHub).
- [3] **Bulbapedia**, the community-driven Pokémon encyclopedia (ver sitio web).
- [4] **Pokémon Database** (ver sitio web).
- [5] **Google API for devs** (ver sitio web).
- [6] **PyReason** (Repositorio de GitHub).