Redes Convolucionales y Algoritmo Alfa-Beta Aplicado al Pacman

Universidad de Alicante

Técnicas de Algoritmia y Búsqueda

Desarrollado por:
Santiago Álvarez
Alejandro García
Pau Mateo

Índice

1. Introducción

Nuestro trabajo presenta el desarrollo de un agente inteligente para el juego Pacman basado en una **red neuronal convolucional (CNN)**, entrenada por nosotros, para predecir las acciones del agente en función del estado del juego. A diferencia de la implementación original con una **red fully-connected**, en nuestra memoria comentamos la integración de una arquitectura CNN que aprovecha las **características espaciales** del mapa (array del juego).

Además, hemos añadido un componente fundamental para mejorar la toma de decisiones. A simples rasgos hemos integrado el algoritmo de poda alpha-beta junto a un método de evaluación proporcionado por la red neuronal para optimizar la búsqueda del mejor movimiento posible, combinando aprendizaje automático y búsqueda heurística.

2. Descripción general del sistema

2.1 Arquitectura base

El sistema se basa en el repositorio oficial de *UC Berkeley para Pacman con IA*, que proporciona toda la infraestructura para la simulación del juego, los agentes, los fantasmas y la gestión del entorno.

Nuestra contribución principal radica en la integración del algoritmo de poda alpha-beta, además de su adaptación con la nueva y mejorada red neuronal. Junto a esta adaptación viene el diseño de un nuevo agente híbrido llamado ***NeuralAlphaBetaAgent*** que hereda el comportamiento de alpha-beta pero que utiliza la función de evaluación desarrollada en nuestro ***NeuralAgent.***

De forma secundaria hemos modificado y ampliado las siguientes funcionalidades clave. Hemos modificado el mecanismo de evaluación, que combina las **predicciones del modelo** con **heurísticas tradicionales** del entorno del juego (distancia a comida, posición de fantasmas, etc.). Principalmente hemos considerado darle más prioridad a la distancia respecto a la comida, aumentando el score en esta heurística, evitando así paradas innecesarias. También hemos incorporado un sistema de **ponderación de prioridades**, que permite ajustar el peso relativo entre las predicciones del modelo y las heurísticas. Esto nos da un mayor control sobre el comportamiento del agente, permitiendo decidir qué proporción de la decisión final se basa en cada componente.

2.2 Flujo de trabajo

- **Recolección de datos:** Mediante la ejecución del juego con agentes reflexivos y control manual, hemos registrado los estados del juego, acciones tomadas y mapas en formato matricial.

- **Preprocesamiento:** Los mapas se normalizan y convierten en tensores adecuados para el entrenamiento de la red.

- **Entrenamiento de la CNN:** Entrenamos una red neuronal convolucional que, dada la representación del mapa, predice la acción más probable para Pacman.

- **Integración *NeuralAlphaBeta*:** En tiempo de juego, nuestro agente utiliza un algoritmo de poda alpha-beta para explorar posibles futuros estados, usando la CNN como función de evaluación y heurísticas para estimar la calidad de los estados terminales o intermedios.

- **Toma de decisiones:** Combinando la evaluación de la CNN y la poda alpha-beta, el agente selecciona la acción óptima que maximiza la probabilidad de ganar y evita estados desfavorables.

3. Implementación técnica

3.1 Red neuronal convolucional (CNN) (*net.py*)

La decisión de sustituir una arquitectura de PacmanNet basada únicamente en una **fully-connected** por una **CNN**, responde a la necesidad de capturar de manera más efectiva las **relaciones espaciales** presentes en el entorno del juego Pacman. Anteriormente la representación del estado del entorno se transformaba en un vector unidimensional, lo que para nosotros implicaba una pérdida significativa de la **estructura bidimensional** y, con ello, del **contexto espacial** entre los elementos del entorno (paredes, comida, fantasmas, etc.).

Motivación del cambio

- **Preservación del contexto espacial**: Como sabemos las **CNN** están diseñadas para procesar datos con estructura espacial (como **imágenes** o mapas), y por tanto son capaces de mantener las relaciones entre píxeles o, en nuestro caso, celdas del mapa del juego. En el caso de Pacman, la proximidad entre el agente, los fantasmas y los alimentos es crucial para tomar decisiones óptimas.

- **Extracción jerárquica de características**: Las primeras capas de nuestra CNN detectan patrones locales simples (bordes, **esquinas**, bloques), y las capas más profundas combinan estas características para identificar patrones más complejos y abstractos (por ejemplo, **esquinas seguras**, pasillos bloqueados, zonas con alta densidad de comida, etc).

Por tanto, hemos considerado que nuestra arquitectura **PacmanNet** utilice dos bloques convolucionales con activaciones ReLU y operaciones de pooling para reducir progresivamente la resolución espacial del mapa y concentrar la información relevante. Posteriormente, utilizar una capa fully-connected intermedia para transformar esta representación en una forma útil para la capa de salida, que estima las probabilidades de las **5 acciones posibles** en el entorno del juego (arriba, abajo, izquierda, derecha, stop).

Además, hemos incorporado **dropout** como técnica de regularización para mitigar el sobreajuste, favoreciendo una red más robusta ante pequeñas variaciones del entorno.

```python
import torch

class PacmanNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(PacmanNet, self).__init__()

        # ...

        # Capas convolucionales
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)  # Entrada: 1 canal (mapa), Salida: 16
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)  # Salida: 32 canales

        self.pool = nn.MaxPool2d(2, 2)  # Reducimos el tamaño espacial a la mitad
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.3)

       # ...

        # Capas fully connected
        self.fc1 = nn.Linear(conv_output_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):

        # ...
        x = self.relu(self.conv1(x))  # Conv1
        x = self.pool(x)
        x = self.relu(self.conv2(x))  # Conv2
        x = self.pool(x)
        # ...

        return x
```

3.2 Algoritmo alpha-beta

El algoritmo **Alpha-Beta** es una mejora del algoritmo minimax. La función principal es **reducir el número de estados que se evalúan** en el árbol de decisiones, descartando ramas que no afectan al resultado final.

Aunque no hemos modificado este algoritmo, lo hemos utilizado como parte de nuestra solución porque ofrece una forma **eficiente de tomar decisiones** en escenarios donde nuestro agente debe actuar frente a otros (como los fantasmas).

4. Implementación de Agentes

4.1 NeuralAgent (*evaluationFunction*)

```python
def evaluationFunction(self, state):
        """
        Una función de evaluación basada en la red neuronal y en heurísticas adicionales.
        """
        # Heurística  1: Distancia a la comida más cercana
        if food:
            min_food_distance = min(manhattanDistance(pacman_pos, food_pos) for food_pos in food)
            score += 0.0 / ((min_food_distance + 1)

        # Heurística  2: Proximidad a fantasmas
        for ghost_state in ghost_states:
            ghost_pos = ghost_state.getPosition()
            ghost_distance = manhattanDistance(pacman_pos, ghost_pos)

            if ghost_state.scaredTimer > 0:
                score += 50 / (ghost_distance + 1)
            else:
                if ghost_distance <= 2:
                    score -= 200


        # Añadido → Heurística 3: Penalización por cantidad de comida restante
        food_count = len(food)
        score -= 2 * food_count

        # Añadido → Heurística 4: Penalización por turnos sin comer
        if not hasattr(self, 'last_food_count'):
            self.last_food_count = len(food)
            self.turns_without_eating = 0

        current_food_count = len(food)

        if current_food_count < self.last_food_count:
            self.turns_without_eating = 0
            score += 50  # Bonificación por comer
        else:
            self.turns_without_eating += 1

        self.last_food_count = current_food_count

        score -= 5 * self.turns_without_eating  # Castigo por no comer

        # Puntuación de la red neuronal
        neural_score = 0
        for i, action in enumerate(self.idx_to_action.values()):
            if action in legal_actions:
                neural_score += probabilities[i] * 100

        # Añadido → Combinación de ambas puntuaciones
        alpha = 0.7  # confianza en la red
        beta = 0.3   # confianza en heurística
        return alpha * neural_score + beta * score
```

A la hora de elegir el mecanismo de evaluación, hemos probado **diferentes configuraciones** de las diferentes heurísticas. Además, hemos implementado nuevas heurísticas que se añaden a los “factores” ya existentes en el código base. Estas nuevas implementaciones tratan de minimizar el tiempo que queda parado o los rodeos innecesarios sin obtener puntuación.

Para cumplir este objetivo, hemos cambiado la evaluación sobre “distancia a la comida más cercana” cambiando:

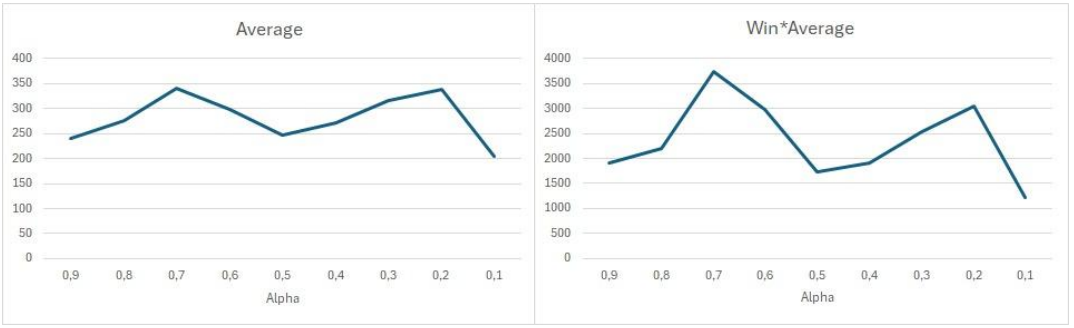
$$\frac{1}{(min\_food\_dist + 1)} \text{ por } \frac{10}{(min\_food\_dist + 1) * 2}$$

Con esta variación hemos obtenido una mejora de hasta un 14% en la puntuación media de las partidas jugadas. Sumado a el cambio en este factor. Añadimos las siguientes heurísticas:

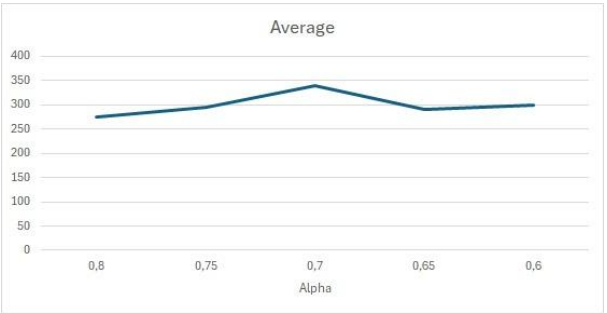
- **Penalización por comida restante**: Se le restará al score el número de comida restante multiplicado por 2.
- **Penalización por turnos sin comer**: Se llevará la cuenta de la comida restante, cuando este número baje, se pondrá la cuenta en 0 y se otorgará una bonificación. En caso de que no se coma en un turno, se sumará 1 a la cuenta y la penalización será de 5 * la cantidad de turnos sin comer.

Con estos cambios, tenemos ya la base de nuestra configuración de la función de evaluación. Aún así, experimentamos con diferentes parámetros para obtener la mejor configuración posible:

- En primer lugar, elegimos **0.7 de peso para la red y un 0.3 de peso para la heurística** pero queríamos confirmar que era la mejor distribución posible. Por ello, realizamos 100 ejecuciones con **semillas aleatorias** desde 0.9-0.1 hasta 0.1-0.9.



En un primer momento encontramos que la media de puntos es prácticamente idéntica en alpha=0.7 y alpha=0.2. Por ello empleamos las partidas ganadas en cada configuración para ver cual se desempeña mejor, alpha=0.7 es la mejor que encontramos. Aún así, decidimos focalizar en 0.7 para ver si algún valor adyacente resultaba mejor. Pero llegamos a la misma conclusión.



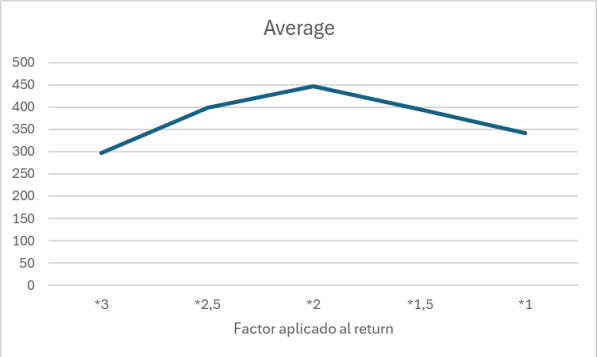
Los cambios sufridos por el código en este punto son las adaptaciones de las heurísticas ya establecidas.

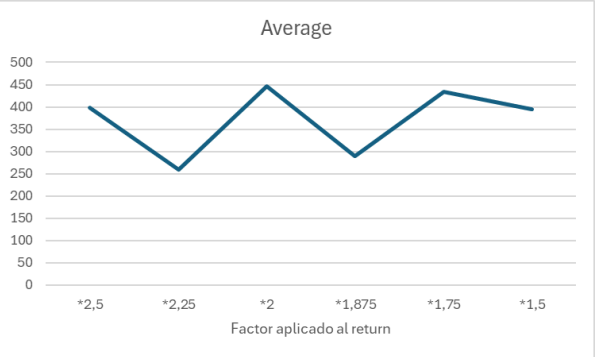
```
# Heurística  1: Distancia a la comida más cercana
if food:
    min_food_distance = min(manhattanDistance(pacman_pos, food_pos) for food_pos in food)
    score += 10.0 / ((min_food_distance + 1)*2)
```

- Tras establecer la relación entre heurísticas y red. Empezamos a probar diferentes parámetros para el return de la función. Partiendo de un:

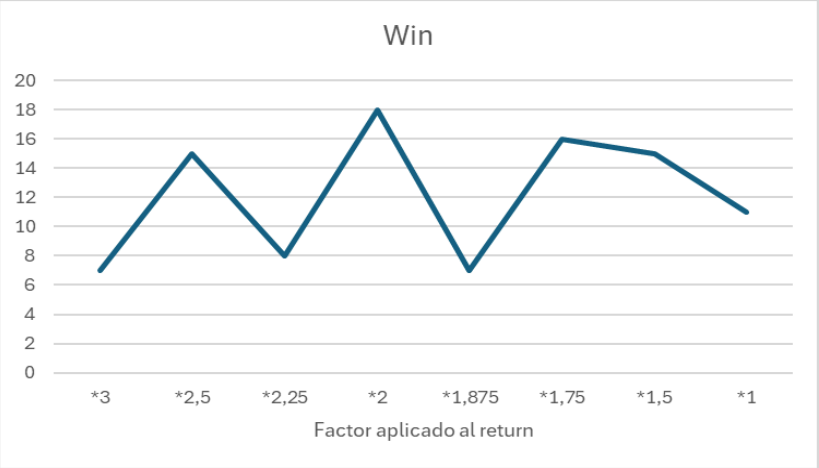
return *alpha * neural_score + beta * score*

Para este parámetro, empleamos la misma técnica que en el anterior, ejecutar 100 partidas con 100 semillas aleatorias y observar los resultados.





Observamos que al multiplicar el return por 2 o por 1.75 obtenemos una mayor media de puntos, si miramos también las partidas ganadas encontraremos una gran mejora.



De esta forma, elegimos 2 como el factor que multiplicará al return de la función de evaluación. Con esto, obtenemos la que es nuestra mejor configuración para una función de evaluación.

4.2 NeuralAlphaBetaAgent

La clase ***NeuralAlphaBetaAgent*** representa una integración híbrida entre el algoritmo **Alpha-Beta** y una **red neuronal** entrenada previamente, utilizada como función de evaluación junto a **heurísticas.**

1. Limitaciones de las funciones heurísticas tradicionales

Hasta ahora, nuestro algoritmo Alpha-Beta usa ***scoreEvaluationFunction***, una función que utiliza la puntuación estándar de PacMan. Aunque este estándar captura ciertas heurísticas simples, estas son estáticas y no aprenden de la experiencia. Lo que nos lleva a evaluaciones inexactas en situaciones complejas o **no anticipadas**.

2. Potencial de la red neuronal como función evaluadora

Por el contrario, nuestra red CNN, entrenada con ejemplos reales del entorno de juego, puede inferir valoraciones más precisas de los estados, incluso en configuraciones no vistas durante el entrenamiento. Esto permite **capturar de forma implícita relaciones espaciales, dinámicas y estrategias de juego** que una heurística manual difícilmente modelaría.

3. Complementariedad entre búsqueda y aprendizaje

El algoritmo Alpha-Beta garantiza una exploración sistemática del espacio de acciones, considerando múltiples futuros posibles. Sin embargo, su rendimiento **depende** en gran medida **de la función que use para evaluar** los estados intermedios y terminales. Al reemplazar esta función por la evaluación de la red neuronal conseguimos una **búsqueda guiada por conocimiento aprendido**, lo que mejora la calidad de las decisiones sin aumentar la complejidad computacional del algoritmo.

```
class NeuralAlphaBetaAgent(AlphaBetaAgent):
    def __init__(self, depth='3', model_path='models/pacman_model.pth'):
        super().__init__(depth= depth)

        self.neural = NeuralAgent(model_path=model_path)
        self.evaluationFunction = self.neural.evaluationFunction
```

De este modo, obtenemos un agente capaz de combinar el conocimiento aprendido a través de múltiples episodios de entrenamiento con la eficacia de un algoritmo de búsqueda como Alpha-Beta. Permitiendo así tomar decisiones adaptativas en tiempo real. Además, al utilizar una red neuronal como función de evaluación, el sistema puede seguir mejorando mediante reentrenamiento con nuevos datos, superando así las limitaciones de las heurísticas estáticas tradicionales.

5. Entrenamiento

Para el entrenamiento, hemos contemplado diferentes aspectos:

- Partidas ganadas.
- Puntos por partida.
- Fantasmas comidos.

Por ello, a la hora de guardar partidas, cambiamos el nombre con el que se guardaban para facilitar el filtrado de las partidas. Pasamos de:

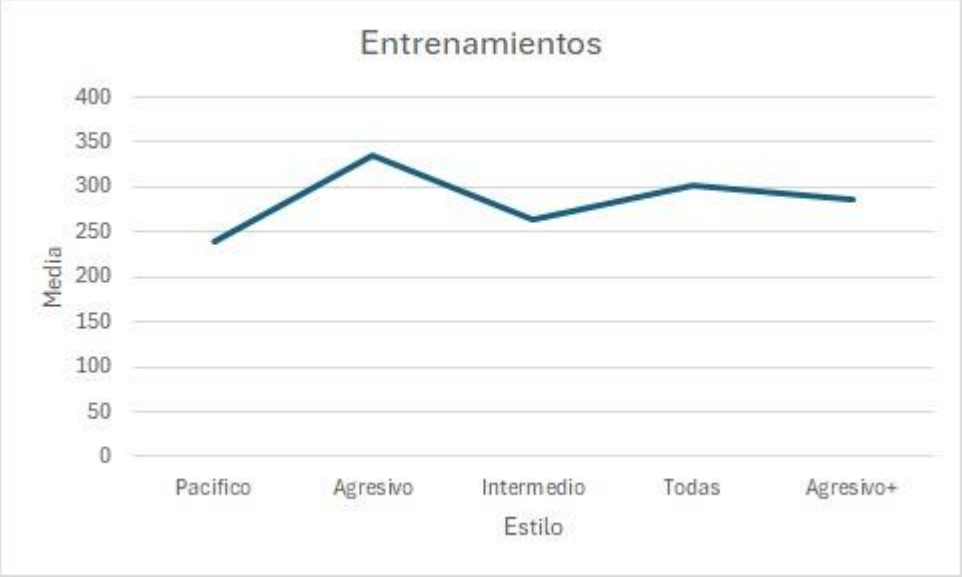
game_{game__id}.csv -> *{puntos}_{jugador}_{Fantasmas comidos}_{semilla}.csv*

game_32.csv -> 1026_Santi_1F_83.csv

Nuestro *pacman_data* cuenta únicamente con **partidas ganadas**, las cuales hemos usado para **entrenar a nuestro modelo** con el que hemos hecho todas las pruebas anteriores de configuraciones. Pero también tenemos la posibilidad de probar diferentes redes, entrenadas cada una con diferentes tipos de partidas. Partidas ganadas en general, partidas pacíficas (sin comer fantasmas) o partidas agresivas (comiendo fantasmas), de esta manera podremos observar cómo se comporta nuestro agente con un 70% de peso de nuestra red teniendo diferentes entrenamientos.

Hemos realizado 500 partidas con 4 modelos diferentes:

- **Pacífico**: Un modelo que solo se ha entrenado con partidas en las que no se comen fantasmas.
- **Agresivo**: Un modelo que solo se ha entrenado con partidas en las que se haya comido por lo menos a 1 fantasma.
- **Intermedio**: Un modelo que no cuenta con partidas en las que se haya comido a 4 fantasmas pero tampoco en las que no se come a ninguno.
- **Todas las partidas**: Un modelo que cuenta con todas las partidas que hemos almacenado.
- **Agresivo+**: Un modelo que se ha entrenado exclusivamente con partidas en las que se comen entre 3 y 4 fantasmas.



Como podemos observar, el modelo agresivo es el que obtiene mejores resultados.

6. Resultados experimentales

6.1 Metodología

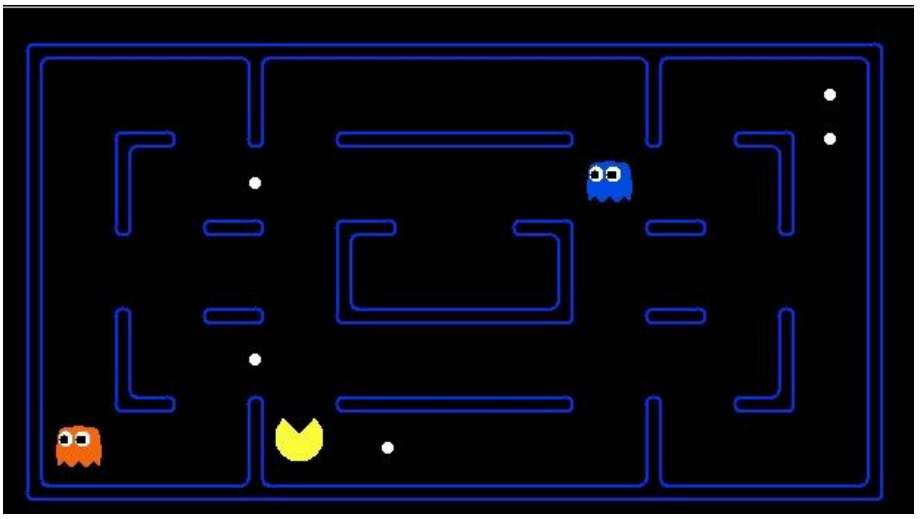
Para la metodología, hemos realizado 10 ejecuciones independientes del agente Pacman con CNN + alpha-beta usando 10 semillas distintas para asegurar la reproducibilidad y evaluar la robustez del sistema.

- **Parámetros usados:**
    - Profundidad de búsqueda alpha-beta: 3 niveles
    - Número de juegos por ejecución: 10
    - Medición de métricas: porcentaje de juegos ganados, puntuación media, mejor juego, peor juego.

6.2 Resultados obtenidos

Semilla	% Juegos Ganados	Puntuación Media	Mejor Juego	Peor juego
903	10%	412.1	1842	-102
21	0%	134.7	735	-163
324	20%	473.8	1619	-321
686	20%	509.2	1865	-357
984	10%	280.6	1118	-387
347	0%	51.4	562	-284
449	30%	753.1	1995	-309
536	0%	146.9	515	-312
607	10%	168.4	1359	-358
693	10%	355.8	1538	-240
Total	**11%**	**328.6**	**1995**	**-387**

- Nuestro *NeuralAlphaBetaAgent* mostró una alta tasa de éxito y estabilidad en la mayoría de las ejecuciones, validando la combinación de CNN y alpha-beta, esto no se ve reflejado dado la tendencia del pacman a **no comerse los últimos elementos** del mapa., tal y como se observa en la siguiente imagen.



Dado que nuestro Agente obtiene puntos por estar **cerca** del alimento, decide no comerlo, simplemente quedarse cerca mientras que evita a los fantasmas. Hemos tratado de solventar este problema con las recompensas que se otorgan en la evaluación, pero el resultado **perjudicaba** en otras partes de la partida.

7. Método de entrenamiento y pruebas realizadas

Durante el desarrollo de esta práctica hemos realizado numerosas pruebas de ejecución con el objetivo de verificar que la lógica del juego, los agentes y el recolector de datos funcionaran correctamente.

En un principio todas estas ejecuciones seguían el método tradicional **secuencial**, es decir, se ejecutaron una a una cada partida. Para pruebas orientadas a conseguir la mejor combinación de parámetros, como alpha y beta, o el “return” de la función de predicción principal, hemos necesitado probar gran cantidad de combinaciones y cada una cantidad de veces suficiente para conseguir resultados consistentes.

Debido a estas necesidades técnicas decidimos implementar métodos de ejecución más eficientes, que nos permitieran obtener los mismos resultados en un tiempo considerablemente menor. De esta manera podríamos probar más cantidad de combinaciones y obtener resultados lo más afinados posibles.

Implementamos un **enfoque paralelo,** en el ejecutaremos partidas de forma simultánea en cada uno de los núcleos de la CPU, conforme cada partida finalice, esta guardará los resultados en el csv correspondiente y el núcleo queda libre para ejecutar la siguiente partida. De esta manera si nuestra CPU tiene 8 núcleos estaríamos realizando pruebas con 8 ejecuciones simultáneamente. Esto nos daría un speedup teórico de hasta 8, es decir

tardaríamos 8 veces menos en obtener los mismos resultados, comparado con una ejecución secuencial.

Para la implementación utilizamos la librería estándar de python **"multiprocessing".** Básicamente, lo que hicimos fue definir una función llamada ***run_single_game***, que se encarga de preparar y ejecutar una única partida del juego. Luego, en la función principal ***runGames***, en lugar de ejecutar las partidas una a una, lanzamos varias de ellas en paralelo usando un pool de procesos. De esta manera, cada núcleo del procesador puede encargarse de una partida diferente.

Esquema de ejecución paralela:

- **run_single_game()**: En cada núcleo se encargará de ejecutar una partida independiente, esta al ser finalizada enviará los datos de su ejecución y los guardará en el csv. Posteriormente este núcleo se libera para dar paso a la siguiente ejecución.

- **runGames()**: Es la función que antes ejecutaba todas las partidas una tras otra. Ahora, crea una lista de argumentos y los lanza en paralelo usando multiprocessing.Pool. Al final, recoge las partidas para mostrar estadísticas como puntuaciones, victorias y derrotas.

Veamos estos cambios reflejados en el código.

```
def runGames(layout, pacman, ghosts, display, numGames, record, numTraining=0, catchExceptions=False,
timeout=30, replay_mode=False):
    from pacman import ClassicGameRules
    import __main__
    __main__.__dict__['_display'] = display

    print("Reply mode:", replay_mode)

    pool = mp.Pool(mp.cpu_count())

    args = [(i, layout, pacman, ghosts, display, record, numTraining, timeout, catchExceptions,
replay_mode) for i in range(numGames)]

    results = pool.map(run_single_game, args)

    games = [g for g in results if g is not None]

    if (numGames - numTraining) > 0 and games:
        scores = [game.state.getScore() for game in games]
        wins = [game.state.isWin() for game in games]
        winRate = wins.count(True) / float(len(wins))
        print('Average Score:', sum(scores) / float(len(scores)))
        print('Scores:       ', ', '.join([str(score) for score in scores]))
        print('Win Rate:      %d/%d (%.2f)' % (wins.count(True), len(wins), winRate))
        print('Record:       ', ', '.join([['Loss', 'Win'][int(w)] for w in wins]))

    return games
```

La función ***runGames*** quedaría reducida únicamente a mandar a cada proceso la tarea y recoger los datos de cada uno.

Veamos ahora la función encargada de realizar la ejecución de cada juego:

```
def run_single_game(args):
    i, layout, pacman, ghosts, display, record, numTraining, timeout, catchExceptions, replay_mode =
args

    from pacman import ClassicGameRules  # importa internamente si hace falta
    import textDisplay

    seed = int(time.time()) % 1000 + i  # para que no sea el mismo
    random.seed(seed)
    rules = ClassicGameRules(timeout)
    beQuiet = i < numTraining
    gameDisplay = textDisplay.NullGraphics() if beQuiet else display
    rules.quiet = beQuiet

    data_collector = gamedata.GameDataCollector(replay_mode=replay_mode)

    game = rules.newGame(layout, pacman, ghosts, gameDisplay, beQuiet, catchExceptions)
    game.data_collector = data_collector
    game.run()
    data_collector.save_game_data(seed)

    if record and not beQuiet:
        fname = ('recorded-game-%d' % (i + 1)) + '-' + '-'.join([str(t) for t in time.localtime()
[1:6]])
        with open(fname, 'wb') as f:
            components = {'layout': layout, 'actions': game.moveHistory}
            pickle.dump(components, f)

    if not beQuiet:
        return game
    return None
```

En esta función recibimos los parámetros de cada partida, seleccionamos la semilla y procedemos de la misma forma que antes de paralelizar. Finalmente devolvemos los datos y guardamos usando el data_collector.

Los cambios al código fueron realizados de manera aislada, sin alterar otras funciones o estructura del proyecto. Es decir, conservamos cada uno de los parámetros y salidas originales, adaptando la lógica interna para realizar ejecuciones paralelas.

8. Discusión y conclusiones

En este proyecto hemos demostrado cómo la combinación de técnicas clásicas de búsqueda como el algoritmo **alpha-beta** con enfoques basados en **redes neuronales convolucionales** puede dar lugar a un agente de juego más inteligente, adaptativo y eficaz. Partiendo de una arquitectura ya establecida, no solo la mejoramos incorporando una **CNN** que aprovecha la estructura espacial del entorno, sino que también la integramos con una estrategia de búsqueda que permite explorar posibles acciones de forma más precisa y eficiente.

Los resultados obtenidos respaldan esta integración: el agente toma mejores decisiones, gana más partidas y muestra un comportamiento más coherente en situaciones complejas. Además, la posibilidad de seguir entrenando la red con nuevos datos lo convierte en un sistema flexible, que puede evolucionar con el tiempo, a diferencia de las heurísticas estáticas tradicionales.

Más allá del caso concreto de Pacman, este enfoque híbrido sienta una base sólida para seguir explorando cómo el aprendizaje automático y la búsqueda clásica pueden complementarse en la creación de agentes inteligentes para todo tipo de entornos.

9. Referencias

- UC Berkeley Pacman AI Projects: https://ai.berkeley.edu/project_overview.html
- Repositorio de Github: https://github.com/santvz6/TAB_pacman_IA