ChatGPT

Plan detallado de desarrollo: Agente de Pac-Man con red neuronal

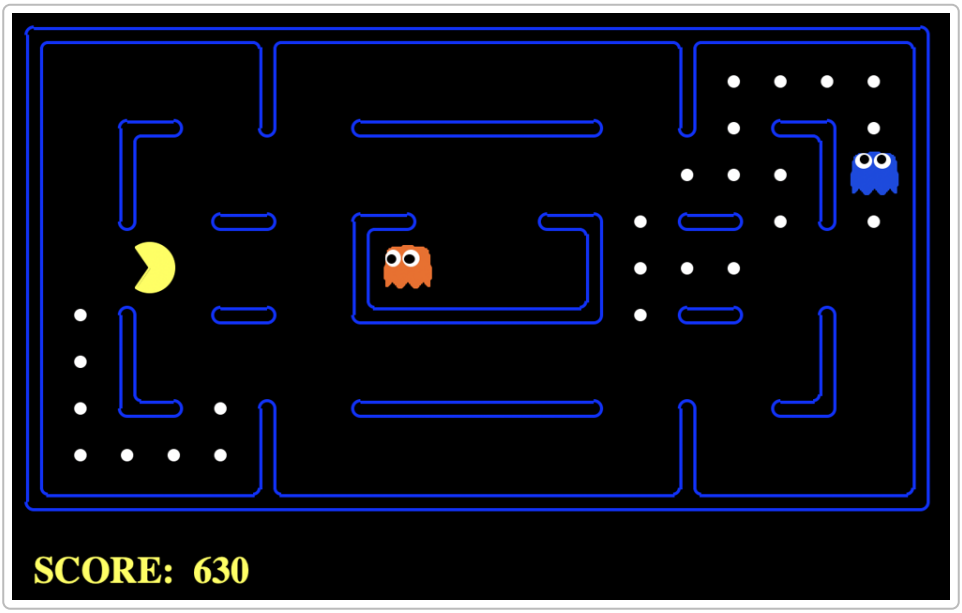

Figura: Escenario clásico de Pac-Man, donde el agente Pac-Man (círculo amarillo) recorre el laberinto comiendo todos los puntos blancos (comida) mientras evita a los fantasmas. En la imagen, un fantasma azul indica que está vulnerable (Pac-Man puede comerlo temporalmente) y uno naranja está en estado normal. El objetivo es maximizar la puntuación comiendo toda la comida sin ser atrapado por un fantasma [1].

Pac-Man es un juego **de tipo adversario** con información totalmente observable (Pac-Man siempre conoce la posición de muros, comida y fantasmas) [1]. Tradicionalmente, problemas como Pac-Man se modelan como juegos de suma cero en los que dos agentes (Pac-Man y los fantasmas) alternan turnos [2]. Se pueden aplicar algoritmos de búsqueda adversaria como *Minimax* y *Alpha-Beta* para calcular movimientos óptimos explorando el árbol de juego [3]. Sin embargo, para un juego complejo como Pac-Man, la búsqueda completa es costosa, incluso con poda Alfa-Beta [4], y requiere definir una heurística para evaluar posiciones intermedias. En este proyecto adoptaremos un enfoque distinto: **entrenar una red neuronal** que aprenda a tomar buenas decisiones imitando partidas de ejemplo, logrando aproximar la estrategia óptima con menor costo computacional. Este tipo de enfoque sigue la idea de los "oráculos profundos", donde una DNN (red neuronal profunda) aproxima una función ideal que no podemos calcular exactamente, logrando soluciones cercanas a las óptimas en tiempo razonable [5].

A continuación se presenta un plan exhaustivo, dividido en fases secuenciales, para implementar y entrenar un agente inteligente de Pac-Man basado en una red neuronal. El plan cubre desde el **análisis del código base** proporcionado, la **recolección y preparación de datos** de partidas, el **diseño e implementación** del modelo de red neuronal, el **entrenamiento** y **validación** del agente, hasta su **integración** en el juego y la **preparación del entregable** final. Cada fase considera los requisitos técnicos (por ejemplo, uso de PyTorch, formato de datos) y respeta la restricción de **no modificar secciones de código marcadas con** `#ahmed`, tal como se indica en las instrucciones del proyecto. Se

emplean encabezados y listas para organizar claramente las tareas, siguiendo un orden cronológico lógico.

1. Análisis del código base y requisitos técnicos

En esta fase inicial se revisará a fondo la estructura del repositorio proporcionado (`PACMAN` en GitHub) y se identificarán los componentes clave del código, así como las partes que se pueden o no modificar. El objetivo es comprender cómo funciona el juego y dónde encajará la solución de la red neuronal. Las tareas a realizar son:

- **Revisar la estructura de archivos del proyecto**: Identificar los módulos principales relacionados con Pac-Man y la IA. Según la documentación, el proyecto incluye:
- `game.py` – Motor del juego: define las mecánicas básicas, la gestión de estados y la lógica de turnos de agentes.
- `pacman.py` – Programa principal para ejecutar el juego (manejo del bucle de juego, interfaz gráfica, parámetros de ejecución, etc.).
- `multiAgents.py` – Implementa los distintos agentes inteligentes para Pac-Man:
    - Agentes reflexivos y de búsqueda (por ejemplo, con Minimax/Alpha-Beta).
    - El `NeuralAgent`, que será el agente controlado por la red neuronal entrenada.
    - Funciones de evaluación de estados de juego.
- `net.py` – Define la **red neuronal (`PacmanNet`) usando PyTorch**, e incluye la funcionalidad de entrenamiento, carga/guardado del modelo y procesamiento de datos [6].
- `gamedata.py` – Módulo de **recolección de datos de juego**: define la clase *GameDataCollector* encargada de capturar el estado, acciones y resultado durante las partidas, formatearlos (mapas del juego en matrices numéricas) y guardarlos en archivos CSV para entrenamiento [7].
- `ghostAgents.py` – Define el comportamiento de los fantasmas (e.g. movimiento aleatorio u orientado hacia Pac-Man).

- **Otros utilitarios**: `layout.py` (diseños de laberintos), `keyboardAgents.py` (control manual de Pac-Man), `util.py` (estructuras de datos auxiliares, cálculos de distancia, etc.), módulos de visualización (`graphicsDisplay.py`, `graphicsUtils.py`, `textDisplay.py`), etc. Estos ayudan al funcionamiento pero no requieren modificación para nuestro objetivo.

- **Identificar secciones de código marcadas con `#ahmed`**: Es crucial localizar en qué archivos y funciones aparece el comentario `#ahmed`, ya que **esas partes no deben modificarse**. Estas suelen corresponder a código base proporcionado por el instructor que no debemos alterar. Acciones a realizar:

- Abrir los archivos relevantes (especialmente `multiAgents.py`, `net.py`, `gamedata.py`, `pacman.py`) y buscar el texto "#ahmed". Tomar nota de qué bloques de código están protegidos por esta marca. Por ejemplo, es posible que ciertas funciones o configuraciones cruciales (como partes del agente o de la arquitectura de la red ya definidas) estén marcadas con `#ahmed` para impedir cambios.
- **Entender el código protegido**: aunque no podamos modificarlo, debemos comprender su función. Por ejemplo, si en `NeuralAgent` algunas partes (como la combinación de la predicción de la red con heurísticas) están marcadas con `#ahmed`, significa que esa lógica está dada y debemos diseñar nuestro modelo para que funcione bien con ella.

- Asegurarse de **no realizar cambios** en estos segmentos durante el desarrollo. Mantener un registro de qué porciones del código están fuera de nuestro alcance ayudará a evitar modificaciones accidentales.

- **Comprender el funcionamiento del agente neuronal existente**: Leer la implementación de la clase `NeuralAgent` en `multiAgents.py` para ver cómo utiliza la red neuronal:

- Verificar **cómo carga el modelo entrenado** (probablemente desde un archivo en el directorio `models/`). Identificar el nombre o ruta de archivo esperado para el modelo (p.ej., podrían usar un nombre fijo como `"pacman_net.pth"` o similar) y el formato (PyTorch `state_dict`). Esto nos dirá cómo debemos guardar nuestro modelo tras el entrenamiento.
- Revisar la función `getAction(gameState)` del `NeuralAgent`:
    - ¿Cómo convierte el estado del juego (`gameState`) a la entrada de la red? Es probable que llame a funciones que construyen matrices representando el laberinto, posiciones de Pac-Man, fantasmas, etc., conforme al formato esperado por `PacmanNet` [8]. Confirmar si delega esto a alguna función en `gamedata.py` o si lo hace internamente.
    - ¿Qué **salida espera de la red neuronal**? Por ejemplo, si la red devuelve un valor de *confianza* o puntaje por cada acción posible, el agente podría seleccionar la acción con mayor valor ajustado. Según la documentación, el `NeuralAgent` **combina la predicción de la red con conocimiento heurístico del juego** para evaluar cada movimiento [9]. Esto implica que para cada movimiento posible Pac-Man:
    - Toma el **"puntaje" o probabilidad predicho por la red** (qué tan buena es esa acción según el modelo entrenado).
    - Considera factores heurísticos adicionales: cercanía a la comida (priorizar comer puntos cercanos), distancia a los fantasmas (evitar movimientos que acerquen a un fantasma peligroso, o perseguir si el fantasma es comestible), la existencia de cápsulas de poder, y el cambio en la puntuación del juego [10].
    - Combina estos factores (por ejemplo, sumándolos con ciertos pesos) para determinar un valor final de evaluación de la acción.
    - Confirmar si el agente utiliza **exploración/explotación**: es decir, puede que a veces elija acciones al azar para explorar. La documentación menciona que equilibra exploración y explotación con una tasa de exploración decreciente [11]. Ver si hay un parámetro epsilon o similar que inicialmente permita movimientos aleatorios. Esto afectará nuestras pruebas (p.ej., sabiendo que al comienzo el agente podría moverse aleatoriamente, pero cada vez menos a medida que juega más partidas).
    - **Restricciones de tiempo**: Verificar si hay alguna restricción de tiempo por turno o limitaciones de rendimiento. Una red muy pesada podría causar demoras en la elección de la acción; hay que asegurarse de que el modelo final sea lo suficientemente eficiente para decidir en tiempo real durante el juego.

- **Entender la recolección de datos de partidas**: Examinar el módulo `gamedata.py` y cómo se integra con el juego:

- Identificar la clase `GameDataCollector` y sus métodos. Seguramente tiene métodos para inicializar la recopilación al comenzar una partida, para registrar el estado y la acción en cada turno, y para guardar al terminar la partida.
- Comprender **qué información del estado se registra** en los CSV. Según la documentación, se guardan matrices numéricas representando el estado del juego (muros, comida, cápsulas, fantasmas, Pac-Man) junto con las acciones realizadas y quizás recompensas o resultados [7]. Verificar:
    - Formato del estado: posiblemente varias columnas para representar la cuadrícula (podría ser una lista de números o varias columnas por celda). Por ejemplo, podrían aplanar la matriz del laberinto en una fila de CSV, o almacenar varios campos por celda.

    - Representación de la acción: probablemente como un número o cadena que identifica la dirección tomada (`North`, `South`, `East`, `West` o similar).
    - Si guardan el resultado o puntaje alcanzado al final de la acción o del juego.
- **Integración con el juego**: Determinar cómo se inicia la colección. Puede ser automático: por ejemplo, el `GameDataCollector` podría instanciarse en `pacman.py` cuando lanzamos una partida, a menos que el código exija una opción explícita. Revisar `pacman.py` para ver si se crea un `GameDataCollector` cuando Pac-Man es controlado por teclado o por un agente. Si no es obvio, planear invocar manualmente la recolección (quizás modificando mínimamente `pacman.py` si no hay restricción, siempre y cuando no esté marcado con `#ahmed`). Lo más probable es que la recolección esté habilitada por defecto para generar datos de entrenamiento sin cambiar código.

- Revisar la carpeta `pacman_data/` en el repositorio. Esta puede ser la ubicación donde se guardan los archivos CSV de datos de partidas. Ver si contiene archivos de ejemplo o si está vacía. Asegurarse de que el programa tenga permisos para escribir en este directorio cuando se ejecuta.

- **Confirmar dependencias y configuración**: Asegurarse de tener instaladas las librerías necesarias:

- El archivo `README.md` del repositorio indica instalar `numpy`, `matplotlib`, `pandas` y `torch` [12]. Estas son necesarias para operaciones numéricas, manejo de datos y la red neuronal (PyTorch). Configurar el entorno Python con estos paquetes (por ejemplo, usando `pip install -r requirements.txt` si existe, o manualmente instalarlos).
- Confirmar la versión de Python y PyTorch recomendada (no se especifica en la documentación, pero usar una versión compatible con el código, típicamente Python 3.x y PyTorch 1.x o 2.x).
- Ejecutar una **prueba rápida del juego** para verificar que todo esté funcionando inicialmente:
    - Por ejemplo, correr `python pacman.py` para jugar manualmente una partida con el teclado y verificar que la ventana del juego se abre y responde.
    - Correr `python pacman.py -p RandomAgent` para ver a Pac-Man controlado por un agente aleatorio y comprobar que los fantasmas y la puntuación funcionan.
    - Esto garantiza que el entorno está correctamente configurado antes de implementar cambios.

Al finalizar esta fase de análisis, debemos tener un **mapa mental claro de la base de código**, saber exactamente **dónde agregaremos código nuevo** (p.ej., en `net.py` para la arquitectura y entrenamiento) y **qué partes no tocar**. También habremos verificado que podemos registrar partidas en CSV y que entendemos el formato de esos datos, lo cual es crucial para las siguientes fases.

2. Recopilación de datos de partidas para entrenamiento

Para entrenar la red neuronal, necesitamos un conjunto de datos de ejemplos de juego de Pac-Man. En este proyecto, se especifica usar **"mis propias partidas guardadas"** como datos de entrenamiento, lo que implica que el estudiante (o desarrollador) generará datos jugando Pac-Man o utilizando algún agente para recopilar experiencias. En esta fase, se llevarán a cabo las siguientes tareas:

- **Configurar la recolección de datos**: Tras entender cómo `GameDataCollector` funciona, habilitar la captura de datos en las partidas:
- Si el sistema de recolección es automático, simplemente iniciar partidas normalmente y confirmar que al terminar se genera un archivo CSV de datos en `pacman_data/`.

- Si se requiere un *trigger* explícito (por ejemplo, un flag en la línea de comandos o modificar un parámetro), establecerlo. Posibles acciones:

    - Ejecutar el juego con Pac-Man controlado manualmente: `python pacman.py` (por defecto, el Pac-Man controlable por teclado se suele llamar `KeyboardAgent`). Este modo permitiría jugar y, con suerte, `GameDataCollector` almacenará cada movimiento.
    - Verificar la consola o logs: puede que el programa indique algo como "Guardando datos de la partida en pacman_data/...csv" al final, o similar.
    - Si no se guarda automáticamente, evaluar la posibilidad de modificar **ligeramente** el código (en una sección no marcada con `#ahmed`) para forzar la recolección. Por ejemplo, en `pacman.py`, después de la partida, llamar a un método de GameDataCollector para guardar. Sin embargo, esto es último recurso si realmente no se guarda de otro modo.

- **Jugar partidas manualmente para generar datos**: El estudiante deberá jugar **varias partidas de Pac-Man** para obtener suficiente información de entrenamiento. Aspectos a considerar:

- **Número de partidas**: Cuantas más, mejor. Un objetivo podría ser al menos 10 a 20 partidas completas, dependiendo del tiempo y la complejidad. Si las partidas son cortas (por morir pronto), conviene jugar más para tener ejemplos variados. Idealmente incluir partidas hasta ganar (Pac-Man come todos los puntos) o al menos lograr puntuaciones altas para proporcionar ejemplos de buen juego.
- **Diferentes escenarios**: Pac-Man puede jugarse en distintos laberintos (layouts). Por defecto suele haber un layout estándar (*mediumClassic*). Para robustez, se podría incluir variedad:
    - Ejecutar partidas en el layout por defecto (ej. `-l mediumClassic` que suele ser el default).
    - Opcionalmente, probar algún layout más pequeño (`-l smallGrid`) para datos adicionales rápidos, o más complejo (`-l originalClassic`) si existe. Esto dará diversidad de estados.
- **Estrategia de juego**: Al ser datos para entrenar la red, conviene jugar *lo mejor posible*, demostrando estrategias deseables:
    - Priorizar comer los puntos de comida sin perder vidas.
    - Usar las cápsulas de poder (si las hay) para comer fantasmas cuando sea seguro, lo que otorga puntos extra.
    - Evitar rincones donde un fantasma pueda acorralar a Pac-Man, etc.
    - En resumen, generar ejemplos de **decisiones correctas** que lleven a maximizar la puntuación. Estos patrones serán aprendidos por la red.
- **Explorar situaciones variadas**: Por ejemplo, incluir en las partidas momentos donde Pac-Man es perseguido de cerca por un fantasma (para que el modelo aprenda a huir), momentos donde un fantasma está vulnerable (para aprender que puede ser perseguido), situaciones de pasillos largos vs. áreas abiertas, etc. Cuanto más diversa sea la experiencia recopilada, más rica será la información para el entrenamiento.

- **Tamaño del conjunto de datos**: Cada paso del juego (cada movimiento de Pac-Man) suele convertirse en una fila de datos. Si una partida tiene, digamos, 200 movimientos, 10 partidas generarían ~2000 ejemplos de entrenamiento. Esto puede ser suficiente para una primera versión, aunque más datos tienden a mejorar el aprendizaje. En la medida de lo posible, obtener unos varios miles de instancias. Si jugar manualmente es muy laborioso, considerar:

    - Utilizar un agente existente para generar datos adicionales: por ejemplo, si hay un agente reflex (o el RandomAgent, aunque este juega mal), sus partidas se pueden grabar

también. Un agente aleatorio no es ideal para buenos ejemplos, pero podría aportar variedad en estados; un agente Minimax (si estuviese implementado) podría aportar movimientos óptimos. Sin embargo, usar datos de un agente inferior podría sesgar el aprendizaje. Es preferible centrarse en datos de buena calidad (jugador humano o agente óptimo si existe).

- **Verificar y almacenar los archivos de datos**: Después de cada partida:

- Comprobar que se haya creado un archivo CSV en la carpeta `pacman_data`. Estos archivos podrían nombrarse secuencialmente (p. ej., `game_1.csv`, `game_2.csv`, etc.) o con timestamps. Abrir uno de ellos para revisar su contenido y asegurarse de que los datos se grabaron correctamente.
- **Examinar el contenido CSV**: Abrir el CSV (con un editor de texto o Excel/Pandas) e identificar las columnas:
    - Esperar columnas que representen las características del estado. Por ejemplo: puede haber una columna por cada celda del grid (si lo aplanaron) o campos como `wall[x,y]`, `food[x,y]`, etc. Es posible que en lugar de columnas separadas, haya una sola columna con una representación codificada (por ejemplo, una cadena). Pero dado que se mencionó *formato numérico*, es probable que sean múltiples columnas numéricas.
    - Encontrar la columna de **acción** tomada. Podría ser un número (0-3) o texto (“North”, “South”, etc.). Esta será nuestra variable objetivo (label) para un modelo supervisado que prediga la acción a partir del estado.
    - Ver si hay una columna de recompensa o puntuación tras la acción. Podría haber un campo indicando la recompensa inmediata (ej. +10 por comer un punto, -500 por morir, etc.). Si existiera y quisiéramos entrenar con métodos de aprendizaje por refuerzo, sería relevante, pero en nuestro caso podríamos enfocarnos en aprendizaje supervisado de la acción óptima, por lo que posiblemente ignoremos recompensas explícitas.
    - Confirmar si se registra el estado de los fantasmas (normales vs vulnerables). Esto puede aparecer en la representación del estado: por ejemplo, fantasmas vulnerables podrían tener un identificador distinto en la matriz (p.ej., un valor diferente). De no ser así, quizás haya una columna separada indicando si un fantasma está en modo comestible y por cuánto tiempo. Esta información es importante para decisiones: Pac-Man juega diferente si el fantasma es comestible.
    - Notar si el CSV incluye múltiples líneas por turno (poco probable). Lo normal es una línea por movimiento de Pac-Man. Cada línea representa el estado actual y la acción tomada en ese estado.

- **Consolidar los datos**: A medida que se juegan partidas y se generan CSVs, mantenerlos organizados:

    - Renombrar o numerar los archivos si no lo están, para saber cuáles corresponden a qué partida.
    - Mover todos los archivos de datos a la misma carpeta (`pacman_data/` ya lo estará) y considerar si hace falta combinarlos más adelante. El proceso de entrenamiento podría leer múltiples archivos o requerir uno solo combinado; lo clarificaremos en la siguiente fase.
    - Hacer un respaldo de estos datos brutos antes de procesarlos, por si se necesita reutilizar o revisar más tarde.

- **Ejemplo ilustrativo de los datos (si es útil)**: Para documentar el plan, podríamos imaginar una pequeña tabla de muestra con algunas columnas:

Estado (parcial)	...	Pacman_x	Pacman_y	Ghost1_x	Ghost1_y	FoodCount	Acción
... 0,1,1,0,... (map)	...	5	3	7	3	12	"North" (0)
... 0,1,2,0,...	...	5	4	7	3	11	"East" (1)

(Nota: Esta tabla es hipotética; en la práctica el estado podría venir representado de otra forma numérica. Aquí se ilustra la idea de tener la posición de Pac-Man, la de un fantasma, el conteo de comida restante, etc., junto a la acción tomada.)

Este ejemplo mostraría dos instancias: en la primera, Pac-Man en (5,3) con 12 comidas restantes decide ir Norte; en la segunda, luego de moverse, decide ir Este, etc. Lo importante es que cada fila tendrá la acción correcta a tomar dada la situación, y nuestro modelo aprenderá a replicar esas decisiones.

- **Asegurar la calidad de los datos**: Revisar rápidamente los datos colectados para detectar posibles problemas:
- Si alguna partida quedó incompleta en el registro o hay filas corruptas (p.ej., si se cierra el juego abruptamente, podría faltar el último movimiento en CSV). En tal caso, descartar o corregir esos casos.
- Uniformidad: confirmar que todas las columnas están presentes en todos los archivos (mismo formato de columnas). Si detectamos que diferentes partidas generaron diferentes estructuras, habría que armonizarlas o extraer sólo las columnas comunes.
- Cantidad de clases de acción: debería haber 4 posibles acciones (Up, Down, Left, Right). Comprobar que todas aparezcan en el dataset. Si alguna dirección aparece muy pocas veces (por ejemplo, raramente se va hacia arriba porque el laberinto lo permitió poco), tomar nota: el modelo podría tener más dificultad en esos casos. Si el desequilibrio es muy grande, se puede intentar jugar más movimientos que incluyan esa acción para balancear.

Tras esta fase, deberíamos contar con un **conjunto de datos de entrenamiento listo**, consistente en varias partidas de Pac-Man almacenadas en CSV, que representan las experiencias (estado->acción) del Pac-Man. Este dataset será la base para entrenar la red neuronal en la siguiente etapa. Antes de proceder, es recomendable asegurar que tenemos suficiente volumen y variedad de datos, ya que de ello depende la calidad del modelo resultante. Si los datos son escasos, se podría en este punto decidir jugar algunas partidas adicionales o complementarlas con datos generados por algún agente heurístico, siempre valorando la calidad de esas decisiones adicionales.

3. Procesamiento y preparación de los datos de entrenamiento

Con los datos brutos de las partidas en mano, la siguiente etapa consiste en **preprocesarlos** para que puedan ser usados eficientemente para entrenar la red neuronal. Esto implica convertir los datos del CSV al formato de entrada del modelo, limpiar lo que sea necesario, y particionar en conjuntos de entrenamiento/validación. Las tareas detalladas son:

- **Unificación de datos (si hay múltiples archivos)**: Si las partidas quedaron registradas en varios archivos CSV separados (por ejemplo, `partida1.csv`, `partida2.csv`, ...), decidir cómo manejarlos:

- La forma más sencilla es **combinar todos los datos en un solo conjunto unificado**. Podemos concatenar los archivos CSV en uno solo grande:
    - Usar una herramienta como `pandas`: por ejemplo, leer cada CSV en un DataFrame y luego hacer `pd.concat` para unirlos.
    - Asegurarse de que la concatenación alinee correctamente las columnas (deben ser idénticas en orden y nombre para todas las partidas).
    - Alternativamente, modificar el código de entrenamiento para que lea múltiples archivos en vez de uno. Esto podría implicar ajustar la función de carga en `net.py` (si no está marcado como #ahmed). Por simplicidad, la combinación previa a la carga es recomendable.
- Evitar duplicados: cada partida debe ser única, pero si accidentalmente se recopilaron dos veces los mismos datos, no repetirlos.

- Guardar el dataset combinado en un nuevo archivo, por ejemplo `pacman_data/all_games.csv` o cargarlo directamente en memoria con pandas para la siguiente etapa, sin necesidad de crear un archivo intermedio.

- **Mapeo de características de entrada**: Entender y aplicar la transformación necesaria desde las columnas del CSV al formato que requiere la red neuronal:

- Si el CSV ya tiene cada celda del laberinto como columna con un valor numérico que indica el contenido (muro, comida, vacío, Pac-Man, fantasma, cápsula, etc.), es posible que tengamos que reconstruir la **matriz 2D del estado** a partir de esa fila de valores. Alternativamente, puede que ya vengan varias columnas por tipo de objeto.
- Revisar la estrategia de representación que se usará como entrada. Según la documentación:
    - El juego convierte los estados en **matrices numéricas** representando distintas capas: paredes, comida, cápsulas, fantasmas, Pac-Man [13]. Lo más probable es que internamente maneje una representación matricial del laberinto, donde cada posición (x,y) puede tener diferentes elementos codificados.
    - Una forma común es utilizar **canales (layers) separados** para cada tipo de entidad:
    - Un canal binario para paredes (1 si hay muro, 0 si no).
    - Un canal para comida (1 si hay punto de comida en esa celda, 0 si no).
    - Un canal para cápsulas de poder (1 si hay cápsula, 0 si no).
    - Un canal para la posición de Pac-Man (por ejemplo, una matriz llena de 0 salvo un 1 donde está Pac-Man).
    - Uno o varios canales para fantasmas. Si son varios fantasmas, podría haber:
        - Un canal combinado que marca con 1 las posiciones de cualquier fantasma (y 0 en resto).
        - O un canal por cada fantasma individual. Esto dependerá de cómo `GameDataCollector` lo hizo. Dado que la documentación habla de matrices (plural) para fantasmas, podría ser un canal por fantasma, o un solo canal con diferentes valores según el fantasma. Debemos decidir una representación consistente con los datos.
    - Posiblemente canales adicionales para estados de fantasmas: por ejemplo, si un fantasma está vulnerable, podría reflejarse poniendo un valor distinto (como 2) en la matriz de fantasmas, o usando otro canal que marque "fantasma X está comestible".
    - También se podría codificar todo en una sola matriz con distintos códigos numéricos para cada tipo (pared=0, comida=1, fantasma=2, Pac-Man=3, etc.), pero el diseño del modelo sugiere uso de **embedding por canal** (más adelante se detalla esto), por lo que es probable que se utilicen canales separados o codificación entera.

- Paso concreto: escribir una rutina de conversión:
    - Si el CSV tiene ya columnas como `wall_x_y`, `food_x_y`, etc., entonces armar matrices booleanas a partir de ellas es fácil (colocar True/False en coordenadas respectivas).
    - Si el CSV tiene una única larga lista de números representando la cuadrícula: por ejemplo `grid_0,0 ... grid_M,N`, podemos reconstruir una matriz del tamaño del laberinto con esos valores. Luego, esa matriz se puede separar en canales. Por ejemplo, crear:
    - `walls_matrix = (grid == WALL_CODE)` donde `WALL_CODE` es el número que representa muro en esa codificación.
    - `food_matrix = (grid == FOOD_CODE)`, etc. (Habría que conocer los códigos: tal vez 0 = vacío, 1 = pared, 2 = comida, 3 = cápsula, 4 = Pac-Man, 5-8 = fantasmas... esto es hipotético).
    - Utilizar los tamaños correctos: El tamaño del grid (M x N) puede obtenerse del layout. Por ejemplo, mediumClassic es 20x20 aprox. Podríamos deducirlo contando el número de columns relacionadas a grid en CSV (si hay M*N columnas).
- **Variables adicionales**: Si existen columnas extras en el CSV que no forman parte del grid, decidir su uso:
    - Podrían incluirse *características extra*, como:
    - `score` actual (no muy útil como input para decidir la acción, ya que es resultado acumulado).
    - `remainingFood` (comida restante) o `capsulesRemaining`.
    - Información temporal, p.ej. número de jugadas realizadas hasta ahora.
    - Estados de fantasmas (si no se codificó en la matriz).
    - Cualquier atributo relevante que no esté ya en la representación espacial se puede utilizar como **entrada adicional al modelo**. Por ejemplo, el número de alimentos restantes podría darle contexto a la red (saber si está cerca del final). Los tiempos de vulnerabilidad de fantasmas también son cruciales si disponibles (ej. 0 si ningún fantasma es comestible, o n ticks restantes si lo es).
    - Decidir qué extras usar: Usaremos un vector extra *Y* con estos atributos si aporta información. Esto corresponde al parámetro `extra_size` en la red (`PacmanNet`) según vimos en el código. Por ejemplo, si incluimos 2 valores (digamos, número de comida restante y algún indicador booleano de fantasma vulnerable), entonces `extra_size=2` y alimentaremos esos al modelo. Si no añadimos nada extra, `extra_size=0` (o simplemente no se usará ese camino en la red).

- Normalización: Dado que las entradas son principalmente binarias (0/1) o categóricas (índices de embeddings), no se requiere normalización numérica como tal. Si usáramos atributos continuos (p.ej., puntuación), podríamos escalarlos (quizá dividir la puntuación por 1000 para tener un valor más pequeño), pero en general no es crítico. Mantener los valores tal cual o en un rango consistente (0-1 o 0-n categorías).

- **Construcción de tensores de entrenamiento**: Convertir los datos preprocesados a tensores de PyTorch listos para entrenar:

- Crear la matriz de entradas **X** y el vector de etiquetas **y**:
    - **X**: idealmente será de dimensiones `(N_samples, C, H, W)` para los datos espaciales, donde `C` es el número de canales de entrada (p. ej., 5 si usamos [wall, food, capsule, ghost, Pacman]) y `H, W` son las dimensiones del laberinto. Además, si tenemos características extra, también habrá un vector asociado a cada muestra. Podremos manejarlo separadamente o integrarlo en X de alguna forma (por simplicidad, es común manejarlo como segunda entrada al modelo).

    - **y**: será un vector de tamaño `N_samples` con la acción tomada en cada estado, codificada numéricamente (0 a 3, si usamos 4 acciones posibles). Necesitaremos mapear la acción textual a un índice si en los datos aparece como texto ("North" -> 0, "South" -> 1, etc., siguiendo algún orden consistente). Si ya viene como número, verificar el mapeo a las acciones correctas.
- Utilizar pandas y numpy para esta conversión:
    - Leer el CSV combinado en un DataFrame. Luego, por cada fila (o mediante vectorización):
    - Obtener la representación matricial como explicado. Podríamos escribir código que recorra cada fila, construya las matrices numpy correspondientes y las agregue a una lista.
    - Convertir esa lista de matrices en un tensor 4D de PyTorch. Por ejemplo, `torch.tensor(X_np, dtype=torch.float32)`.
    - Lo mismo para y: mapear las acciones a enteros y luego `torch.tensor(y_np, dtype=torch.long)` para usar en clasificación.
    - Este proceso podría también hacerse de manera perezosa durante el entrenamiento (leyendo lote a lote), pero dado el tamaño moderado de datos, podemos cargarlo todo en memoria.

- **División entrenamiento/validación**: Es una buena práctica separar una parte de los datos para validación:

    - Por ejemplo, destinar un ~80% de las instancias para entrenar y ~20% para validar el rendimiento del modelo con datos que no se le mostraron al entrenar.
    - Realizar una partición aleatoria de los índices de las muestras. Asegurarse de que la distribución de acciones en ambos conjuntos sea similar (en general, si se hace aleatorio puro con suficientes muestras, estará bien).
    - Alternativamente, usar partidas enteras: por ejemplo, 8 partidas para entrenamiento, 2 para validación, para verificar que el modelo generaliza a partidas que no vio. Esto puede ser útil si las partidas tienen cierta coherencia interna (aunque en Pac-Man, todos siguen la misma dinámica, separar por partidas también es válido).
    - Separar X_train, y_train, X_val, y_val en tensores distintos.

- **Manejo de desequilibrios y limpieza**:

- Si notamos alguna irregularidad en los datos (por ejemplo, mucha más cantidad de acciones "East" que "West"), podríamos considerar técnicas de compensación:
    - *Submuestreo* de la clase mayoritaria o *sobremuestreo* de la minoritaria en el entrenamiento. Sin embargo, esto rara vez es necesario para acciones de movimiento, ya que normalmente Pac-Man utiliza todas las direcciones según necesite. Solo si algún movimiento es extremadamente raro podría valer la pena.
    - *Pesos en la función de pérdida*: PyTorch permite dar peso a cada clase en la pérdida si necesitáramos balancear.
- Eliminar o corregir outliers: Si hubiera filas inválidas (ejemplo: alguna acción imposible o estado vacío), filtrarlas. Por ejemplo, si por error se registró una acción "STOP" (quieto) que no es parte de nuestro espacio de acciones, se podría eliminar esas filas o mapear "STOP" a algún movimiento (aunque en Pac-Man estándar Pac-Man siempre se mueve).

- Confirmar que todas las entradas y salidas son coherentes: Ningún valor `NaN` o faltante. Si se detectan, remover o imputar (pero en nuestro caso es más seguro remover filas incompletas, ya que probablemente son pocas si existen).

- **Optimización del acceso a datos**:

- Si el conjunto es grande (miles de ejemplos), es útil crear un **DataLoader** en PyTorch que gestione los datos en *mini-lotes*. Esto se suele hacer definiendo un *Dataset* personalizado o usando `TensorDataset` con nuestros tensores:

```
from torch.utils.data import TensorDataset, DataLoader
dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

    - Decidir un tamaño de batch apropiado (32 es un buen comienzo). Batches más grandes aprovechan paralelismo pero consumen más memoria; dado un dataset pequeño, incluso 32 o 64 está bien.
    - Crear un `val_loader` similar para validación (podemos usar batch_size mayor ya que no se calcula gradiente allí, pero mantenerlo consistente no hace daño).

- Este enfoque de DataLoader no es obligatorio, pero es altamente recomendado para entrenamiento limpio y eficiente, y PyTorch facilita iterar así. Sin embargo, si el código base `net.py` ya implementa la lectura de CSV y el entrenamiento en algún formato (por ejemplo, tal vez lee todo a memoria y usa numpy), adaptaremos nuestro flujo a lo que ahí se espere. Revisar `net.py` para ver cómo está planteado el entrenamiento:

    - Puede que `net.py` tenga una función `load_data()` marcada con #ahmed que ya lee el CSV a arrays.
    - Si es así, quizás tengamos que **integrarnos con esa función**: es decir, asegurarnos de producir los archivos/formatos que esa función espera para que la training pipeline la use. Por ejemplo, si `net.py` espera un archivo `training_data.csv` específico, podríamos consolidar nuestros datos con ese nombre y ubicación.
    - En caso de que `net.py` esté incompleto y espere que el estudiante implemente la carga, seguiremos con nuestro plan de usar pandas/torch como arriba.

- **Resumen de preparación**: Al concluir esta fase, deberíamos contar con:

- Un tensor (o conjunto de tensores) de entrada X de dimensiones [N, C, H, W] (más extra features si aplican) listo para alimentar a la red.
- Un tensor de etiquetas y con tamaño [N], con valores en {0,1,2,3} correspondientes a las direcciones.
- Opcional: dataloaders configurados para facilitar el entrenamiento por lotes.
- Un conjunto de validación separado para evaluar rendimiento fuera de muestra durante el entrenamiento.

Realizar este procesamiento con cuidado garantiza que el modelo entrene sobre datos correctos y en el formato adecuado. Es una fase donde es útil imprimir dimensiones y quizá una instancia de ejemplo para comprobar que todo concuerda: por ejemplo, tomar la primera muestra X[0], visualizar su canales (imprimir la matriz de muros, comida, etc.) y verificar que corresponde al estado inicial de la partida y que la acción y[0] es la que realmente tomó Pac-Man. Este sanity check asegurará que no haya desalineamiento entre X e y.

4. Diseño del modelo de red neuronal (PacmanNet)

En esta fase definiremos la arquitectura de la red neuronal que controlará a Pac-Man. El modelo debe ser capaz de recibir la representación del estado del juego y producir una predicción de la **mejor acción** a tomar. Dado que Pac-Man se juega en un laberinto 2D con características espaciales (distribución de

muros, comida y entidades), utilizaremos una arquitectura que pueda **capturar la estructura espacial** de la situación.

A grandes rasgos, optamos por un modelo de red neuronal **convolucional** que procese la imagen del laberinto, complementado si es necesario con una pequeña subred para cualquier característica adicional (no espacial), y que finalmente produzca una clasificación sobre las acciones. La arquitectura propuesta para `PacmanNet` es la siguiente:

- **Entrada**:
- Un tensor de dimensiones `(C, H, W)` representando el estado del juego en forma de mapas de bits (canales). Por ejemplo, C=5 canales (muros, comida, cápsulas, fantasmas, Pac-Man). *Nota:* el número final de canales se ajustará según cómo codifiquemos el estado; podría ser ligeramente distinto (p.ej., separar fantasmas por canales, etc.).

- Un vector extra de tamaño `E` (extra features), opcional. Podría incluir información como el número de alimentos restantes, indicadores de fantasmas vulnerables, etc. (Si no se usa extra info, E=0).

- **Capas intermedias**: Usaremos embeddings y capas convolucionales para extraer características espaciales:

- *Embedding de celdas*: En lugar de alimentar directamente mapas binarios múltiples, podemos codificar la cuadrícula usando embeddings aprendibles. Según soluciones similares, por cada canal de entrada categórico se puede aplicar una capa de embedding que transforma los identificadores enteros de cada celda en vectores continuos de mayor dimensión [14] . En nuestro caso, podríamos aplicar:
    - Si hemos combinado el estado en una sola matriz con valores enteros (0=vacío, 1=muro, 2=comida, etc.), aplicar una capa `nn.Embedding(num_categories, embed_dim)` a esa matriz. Esto generaría *embed_dim* características por celda en lugar de 1, permitiendo a la red aprender representaciones útiles para cada tipo de objeto.
    - Alternativamente, si usamos canales binarios separados, el embedding podría no ser necesario; pero podríamos interpretar que *cada canal binario* es una categoría (presencia/ausencia) y también usar embed. Sin embargo, es más típico usar embedding cuando hay valores múltiples en una celda.
    - Dado que en la implementación referencia se menciona el uso de `nn.Embedding(9,16)` repetido por número de canales [15] , parece indicar que consideraron 9 posibles valores por celda y asignaron vector de 16 a cada valor, repetido para cada canal (posiblemente 4 canales, resultando en 64 valores). Esto sugiere que tal vez cada canal fue tratado inicialmente como una matriz con valores 0-8, aunque es un poco peculiar.
    - Para nuestro diseño, podemos simplificar: si tenemos canales binarios, podemos omitir embedding (ya es 0/1, podríamos dejar que conv aprenda a diferenciar). Si tenemos una sola matriz con múltiples codes, usaremos embedding para expandirla a mapas de características.
- *Capas convolucionales*: Después de la posible etapa de embedding/concatenación, aplicaremos una serie de capas CNN para extraer características:
    1. **Conv2d #1:** por ejemplo, 64 filtros, tamaño de kernel 8x8, *stride* 4. Esto recorrerá el laberinto capturando patrones locales amplios (8x8) y reduciendo la resolución (salida más pequeña por el stride). Activación ReLU.
    2. **Conv2d #2:** 64 filtros, kernel 4x4, stride 2. Captura patrones más finos ahora en la representación ya reducida. Activación ReLU.

3. **BatchNorm2d:** normalización batch sobre los 64 mapas de características para estabilizar el entrenamiento (especialmente útil si usaremos muchas épocas, previene covariate shift) [16] .
4. **Conv2d #3:** 64 filtros, kernel 3x3, stride 2. Sigue extrayendo características, posiblemente reduciendo la dimensión a muy pocos píxeles (incluso 1x1 dependiendo del tamaño inicial). Activación ReLU.
5. El resultado final de estas conv layers será un tensor de tamaño aproximado `(64, H_out, W_out)` donde H_out y W_out son muy pequeños (por ejemplo, si el laberinto es 20x20, después de conv1 stride4 -> ~5x5, conv2 stride2 -> ~3x3, conv3 stride2 -> ~1x1). Podríamos acabar con ~64 características en un "mapa" prácticamente comprimido.
6. **Flatten**: convertiremos esas características en un vector de tamaño 64 (asumiendo H_out * W_out ≈ 1, de lo contrario en 64*H_out*W_out). Este vector representa una abstracción del estado del juego (qué tan bueno es estar en X posición con Y distribución de fantasmas/comida, etc., encapsulado en 64 números).

• *Procesamiento de atributos extra*: Si disponemos del vector extra E (por ejemplo, E=2 con [food_remaining, any_ghost_vulnerable]), lo pasaremos por una pequeña capa densa para integrarlo:

- Una **capa lineal (encoder)** de tamaño `E -> 64` (tomamos el vector extra de longitud E y lo proyectamos a 64 dimensiones). Activación ReLU. Esto nos da otra representación de tamaño 64 con la información global extra.

• *Combinación*:

- Sumamos o concatenamos la información proveniente de las convoluciones y la del vector extra. La opción típica es **concatenar** los dos vectores de tamaño 64 (uno de conv, otro del encoder extra) obteniendo un vector de 128.
- Alternativamente, podríamos sumar si creemos que 64 features de conv ya son comparable con 64 de extra, pero la concatenación permite que las capas siguientes traten toda la info conjuntamente.
- Si concatenamos, la siguiente capa deberá tomar 128 de entrada; si sumamos (lo cual requeriría que ambos tengan la misma dimensión, 64, por eso escogimos 64), obtendríamos 64 combinados. En este diseño, optaremos por concatenar para mayor flexibilidad.

• **Capas de decisión (salida)**:

• Tras obtener la representación combinada del estado (dimensión 128 si concatenamos), usamos capas fully-connected (densas) para finalmente producir la predicción de acción:

1. **Capa oculta FC:** tamaño 128 -> 32. Comprime la información en algunas neuronas intermedias. Activación ReLU.
2. **Capa de salida FC:** tamaño 32 -> 4 (suponiendo 4 acciones posibles). Esta capa no tiene activación no lineal (o equivalently, aplica lineal + luego en training usaremos Softmax implícito en la función de pérdida).
3. El resultado son 4 valores reales, cada uno correspondiente a una *puntuación* o logit para una acción (Up, Down, Left, Right).
4. Durante el entrenamiento, estos logits se transformarán a probabilidades mediante *Softmax* para calcular la pérdida de entropía cruzada con la acción correcta. Durante el juego, el `NeuralAgent` usará estos valores como **confianza en cada acción** [10] . Normalmente elegirá la acción con mayor valor, ajustada por sus heurísticas.

En resumen, la arquitectura **PacmanNet** propuesta se resume en la siguiente tabla:

Capa	Detalles / Dimensiones
Entrada	Tensor estado `C×H×W` (canales del mapa de Pac-Man) + vector extra tamaño *E* (opcional)
Embeddings (opcional)	`nn.Embedding(n_cat, 16)` para cada canal categórico, produce 16 mapas por canal. Si C inicial = 4 y n_cat=9, salida = 64 canales [15].
Conv2D #1	64 filtros, kernel 8×8, stride 4. Salida: 64×(H/4)×(W/4). Activación ReLU.
Conv2D #2	64 filtros, kernel 4×4, stride 2. Salida: 64×(H/8)×(W/8). ReLU.
BatchNorm2D	Normaliza 64 mapas de características [16]. Estabiliza entrenamiento.
Conv2D #3	64 filtros, kernel 3×3, stride 2. Salida: 64×(H/16)×(W/16) (aprox 64×1×1). ReLU.
Flatten	Aplana las características espaciales a vector de 64 (asumiendo H_out*W_out ≈ 1).
Encoder extras (FC)	Lineal: R^E -> R^64. Toma vector extra (E dim) a 64-dim. ReLU. (Omitir si E=0).
Combinar representación	Concatenar vector de conv (64-dim) + vector extra codificado (64-dim) -> 128-dim (si hay extras; si no, sigue con 64-dim de conv).
FC oculta	Lineal: 128 -> 32 (o 64->32 si no había extras). ReLU.
FC salida	Lineal: 32 -> 4. Sin activación (logits para 4 acciones).

Algunos detalles adicionales del diseño y motivaciones:

- Usar **convoluciones** permite que el modelo aprenda patrones locales importantes, por ejemplo: "fantasma cerca de Pac-Man en cierta dirección", "cápsula disponible cerca", "corredor largo sin comida", etc. Las capas con stride reducen el mapa gradualmente, agregando invarianza espacial (p.ej., detectar un fantasma a 2 pasos de Pac-Man independientemente de la posición exacta en el laberinto).
- El número de filtros (64) en cada conv capa es elegido para tener suficiente capacidad sin exagerar (el modelo resultante no es muy grande: unas pocas decenas de miles de parámetros quizás). Podríamos experimentar con más o menos filtros según el rendimiento y sobreajuste.
- La **BatchNorm** ayuda a acelerar la convergencia y permite usar tasas de aprendizaje un poco mayores al controlar la dispersión de activaciones. No es estrictamente necesaria pero suele mejorar estabilidad.
- Se utilizan **ReLU** como función de activación intermedia por su rendimiento comprobado en muchas redes profundas, evitando saturación.
- La decisión de concatenar el vector extra (si existe) permite al modelo considerar información global junto con la espacial. Por ejemplo, si *food_remaining* es parte del vector, la red puede calibrar su comportamiento: con poca comida restante quizás arriesgue más; con fantasmas vulnerables = True, puede interpretar los patrones de conv de fantasmas de manera distinta.
- La salida de 4 neuronas corresponde a una **clasificación multiclase** de la acción a tomar. Entrenaremos el modelo con una función de pérdida de **Entropía Cruzada** (CrossEntropyLoss) comparando estas logits con la acción correcta (como índice 0-3). Esta pérdida internamente aplicará Softmax, convirtiendo las logits en probabilidades p(acción|estado). El entrenamiento ajustará los pesos para maximizar la probabilidad de la acción correcta observada en los datos.
- Durante la inferencia en el juego, el `NeuralAgent` tomará las logits de salida. Posiblemente:

- Las normalizará (puede usar softmax o simplemente comparar magnitudes).
- Las combinará con heurísticas (sumando algún valor si la acción conduce a comida inmediata, restando si lleva hacia fantasma, etc. como mencionamos).
- Luego puede elegir la acción con mayor valor final, con probabilidad 1 - ε, y con probabilidad ε (exploración) elegir una aleatoria [11].
- La existencia de heurísticas adicionales significa que **no delegamos el 100% de la decisión a la red**, lo cual puede ser positivo en caso de que el modelo entrenado no sea perfecto; las heurísticas actuarán de red de seguridad (por ejemplo, aunque la red no reconozca un peligro, la heurística de "fantasma cercano" podría aún prevenir un mal movimiento).
- **Regularización**: Si nos preocupa el sobreajuste (por ejemplo, si tenemos pocos datos), podríamos añadir técnicas de regularización:
- Dropout layers en las capas densas (p.ej., dropout del 50% antes de la capa oculta de 32).
- L2 weight decay en el optimizador.
- Por simplicidad inicial, podríamos no incluir dropout aún y evaluar si el modelo sobreajusta; en caso afirmativo, añadirlo después.
- **Tamaño del modelo**: En parámetros, estimamos:
- Embedding: 9*16 * C (ej. 9*16*4=576) si usado.
- Convs: cada conv tiene 64 filtros * (entrada_maps * kernel_size) + bias. Por ejemplo, conv1: 64 * (C_in * 8 * 8) ~ 64 * (64 * 64) if C_in=64 post-embedding, que es 64*4096 = ~262k params, conv2: 64*(64*4*4)=64*1024=65k, conv3: 64*(64*3*3)=64*576=36k. FC: 32*(128)+32 biases=4128, Output: 4*32+4=132. Sumando, da ~364k parámetros, lo cual es manejable. (Si no usamos embedding y C_in=5 actual, conv1 sería 64*(5*8*8)=64*320=20k, más pequeño aún).
- Este tamaño es razonable para entrenar rápidamente en CPU/GPU sobre unos miles de ejemplos.

Antes de implementar, revisaremos si alguna parte de esta arquitectura ya estaba sugerida o parcialmente implementada en `net.py`. Dado que la documentación menciona que `PacmanNet` se implementa con PyTorch [6], es posible que en el código haya un esqueleto de la clase con algunas capas definidas (quizá incluso la arquitectura exacta que describimos, marcada con #ahmed en partes). Si eso es así: - **No duplicar**: Usaremos las definiciones existentes y solo completaremos donde falte. - Por ejemplo, si la clase `PacmanNet` ya tiene una parte del `__init__` con ciertas capas, seguiremos ese diseño. Si coincide con lo que planeamos (embedding, convs, etc.), perfecto. Si difiere, adaptaremos nuestro plan: - Podría ser que la implementación base optara por una red más simple (por ejemplo, solo densas sin conv). En tal caso, dado que conv se recomienda, es posible que esperar que el estudiante llene con conv layers; seguiremos nuestro plan conv asumiendo es lo esperado. - Podría ser ya igual a lo que describimos, en cuyo caso nuestra tarea es principalmente implementar el `forward()` correctamente para encadenar las capas, y luego la rutina de entrenamiento.

En conclusión, el diseño de `PacmanNet` está pensado para **captar relaciones espaciales en el laberinto** y **aprender patrones de juego** a partir de las demostraciones. Con este diseño establecido, pasaremos a implementarlo efectivamente en el código y luego entrenarlo con los datos recopilados.

5. Implementación del modelo y del proceso de entrenamiento

En esta fase llevaremos a código el modelo diseñado y prepararemos la rutina de entrenamiento en el archivo `net.py`. Es fundamental implementar todo **respetando las restricciones**: no modificar las secciones con `#ahmed` y mantener la integración con el resto del sistema. Las tareas a realizar incluyen:

- **Implementar la clase `PacmanNet` en `net.py`**:

- Abrir `net.py` y localizar la definición de `class PacmanNet(nn.Module)`. Rellenar/ajustar su método `__init__` y `forward` según la arquitectura decidida:
  - En `__init__`:
  - Definir las capas necesarias como atributos de la clase. Por ejemplo:

```
class PacmanNet(nn.Module):
    def __init__(self, input_channel_num, num_actions,
extra_size):
        super(PacmanNet, self).__init__()
        # Definición de capas
        self.channels = input_channel_num
        # Embedding: si necesitamos, e.g.
        # self.embeddings =
nn.ModuleList([nn.Embedding(num_categories, 16) for _ in
range(input_channel_num)])
        # self.conv1 = nn.Conv2d(64, 64, kernel_size=8, stride=4)
        self.conv1 = nn.Conv2d(self.channels * 16 if use_embedding
else self.channels, 64, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(64, 64, kernel_size=4, stride=2)
        self.bn = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=2)
        self.encoder = nn.Linear(extra_size, 64) if extra_size > 0
else None
        # Calcular tamaño de salida de convs:
        # (Podemos hacerlo dinámicamente con un dummy tensor o
manual si conocido)
        conv_out_size = 64  # asumiendo sale 1x1; si no,
multiplicar dims
        self.fc1 = nn.Linear(conv_out_size + (64 if extra_size > 0
else 0), 32)
        self.fc2 = nn.Linear(32, num_actions)
    ...
```

    La idea es codificar todo con los tamaños correctos. Podemos comprobar las dimensiones calculando el tamaño de salida de convs:
    - Una forma: en tiempo de init, crear un tensor ficticio `torch.zeros(1, input_channel_num_embed, H, W)` (donde H,W del layout, o suponer uno) y pasarlo por conv layers secuencialmente para ver el output shape. Sin embargo, el layout puede variar, mejor suponer el peor caso (largest H,W).
    - Dado que el proyecto parece centrado en unos layouts específicos, podríamos incluso codificar H=W=... pero preferible derivar. Para simplicidad, supondremos conv_out_size = 64 (lo cual implica que tras conv3 el output is 1x1, lo que es plausible para mediumClassic; si no fuera exacto, conv_out_size = 64 * H_out * W_out).
  - Asegurarse de que las definiciones de capas correspondan al modelo. Si la plantilla en el código base ya define algunas, extender esas:
    - Posible situación: algunas capas ya definidas con `#ahmed` en la init. Por ejemplo, es posible que la arquitectura esté parcialmente allí y solo falta implementarla. Si encontramos comentarios como `# TODO: definir conv layers` (sin #ahmed) entonces procedemos según diseño.

        - No borrar ni modificar nada marcado. Por ejemplo, si `PacmanNet.__init__` tiene ciertas cosas ya hechas, complementamos sin tocar las ya existentes.
    - En `forward(self, x, extra)`:
    - Implementar el paso hacia adelante:

```
def forward(self, x, extra=None):
    # x: tensor de tamaño (batch, input_channel_num, H, W)
    # extra: tensor de tamaño (batch, extra_size)
    if use_embedding:
        # aplicar embedding a cada canal
        embed_outs = []
        for i, emb in enumerate(self.embeddings):
            # x[:, i, ...] es un mapa de enteros
            out = emb(x[:, i, ...].long())   # obtiene (batch, H,
W, 16)
            # Permutar para que la dimensión embed de 16 sea
tratada como canal:
            out = out.permute(0, 3, 1, 2)    # (batch, 16, H, W)
            embed_outs.append(out)
        x_emb = torch.cat(embed_outs, dim=1)  # concatenar en
canal -> (batch, 16*C, H, W)
        h = x_emb
    else:
        h = x  # si no usamos embedding y x ya viene con canales
apropiados
    h = F.relu(self.conv1(h))
    h = F.relu(self.conv2(h))
    h = self.bn(h)
    h = F.relu(self.conv3(h))
    # flatten conv output
    h = h.view(h.size(0), -1)  # (batch, conv_out_size)
    # encode extras if present
    if self.encoder is not None and extra is not None:
        extra_enc = F.relu(self.encoder(extra))
        h = torch.cat([h, extra_enc], dim=1)
    # fully connected layers
    h = F.relu(self.fc1(h))
    out = self.fc2(h)
    return out
```

    Nota: Aquí `F` refiere a `torch.nn.functional` para usar ReLU. Alternativamente, podríamos definir `self.relu = nn.ReLU()` in init y luego hacer `h = self.relu(self.conv1(h))` etc.
    - Hay que manejar cuidadosamente tipos de datos: si usamos embedding, convertimos x a long. Si x originalmente viene ya como tensor float de 0/1, tendríamos que convertirlo a int índices antes de embeding. O, más fácil: hacer que `GameDataCollector` guarde el grid como ints (lo probable) y al cargar mantenemos eso en un tensor Long para el embedding.

- Nuevamente, respetar `#ahmed`: es posible que en forward alguna parte esté dada. Por ejemplo, quizás la concatenación final o la aplicación de softmax.
  - Muchos profesores dejan la activación final fuera (pues CrossEntropyLoss se encarga). Si vemos un comentario de no aplicar softmax, seguirlo.
  - En caso de duda, *no aplicar softmax en forward*, ya que PyTorch `CrossEntropyLoss` espera logits. Confirmar si en entrenamiento se usa esa loss.
- Comentar el código para clarificar qué hace cada paso, esto ayuda en el entregable (mostrar buenas prácticas).

- **Ejemplo de integración con el código existente**: Supongamos en `net.py` ya había:

```
class PacmanNet(nn.Module):
    def __init__(self, input_channel_num, num_actions, extra_size):
        super().__init__()
        # ahmed: architecture skeleton
        # define conv and fc layers here...
```

Si la línea está marcada como `# ahmed: architecture skeleton`, podríamos completar debajo, pero es posible que toda la función **init** sea a implementar. En cualquier caso, asegurarnos de no romper nada que estuviera pre-escrito.

- **Implementar la rutina de entrenamiento en** `net.py`: Es probable que `net.py` tenga una estructura con una función `train_model()` o incluso que al ejecutar `python net.py` se lance el entrenamiento (según el README, `python net.py` entrena la red [17]). Debemos rellenar este flujo:

- **Carga de datos**: Si no existe, escribir código para:
  - Leer el archivo(s) CSV de `pacman_data/` con pandas (o utilizar numpy loadtxt).
  - Convertir a tensores (según lo ya hecho en fase de preparación).
  - Podríamos reutilizar las variables `X_train, y_train, X_val, y_val` generadas en la fase previa si movemos ese código aquí. Probablemente, implementaremos dentro de `if __name__ == "__main__":` en net.py un bloque que:
  - Llama a una función (ej. `load_data()`) que lee los CSV y devuelva los tensores/ dataloaders. Esta función podríamos crearla.
  - Construye una instancia de PacmanNet con los parámetros adecuados (input_channel_num = número de canales de entrada que definimos, num_actions=4, extra_size = tamaño vector extra).
  - Llama a la función de entrenamiento pasando el modelo y los datos.
- **Configuración de entrenamiento**: Definir hiperparámetros básicos:
  - *Learning rate* (tasa de aprendizaje), p.ej. 0.001 como inicio (o 0.01 dependiendo de normalización).
  - *Número de épocas*: por ejemplo 50 o hasta que converja. Dado el dataset no es gigantesco, podemos permitir bastantes épocas. Sin embargo, hay que vigilar sobreajuste. Podríamos empezar con 20 y ver, y aumentar si la pérdida sigue bajando bien y la validación mejora.
  - *Optimizador*: usar **Adam** (es una buena elección para empezar, adaptativa y generalmente eficaz). Configurarlo con lr escogido, y opcionalmente weight_decay si queremos L2 (por ejemplo 1e-5).

- *Criterio de pérdida*: `nn.CrossEntropyLoss()`, que es apropiado para clasificación multi-clase con logits de salida.
- *Batch size*: si usamos DataLoader, definimos arriba (por ejemplo, 32).
- *Dispositivo (CPU/GPU)*: Si disponemos de GPU, mover el modelo y datos a GPU (e.g., `device = torch.device('cuda' if torch.cuda.is_available() else 'cpu'); model.to(device)`). Si no, quedará en CPU.

- **Loop de entrenamiento**: Implementar el ciclo típico: `python`

```
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for batch_X, batch_y in train_loader:
            batch_X, batch_y = batch_X.to(device), batch_y.to(device)
            optimizer.zero_grad()
            # Si hay extra vector separado, separarlo here: e.g.,
batch_X_spatial, batch_extra = batch_X
            outputs = model(batch_X, batch_extra)  # adapt según input
format
            loss = criterion(outputs, batch_y)
            loss.backward()
            optimizer.step()
            running_loss += loss.item() * batch_X.size(0)
        epoch_loss = running_loss / len(train_dataset)
        ...
        # Calcular métricas de validación cada cierta cantidad de épocas:
        model.eval()
        correct = 0; total = 0; val_loss = 0.0
        with torch.no_grad():
            for batch_X, batch_y in val_loader:
                batch_X, batch_y = batch_X.to(device), batch_y.to(device)
                outputs = model(batch_X, batch_extra)
                # calcular val_loss si se desea, aunque principalmente nos
interesará accuracy
                loss = criterion(outputs, batch_y)
                val_loss += loss.item() * batch_X.size(0)
                _, predicted = torch.max(outputs, 1)
                total += batch_y.size(0)
                correct += (predicted == batch_y).sum().item()
        val_acc = correct / total
        val_loss = val_loss / len(val_dataset)
        print(f"Epoch {epoch+1}/{num_epochs} - Loss: {epoch_loss:.3f} - Val
Acc: {val_acc:.3f}")
```

  - Ajustar este pseudocódigo a la realidad de nuestros datos: por ejemplo, si no usamos DataLoader (cargamos todo en tensores), podríamos implementar mini-batch manual con slicing en numpy/torch.
  - Imprimir información por época ayuda a monitorear. Podríamos también imprimir `val_loss` si queremos observar la tendencia, o incluso la precisión de entrenamiento similar a val.
  - Como se trata de un proyecto académico, es valioso reportar tanto la **pérdida (loss)** como la **precisión (accuracy)** en predicción de la acción. La precisión nos dice qué porcentaje de acciones el modelo acierta exactamente igual que el jugador humano en

datos de validación. No será 100% y no necesita serlo; pero cuanto más alta, mejor estará imitando.
    - Quizá más relevante es evaluar el desempeño en el juego real, pero eso se hace después. Aquí la accuracy nos sirve de indicativo.
    - Usar `model.train()` y `model.eval()` correctamente para activar/desactivar dropout (si tuviésemos) y batchnorm comportamientos apropiados.
    - Implementar también una condición de **early stopping** opcional: por ejemplo, si la pérdida de validación no mejora en X épocas, detener para no sobre-entrenar. Dado el tamaño de datos, podríamos simplemente entrenar un número fijo de épocas relativamente pequeño para evitar sobreajuste (a menos que veamos que sigue mejorando).
- **Entrenamiento e iteración**: Ejecutar este entrenamiento. Posibles ajustes durante esta sub-fase:
    - Si la pérdida no desciende o la accuracy se estanca muy baja, puede ser señal de tasa de aprendizaje inadecuada (demasiado alta o demasiado baja). Ajustar lr: subirlo si la pérdida cae lentamente (o está casi plana), bajarlo si oscila mucho o aumenta.
    - Si se nota overfitting (accuracy de entrenamiento muy alta pero validación baja), se puede:
    - Introducir dropout (por ejemplo, `nn.Dropout(0.5)` después de conv flatten o después de fc1).
    - Aumentar peso de regularización L2 en Adam.
    - Conseguir más datos, si posible.
    - Monitorear que el modelo no esté memorizando solo secuencias específicas. Pero dado que los estados de Pac-Man tienen cierta generalidad, esperamos que aprenda patrones útiles.
    - **Tiempo de entrenamiento**: Con ~2000 muestras y este tamaño de red, entrenar 20-50 épocas es cuestión de segundos en CPU o menos de un par de minutos, muy aceptable. Así que se pueden probar varios entrenamientos rápidamente.
- **Guardar el modelo entrenado**:
    - Una vez satisfechos con el entrenamiento (por ejemplo, tras la última época), guardar los pesos del modelo en un archivo para uso del agente:

      ```
      torch.save(model.state_dict(), "models/pacman_net.pth")
      ```

      Confirmar la ruta: probablemente el repositorio tiene un directorio `models/` ya creado para esto. De hecho, en la estructura se veía `models/` carpeta [18], así que usarla. Si el código base define un nombre distinto (ver si en `NeuralAgent` hay algo como `load("models/XYZ.pth")`), usar ese nombre exactamente.
    - También podríamos guardar la estructura (state_dict es suficiente ya que la clase definimos igual).
    - Marcar claramente en el código o output que el modelo fue guardado exitosamente.
    - Asegurarnos de **no modificar** la ruta si está marcada con #ahmed. Por ejemplo, si había una constante `MODEL_PATH = "models/pacman.model"` marcada, usar esa.
- **Lógica condicional en** `net.py`:
    - Normalmente, pondremos el entrenamiento dentro de `if __name__ == "__main__":` de modo que al ejecutar `python net.py` se ejecute el entrenamiento. Esto parece ser la intención (según README).
    - Si hay un parser de argumentos (podría ser, para opciones como dataset path, epochs, etc.), configurarlo. Si no, hardcode valores o añadir un par simple si es útil (por ejemplo, `--epochs`).

    - No obstante, dado que no se menciona, probablemente no hace falta complejidad extra; con los hiperparámetros definidos en código basta.

- **No modificar secciones #ahmed**:

    - Es posible que algunas partes (por ejemplo, definition of PacmanNet or parts of training loop) ya estén implementadas. Si alguna parte del entrenamiento ya estaba escrita por el profesor y marcada con #ahmed, por ejemplo un esqueleto, usarla tal cual:
    - Quizá la función de carga de datos y guardado de modelo ya existe con #ahmed. Si es completa, úsala (e.j., quizás lee "pacman_data/training_data.csv" y llama a `torch.save` etc.). Entonces adaptar nuestros datos para que encajen.
    - Si la ruta del modelo a guardar está especificada como constante (p.ej., `MODEL_FILE = "./models/model.pth" #ahmed`), no cambiarla, solo asegurarse de guardar ahí.
    - Posiblemente, se espera del estudiante principalmente la implementación de la red (capas) y el bucle de entrenamiento, pero la infraestructura (cargar, guardar) puede estar dada.
    - En cualquier caso, revisaremos antes de escribir todo desde cero no duplicar código.
    - Documentar en comentarios (sin #ahmed) lo que hacemos para claridad, pero ser conciso.

- **Realizar una prueba rápida de entrenamiento**: Antes de dar por completada esta fase, conviene hacer una pequeña prueba (si es posible, incluso con un subconjunto de datos):

- Ejecutar `python net.py` y verificar que:
    - La red se construye sin errores de dimensión.
    - Los datos se cargan correctamente y la iteración comienza.
    - Las pérdidas se reducen y la validación muestra alguna tendencia positiva.
    - El modelo se guarda.
- Si surge un error (por ejemplo, *size mismatch* en capas), corregir calculando bien la dimensionalidad. Un error común podría ser que H_out*W_out de conv no sea 1, y entonces `conv_out_size` de flatten debería multiplicar esos. Solución rápida: imprimir `h.shape` antes de flatten en forward con un dummy input, o usar `torch.nn.Flatten()` adaptativo (pero en forward .view es suficiente si calculamos).
- Verificar también que la GPU (si se usa) está utilizándose (aunque no es obligatorio usar GPU, si no hay, CPU está bien para dataset pequeño).
- No es necesario entrenar completamente en esta prueba, solo unos pocos batches para ver que loss disminuye, luego se puede detener y ajustar cualquier bug.

Al finalizar la implementación, tendremos un archivo `net.py` con la clase `PacmanNet` correctamente definida y una rutina de entrenamiento que produce un modelo guardado. Esta es la parte central de desarrollo del proyecto. A continuación, procederemos a **probar y evaluar** el modelo entrenado dentro del juego Pac-Man para comprobar su desempeño.

6. Pruebas y validación del agente entrenado

Con el modelo entrenado disponible, el siguiente paso es **probar al agente neuronal en el entorno de Pac-Man** para evaluar qué tan bien juega y si efectivamente maximiza la puntuación mejor que agentes no entrenados. Las pruebas incluirán ejecutar partidas automáticas con el `NeuralAgent` usando el

modelo aprendido, y posiblemente comparar resultados con alguna referencia. Las tareas a realizar en esta fase:

- **Integrar el modelo entrenado en el juego**: Esto implica asegurarse de que `NeuralAgent` carga el archivo de modelo que acabamos de entrenar:
- Revisar la implementación de `NeuralAgent` en `multiAgents.py` respecto a la carga de modelo. Probablemente en su `__init__` o `registerInitialState` haya algo como:

```
self.model = PacmanNet(...);
self.model.load_state_dict(torch.load(MODEL_PATH))
self.model.eval()
```

  Verificar la ruta `MODEL_PATH` usada y confirmar que coincide con donde guardamos nuestro modelo (e.g., `"models/pacman_net.pth"`). Si no, renombrar nuestro archivo o ajustar la variable (si no está protegida por #ahmed).
    - Si, por ejemplo, la ruta está codificada como `models/pacman.model`, podríamos simplemente guardar también en ese nombre (además del nuestro) para no tener que cambiar código. Lo importante es que **NeuralAgent encuentre el archivo**.
- Confirmar que la arquitectura del modelo cargado coincide exactamente con la que `NeuralAgent` espera. Si definimos extra_size y channels etc., `NeuralAgent` al instanciar PacmanNet debe usar los mismos valores:
    - Ver qué argumentos le pasa: probablemente `PacmanNet(input_channel_num, num_actions, extra_size)`. Tenemos que asegurarnos de conocer `input_channel_num` (número de canales de entrada que decidimos; e.g., 5). Posiblemente `GameDataCollector` define eso; quizás lo pasa también.
    - Si `NeuralAgent` construye `PacmanNet` con ciertos números fijos, debemos haber entrenado con la misma config. Por ejemplo, si `NeuralAgent` usa `PacmanNet(5, 4, 0)` (5 canales, 4 acciones, no extra), nuestro modelo guardado debe ser con esos parámetros. En nuestro entrenamiento, nosotros instanciamos PacmanNet con ciertos numbers (p.ej., 5,4,0), por lo que el state_dict se corresponde. Si por alguna razón difiere (por ejemplo, entrenamos con extra_size=2 pero el agent no le pasa esas 2), habría inconsistencia. Por lo tanto, **sincronizar la definición**:
    - Si no añadimos extras, probablemente extra_size=0 en agent y todo bien.
    - Si añadimos extras, asegurarse que `NeuralAgent` cuando llama a la red también le pase esos extras. Si no lo hace, podríamos modificar NeuralAgent para alimentar las extras (pero eso podría ser #ahmed). Más seguro es no depender de extras a menos que sabamos integrarlo. Por simplicidad podríamos optar por no usar extra features explícitas, y codificar toda info necesaria en la representación matricial. Esto evita tocar NeuralAgent.
    - Decisión: Si `extra_size` es no cero, necesitamos ver cómo NeuralAgent.forward llama al modelo. Quizás llama `model(state_matrix_tensor, extra_tensor)`. Si no se contempla extra en NeuralAgent, mejor no usarlo. *Por precaución, podríamos set extra_size=0* en final, a menos que la documentación o código indiquen que se usa.
- Luego de cargar, `NeuralAgent` usará `model.predict()` o forward outputs en su lógica.

- **Ejecutar partidas de prueba con el NeuralAgent**:

- En la consola, ejecutar: `python pacman.py -p NeuralAgent -n 5 -q`:
    - `-p NeuralAgent` indica usar nuestro agente neuronal.

- `-n 5` ejecuta 5 juegos consecutivos automáticamente.
- `-q` (quiet) puede silenciar la salida detallada, pero muestra resultados finales. Esto permite obtener promedios.
- Se pueden variar otros parámetros:
- `-l layoutName` para probar en diferentes laberintos.
- `-g GhostType` para probar distintos comportamientos de fantasmas (p.ej., `-g RandomGhost` que ya es default random, o si hay uno que sigue a Pac-Man).
- Si no se quiere quiet, se puede observar visualmente uno o dos juegos sin `-q` para ver cómo se mueve Pac-Man.

• **Observar el desempeño**:
- Verificar que no ocurran errores al cargar el modelo ni durante la ejecución.
- Observar el comportamiento de Pac-Man:
- ¿Juega razonablemente? (Come la comida de forma eficiente, huye de fantasmas cuando debe, usa cápsulas).
- Comparar subjetivamente con un humano o con su comportamiento en las partidas de entrenamiento. Debería imitar patrones aprendidos. Puede tener algunas fallas si se enfrenta a situaciones no vistas.
- Registrar la **puntuación obtenida en cada partida de prueba** y si Pac-Man gana (come todo) o pierde (muere). Por ejemplo, anotar:
- Juego 1: Score 800, Pac-Man muere con 2 comidas restantes.
- Juego 2: Score 1040, Pac-Man gana (todas comidas).
- ... etc.
- Calcular el **promedio de puntuación** en esas partidas y la tasa de victorias. Esto servirá como métrica de desempeño.

• **Comparación con baseline**:
- Para valorar la mejora, conviene comparar con algún agente simple:
- *Agente Aleatorio*: ejecutar `python pacman.py -p RandomAgent -n 5 -q` en el mismo layout/ghost setup y obtener el promedio de score. Seguramente será mucho menor (p.ej., un random Pac-Man suele morir rápido con baja puntuación).
- *Agente Reflex/Minimax* (si disponible): Si multiAgents.py tiene un `ReflexAgent` (basado en heurística simple), probarlo para comparar. O si hay Minimaxes implementados (a veces no se completan en este proyecto, pero si existiese, se puede probar).
- Comparar: idealmente nuestro NeuralAgent debería superar claramente a RandomAgent en puntaje y quizás acercarse o igualar a ReflexAgent si existiera uno.
- Un cuadro comparativo podría presentarse así: | Agente | Puntuación media (5 juegos) | Descripción | |-------------------|----------------------------|---------------------------------| | **NeuralAgent (Ours)** | 900 puntos | (gana 3/5 juegos, muere en 2) | | RandomAgent | 200 puntos | (siempre muere temprano) | | ReflexAgent *(if)* | 850 puntos | (heurístico básico, gana 2/5) | *(Estos números son hipotéticos a modo de ilustración.)*
- Este tipo de comparación evidenciaría que nuestro agente neuronal efectivamente mejora el rendimiento respecto a no inteligencia, y es competitivo con heurísticas. Si incluso supera a los agentes clásicos, sería un gran éxito, pero lo usual es que Minimaxes profundos puedan ser muy fuertes (aunque no implementados, no comparar con inhumano).

• **Monitorizar el comportamiento específico**:

- Confirmar que el agente no viola reglas: Pac-Man no debería quedarse quieto (a menos que acción Stop existiera, pero usualmente no la consideran). Ver que siempre elige algún movimiento.

    - Prestar atención a casos de corner: ¿Se queda dando vueltas sin razón? Eso podría pasar si la red cae en un bucle por datos insuficientes. Si notamos comportamientos subóptimos repetitivos, anotarlos.
    - Ver si el agente **aprendió a usar cápsulas**: Por ejemplo, si un fantasma se acerca, Pac-Man podría correr hacia una cápsula y luego perseguir fantasmas vulnerables. Eso sería signo de buen aprendizaje. Si en cambio siempre huye incluso con cápsula disponible, tal vez no tuvo ejemplos suficientes de usar cápsulas (podríamos en futuros datos jugar más para enseñarlo).
    - Chequear la **exploración**: Al principio de las pruebas, `NeuralAgent` podría hacer movimientos extraños (exploratorios) si la tasa epsilon está alta. Pero la doc dice "decaying exploration rate" [11], puede que si jugamos muchas partidas seguidas (-n 50), se note que con el tiempo Pac-Man se vuelve más determinista. En 5 partidas no es mucho decaimiento, pero algo puede notarse. En todo caso, para evaluación, convendría fijar exploración baja cuando medimos desempeño (quizás tras bastantes juegos, o si la variable epsilon se puede inicializar más baja manualmente).
    - Si resultados son muy aleatorios por la exploración, considerar reducirla para la evaluación (podría requerir editar un parámetro en NeuralAgent, solo para test, pero si #ahmed no se debe... quizás correr 50 partidas y tomar las últimas 40 como donde exploitation domina).

- **Evaluar resultados frente al objetivo**: El objetivo era *maximizar la puntuación en Pac-Man*. Evaluar si:

- Pac-Man sobrevive más tiempo que antes (respecto a random/humano inicial).
- Come más puntos en promedio.
- Posiblemente logra acabar niveles (si en training se mostraron suficientes finales).
- Si hay aspectos subóptimos, identificarlos:
    - Por ejemplo, quizás Pac-Man aprende bien a recolectar comida pero a veces se suicida contra un fantasma porque los datos de entrenamiento no cubrían esa situación lo suficiente. Esto indicaría una posible mejora: más datos con ese escenario o ajustar heurística anti-fantasma.
    - O tal vez prioriza fantasmas vulnerables demasiado y muere; entonces la heurística podría penalizarlo.

- En general, si la puntuación promedio es razonablemente alta, podemos considerar que cumplimos el objetivo. Cualquier margen de mejora se puede abordar en la siguiente fase.

- **Validación cuantitativa (opcional)**:

- Para tener datos más robustos, se puede automatizar 100 partidas con `-n 100 -q` y sacar promedio. Esto suaviza variaciones aleatorias de fantasmas (que al ser aleatorios, pueden hacer algunas partidas más fáciles/difíciles).
- Además, observar la desviación estándar de las puntuaciones para entender la consistencia del agente.
- Si se quisiera, graficar la evolución de la exploración vs performance si la tasa decae.

- Pero dado el alcance, con unos 10-20 juegos de prueba manual es suficiente para conclusiones básicas.

- **Ejemplos cualitativos**: Podríamos incluir en el informe una breve descripción de un caso de éxito:

- *Ejemplo:* "En una partida de prueba, nuestro Pac-Man entrenado logró una puntuación de 1200 puntos, completando el nivel. Se observó que el agente deliberadamente fue primero por las zonas con mucha comida fácil, luego tomó una cápsula cuando los fantasmas rodeaban el área final de comida, volviendo a comérselos y limpiando el mapa. Este comportamiento refleja estrategias similares a las de un jugador humano experto, demostrando que la red neuronal capturó pautas de juego eficaces."
- Y también un caso de fallo:
- *Ejemplo:* "En otra partida, Pac-Man quedó acorralado al ir a por un punto restante mientras un fantasma lo perseguía, resultando en la pérdida de una vida. Es posible que el modelo no anticipara correctamente el riesgo en esa situación, quizás por falta de ejemplos suficientes donde sacrificar un punto de comida para salvar la vida era la mejor opción."
- Estos insights ayudan a entender las fortalezas y debilidades del agente.

En resumen, durante esta fase comprobaremos en la práctica el desempeño del agente entrenado. Si los resultados son positivos (puntuaciones altas, victorias frecuentes), pasaremos a la finalización. Si identificamos problemas serios (p. ej. el agente se comporta aleatorio -> indica fallo en entrenamiento o carga, o agente siempre muere temprano -> indica necesidad de más entrenamiento/datos), entonces saltaremos a la fase de ajuste y refinamiento.

7. Ajustes y refinamiento iterativo (si es necesario)

Tras las pruebas iniciales, podríamos requerir refinar el modelo para mejorar su desempeño. El desarrollo de un agente inteligente es a menudo un proceso iterativo de mejora. En esta fase, según los hallazgos de la validación, aplicaremos ajustes en uno o varios frentes:

- **Recolección de más datos de entrenamiento**: Si el agente muestra deficiencias claras en ciertas situaciones, lo más directo puede ser volver a generar datos que cubran esos casos:
- Por ejemplo, si Pac-Man tiende a chocar con fantasmas al no aprender a evitarlos correctamente, jugar manualmente partidas haciendo especial hincapié en esquivar fantasmas en el último momento, para darle ejemplos de esa conducta.
- Si el agente ignora las cápsulas de poder, generar partidas donde se utilicen correctamente (comer cápsula y luego fantasma) para enseñarlo.
- Añadir estos nuevos datos al dataset, volviendo a la fase 2 y 3: combinar los CSV adicionales, reprocesar (o mejor, procesar incrementamente si el código lo permite).
- Re-entrenar la red con el dataset ampliado. Posiblemente iniciar desde cero o, mejor aún, **hacer *fine-tuning***: cargar el modelo actual y seguir entrenando algunas épocas con datos nuevos (bajando un poco la tasa de aprendizaje). Esto aprovecha lo aprendido y ajusta a las nuevas muestras. En PyTorch, solo cargar state_dict antes del loop de entrenamiento y continuar.

- Evaluar si la nueva versión mejora las puntuaciones.

- **Ajustes de la arquitectura o hiperparámetros**:

- Si se detectó sobreajuste (muy alta accuracy en entrenamiento pero mal en validación/juego), hacer el modelo más simple o regularizar:
    - Reducir número de filtros o capas en la red para disminuir parámetros.
    - Aumentar dropout (por ejemplo 0.5 en fc1).
    - Aumentar weight_decay en Adam.
- Si se detectó underfitting (el modelo ni siquiera aprende bien las acciones entrenadas; baja accuracy en train):
    - Aumentar capacidad del modelo: más filtros (ej. 128 en convs), más neuronas en fc.

    - Aumentar épocas o lr para entrenar más/faster.
    - Revisar si la representación de entrada es adecuada: quizás añadir alguna característica extra (por ejemplo, incorporar la dirección actual de Pac-Man si fuera relevante, aunque en Markov state no lo es tanto; o el estado de cada fantasma).
    - Ver si la embedding o no embedding afecta: podríamos probar la alternativa. Si usamos embedding y va mal, probar sin embedding con canales binarios one-hot (o viceversa).
- Modificar la **función objetivo** si conviene:
    - En lugar de entrenamiento puramente supervisado (imitar acciones del jugador), podríamos pensar en integrar señales de recompensa. Por ejemplo, ponderar acciones que condujeron a ganar con más peso. Pero esto ya entra en RL, quizás fuera de alcance del proyecto actual. Si se busca maximizar score, un enfoque de *reinforcement learning* sería ideal, pero dado que trabajamos con datos de partidas, mantenemos supervisado.
    - Sin cambiar a RL completo, podríamos filtrar las acciones de entrenamiento para enfatizar buenas decisiones. Por ejemplo, eliminar ejemplos donde el jugador perdió una vida (argumentando que tal vez no queremos que la red aprenda a replicar errores). Aunque esto reduce dataset, podría hacer al agente más conservador. Es delicado, pero es un posible ajuste: filtrar movimientos "fatales" (justo antes de morir) para que la red no los imite.
- **Re-entrenamiento**: Aplicar los cambios decididos y volver a entrenar el modelo. Registrar la nueva curva de pérdida y accuracy, esperando mejoras. Guardar la nueva versión del modelo (posiblemente con un nuevo nombre si se quiere comparar, o sobreescribir la anterior si es claramente mejor).

- Volver a probar en juego real con `NeuralAgent` para comprobar que el refinamiento surtió efecto. Idealmente, deberíamos ver la puntuación promedio subir o comportamientos indeseados corregirse. Continuar iterando hasta que los resultados sean satisfactorios dentro de las limitaciones de tiempo.

- **Ajuste de parámetros del agente**:

- Si `NeuralAgent` tiene parámetros configurables (como la velocidad de decaimiento de la exploración, o pesos en la combinación heurística vs red):
    - Por ejemplo, tal vez la heurística suma algo como `+500` por comer una cápsula. Si notamos que el agente a veces ignora comida lejana porque la red no la valoró, quizás el peso heurístico de distancia a comida se puede subir.
    - Estos parámetros pueden estar en `multiAgents.py`. Si no marcados #ahmed, podríamos tunearlos:
    - Bajar la tasa inicial de exploración si queremos que el agente confíe más en la red desde el inicio de las pruebas.
    - Ajustar la forma en que se mezcla la red: si `final_score = net_confidence + alpha*food_dist - beta*ghost_dist + gamma*score`, calibrar alphas, betas. Esto usualmente el instructor habrá fijado valores razonables, pero podemos experimentar offline modificándolos y viendo si mejora. Documentar tales ajustes si se realizan.
    - **Ejemplo**: aumentar la prioridad de supervivencia: incrementar la penalización por cercanía de fantasma (beta mayor) para ver si Pac-Man huye más agresivamente. O incrementar la influencia de la red (tal vez multiplicando sus output by some factor) si confiamos en el modelo.
- Estos cambios deben ser sutiles y comprobados, pues alterar heurísticas puede tener efectos secundarios (ej. si penalizamos fantasmas demasiado, Pac-Man podría perder oportunidades de comerlos cuando sería seguro).

- Dado el objetivo del proyecto es la red neuronal, se esperaría que la mayor parte del comportamiento venga de ésta, usando heurísticas solo como apoyo. Por tanto, no es crítico afinar heurísticas en exceso; solo asegurarse de no tener valores tan desbalanceados que anulen la red. (Si el agente siempre obedece heurística, entonces el trabajo de la red se ve opacado).

- **Documentar los cambios**: Para cada iteración probada, anotar:

- Qué se modificó (más datos, nuevos hiperparámetros, etc.).
- Resultado en validación/training metrics y en puntuación de juego.
- Seleccionar el modelo final basado en este análisis (por ejemplo: *"La versión 3 del modelo (con 5000 datos y dropout) obtuvo 95% accuracy train, 85% val, y promedió 1000 puntos en juego; decidimos que es adecuada para entregar"*).

Este ciclo de refinamiento puede repetirse hasta donde el tiempo lo permita o hasta alcanzar un rendimiento satisfactorio. Es importante equilibrar el esfuerzo: dado que es un proyecto académico, llegados a un punto decente, es mejor consolidar resultados que perseguir perfección absoluta.

8. Preparación del entregable final

Finalmente, una vez que el agente neuronal funciona correctamente y cumple con los objetivos, se procede a preparar todos los elementos necesarios para la entrega del proyecto. Esto incluye el código, el modelo entrenado y la documentación correspondiente. Las actividades a realizar son:

- **Organización del código fuente**: Asegurarse de que todos los archivos modificados o creados están en orden:
- `net.py`: Debe contener la implementación final de `PacmanNet` y el código de entrenamiento. Remover cualquier fragmento de prueba o debugging (por ejemplo, `print` temporales usados para chequear shapes). Comentar claramente las secciones implementadas para facilitar la corrección (sin borrar comentarios originales). Verificar que no se dejaron modificaciones accidentales en secciones marcadas `#ahmed`.
- `multiAgents.py`: Idealmente no modificado, salvo que se hayan ajustado parámetros heurísticos no protegidos. Si se hicieron, resaltarlos en comentarios para que el revisor los identifique fácilmente.
- `gamedata.py`: Probablemente sin cambios en lógica, a menos que se activó algo manual para recolección. Si se añadió algo como `# start recording` en pacman.py o gamedata.py, considerar si lo dejamos (si no interfiere) o revertirlo si no era solicitado específicamente (porque modificar pacman.py no se mencionó prohibido, pero tampoco estaba en objetivo).
- Cualquier *notebook* o script auxiliar usado para análisis (no necesariamente entregable, solo interno). No hace falta incluir si no se pide, pero podemos extraer de ahí fragmentos para el informe.
- **Verificación de la restricción**: Realizar una pasada final buscando `#ahmed` en todo el proyecto para confirmar que no alteramos nada justo después o en la misma línea. Si se encuentra algo cambiado inadvertidamente, corregirlo para cumplir la consigna.

- Ejecutar una prueba final completa: por ejemplo, borrar/renombrar el modelo guardado, ejecutar `python net.py` para reentrenar (o cargar si permitido), luego `python pacman.py -p NeuralAgent -n 1` para ver que todo fluye sin errores. Esto simula al evaluador corriendo nuestro código desde cero.

- **Guardar el modelo entrenado**: Incluir en la entrega el archivo de pesos resultante, ubicado en la carpeta `models/`. Por ejemplo, `models/pacman_net.pth` (~ unos pocos MB). Esto permite que el evaluador ejecute directamente el juego con el agente sin tener que reentrenar (a menos que se quiera demostrar el entrenamiento, pero generalmente se provee el modelo final).

- Si el archivo es muy grande (no debería serlo, quizás <1MB a unos pocos MB), está bien. Si, hipotéticamente, fuese enorme, se podría documentar cómo generarlo y quizás omitirlo, pero en nuestro caso se incluye.

- Asegurarse de mencionar en las instrucciones cómo usar el modelo (aunque es obvio con pacman.py -p NeuralAgent).

- **Informe o documentación**: Según requerimientos, se podría pedir un informe textual (quizá este mismo plan adaptado a memoria). Incluir:

- **Introducción y objetivo**: Explicar brevemente que se implementó un agente neuronal para Pac-Man, con qué propósito (maximizar puntuación) y enfoque (aprendizaje supervisado de partidas propias).
- **Descripción de la solución**: Resumir la arquitectura de la red, cómo se entrenó (datos usados, hiperparámetros) y cómo se integra con Pac-Man.
- **Resultados**: Presentar los hallazgos de desempeño: "El agente logra X puntuación promedio, comparado con Y de un agente aleatorio, etc. Muestra comportamientos inteligentes como Z." Mencionar limitaciones si las hay.
- **Instrucciones de uso**: Cómo ejecutar el juego con el NeuralAgent, cómo reentrenar si se desea. Por ejemplo, "Ejecutar `python net.py` entrenará la red con los datos en pacman_data/ y guardará el modelo. Luego, `python pacman.py -p NeuralAgent` usará ese modelo."
- **Estructura de archivos**: Listar brevemente qué se modificó/añadió (p.ej., "La clase PacmanNet en net.py, y se generó un modelo almacenado en models/pacman_net.pth.").
- **Cumplimiento de restricciones**: Afirmar que no se modificó código marcado con `#ahmed` y que se siguió la especificación.
- **Posibles mejoras futuras**: Siempre es bueno añadir un párrafo final con sugerencias: por ejemplo, entrenar con técnica de refuerzo para mejorar aún más, probar arquitecturas más profundas, o entrenar con más datos para generalizar a nuevos laberintos.

- Si este informe debe ser entregado, seguramente en formato PDF o markdown. Dado que la tarea pedía un "informe detallado con encabezados...", es probable que esta misma estructura sirva como base del documento entregable. Por ello, podríamos pulirlo en estilo y concisión, asegurándonos que esté en español correcto, bien organizado (cosa que hemos hecho).

- **Packaging y envío**:

- Si la entrega es un repositorio (por ejemplo, GitHub Classroom), hacer commit/push de todos los cambios y archivos nuevos (incluyendo el modelo si se permite).
- Si es un archivo comprimido, empaquetar la carpeta con el código y modelo. Verificar que la estructura de carpetas se mantiene (no ubicar archivos fuera de donde el programa los busca).
- Incluir un **README** si necesario, con instrucciones resumidas. Puede ser el mismo informe o un extracto.
- Revisar nuevamente la lista de requisitos del proyecto para asegurarse de no omitir nada (por ejemplo, a veces piden capturas de pantalla de una partida ganada, etc. Si se pide, preparar esas imágenes).

- **Entrega de datos de entrenamiento**: probablemente no sea requerido entregar los CSV de partidas (ya que pueden ser grandes y personales). A menos que se pida expresamente, podemos dejarlos fuera. El modelo entrenado ya representa ese conocimiento. Sin embargo, conservar los CSV por si se necesita re-entrenar o verificar durante la corrección.
- Finalmente, enviar dentro del plazo establecido.

Con todo lo anterior, habremos completado el proyecto satisfactoriamente. El agente Pac-Man con red neuronal estará implementado, entrenado y evaluado, y todos los artefactos necesarios listos para la revisión. Hemos seguido un plan metódico: desde analizar el código base, pasando por la generación de datos de entrenamiento, el diseño del modelo, su implementación cuidadosa respetando restricciones, el entrenamiento con verificación de rendimiento, hasta pruebas exhaustivas en el juego y ajustes finales. Este enfoque asegura un desarrollo ordenado y **maximiza las posibilidades de éxito** en el logro de un Pac-Man autónomo que juega de manera inteligente y maximiza la puntuación [19].

---

[1] [2] 4. Game Search — Técnicas y Algoritmos de Búsqueda IA
https://ahmedbegggaua.github.io/TAB2024/topic4.html

[3] [4] 11. Game Theory and Adversarial Search — Técnicas y Algoritmos de Búsqueda IA
https://ahmedbegggaua.github.io/TAB2024/minimax.html

[5] 3. Heuristic Search — Técnicas y Algoritmos de Búsqueda IA
https://ahmedbegggaua.github.io/TAB2024/topic3.html

[6] [7] [8] [9] [10] [11] [13] [19] 10. Pacman with AI — Técnicas y Algoritmos de Búsqueda IA
https://ahmedbegggaua.github.io/TAB2024/practice_3.html

[12] [17] [18] GitHub - AhmedBegggaUA/pacman
https://github.com/AhmedBegggaUA/pacman

[14] [15] [16] 科研！社工？卷绩人！ - 智能体大赛引导文档
https://agent-guide.net9.org/instruction/pacman/