UAB
Universitat Autònoma
de Barcelona

Facultad de ciencias

Grado en matemáticas

Aproximación de Estrategias en Juegos Bayesianos mediante Redes Neuronales

Trabajo de final de grado

Autor: Álvaro Martorell Ortuño

Tutor: Roberto Rubio

Junio de 2025

Índice

1. Introducción

En los últimos años, el uso de inteligencia artificial para resolver juegos estratégicos complejos ha generado avances notables. Un logro muy destacado ha sido el desarrollo del bot Pluribus, que logró vencer a los mejores jugadores profesionales de póker en partidas de *Texas Hold'em* para varios jugadores.

En este trabajo se explora la construcción de un modelo predictivo capaz de replicar la estrategia del bot Pluribus. Concretamente se diseña una red neuronal, utilizando *Python*, que al observar un estado de la partida sea capaz de predecir la acción más probable que el bot habría elegido en esa situación. El modelo será entrenado sobre datos reales extraídos de logs de partidas.

Para abordar este problema, primero definiremos de manera consistente los juegos bayesianos, también conocidos como juegos de información incompleta, (véase la sección 2.2) y revisaremos los conceptos fundamentales del Teorema de Aproximación Universal y su demostración (véase 4.2). Estos fundamentos teóricos nos proporcionan la justificación necesaria para aplicar modelos de aprendizaje profundo en este contexto.

Un juego se modela matemáticamente como una interacción estratégica entre varios agentes. En un juego de información completa, todos los jugadores comparten conocimiento exhaustivo de reglas, estrategias posibles y pagos posibles. Este escenario, si bien es útil para establecer definiciones formales, no refleja muchos entornos reales que implican incertidumbre sobre el estado del juego o las intenciones de los oponentes. En un juego bayesiano, cada jugador posee su información privada, el propio "tipo", por ejemplo, en una mano de póker las dos cartas que se les reparten a cada uno, además de información que es conocida por todos, como el tamaño del bote, las cartas comunitarias, etc. Y se toman decisiones basadas en sus creencias probabilísticas de la información oculta de los oponentes. Este procedimiento de asignar y actualizar creencias frente a la incertidumbre es precisamente lo que hace del póker un caso de estudio paradigmático.

Para ilustrar estos conceptos, tomaremos como ejemplo principal el póker Texas Hold'em. Aquí, la estrategia de un jugador resulta de combinar sus dos cartas y el estado de juego, que es la información que todos tienen de la partida: las cartas comunitarias reveladas en cada ronda (*flop, turn* y *river*), el tamaño del bote, el dinero de todos los jugadores, el historial de apuestas (acciones previas de todos los participantes), etc.

El bot Pluribus ha demostrado que, con una representación adecuada del estado y un entrenamiento sobre una gran base de datos de partidas, es posible aproximar eficazmente la función de decisión óptima en este tipo de juegos bayesianos. En este trabajo se formaliza dicha función como una aplicación continua sobre un espacio de información codificada, y se aproxima mediante una red neuronal, en línea con el Teorema de Aproximación Universal.

En las secciones siguientes, se detalla esta construcción paso a paso: desde la formulación teórica hasta el diseño y entrenamiento del modelo, pasando por la codificación de datos y la evaluación de los resultados. El objetivo final es mostrar cómo herramientas matemáticas y computacionales pueden integrarse para emular decisiones complejas en juegos estratégicos reales.

2. Formalización de las definiciones

A continuación se presentan las fórmulas clave, acompañadas de su interpretación en lenguaje natural y de ejemplos que ilustran cada caso.

2.1. Juego de Información Completa

Definición 2.1. Un **juego de información completa** se define como la tupla

$$G = (I, A, u) \tag{1}$$

donde:

- I es el conjunto de jugadores.
- $A = A_1 \times A_2 \times \cdots \times A_n$, donde A_i es el conjunto de acciones o estrategias disponibles para cada jugador, intuitivamente, A recoge todas las combinaciones posibles de acciones.
- $u = (u_1, u_2, \ldots, u_n)$ son las funciones de pago, donde $u_i : A \to \mathbb{R}$ para cada jugador $i \in I$, es decir, u mide cuánto gana o pierde cada jugador según la estrategia elegida.

En este tipo de juego, se asume que todos los jugadores tienen conocimiento completo sobre la estructura del juego, incluyendo estrategias, pagos y posibles decisiones de los demás jugadores.

Una vez definidos los elementos que conforman un juego de información completa, resulta fundamental analizar cuáles son las posibles soluciones que pueden surgir de la interacción estratégica entre los jugadores. En este contexto, uno de los conceptos centrales de la teoría de juegos es el equilibrio de Nash, que formaliza la idea de estabilidad estratégica: una situación en la que ningún jugador tiene incentivos para modificar su estrategia de forma unilateral, dado el comportamiento de los demás.

Definición 2.2. Sea $G = (I, A, u)$, un juego de información completa, decimos que un perfil de estrategias $s^* = (s_1^*, ..., s_n^*) \in A$ es un **equilibrio de Nash** si $\forall i \in I$ se cumple:

$$u_i(s_i^*, s_{-i}^*) \geq u_i(s_i, s_{-i}^*),\ \forall s_i \in A_i$$

Es decir, ningún jugador puede mejorar su utilidad desviándose unilateralmente de su estrategia óptima.

Como los jugadores conocen completamente las funciones de pago y los conjuntos de estrategias de todos los participantes, lo que permite la existencia de razonamientos estratégicos de este tipo.

Ejemplo: El dilema del prisionero

El dilema del prisionero es un juego clásico en teoría de juegos que cumple con la definición de juego de información completa.

- **Conjunto de jugadores** I:

  $$I = \{1, 2\}$$

  donde los jugadores representan dos prisioneros (Prisionero 1 y Prisionero 2).

- **Conjunto de estrategias** A: Cada jugador tiene dos acciones posibles:

$$A_1 = A_2 = \{\text{Cooperar (C)}, \text{Traicionar (T)}\}$$

Por lo tanto, el espacio de estrategias conjunto es:

$$A = A_1 \times A_2 = \{(C, C), (C, T), (T, C), (T, T)\}$$

- **Funciones de pago** $u = (u_1, u_2)$: Las funciones de pago dependen de la estrategia elegida por ambos jugadores:

  - Si ambos cooperan (C, C), reciben una pena reducida de 1 año cada uno.
  - Si uno traiciona y el otro coopera (T, C) o (C, T), el que traiciona queda libre (0 años) y el otro recibe una condena de 3 años.
  - Si ambos traicionan (T, T), reciben 2 años cada uno.

Las funciones de pago entonces son:

$$u_1(C, C) = -1, \quad u_1(C, T) = -3, \quad u_1(T, C) = 0, \quad u_1(T, T) = -2$$

$$u_2(C, C) = -1, \quad u_2(C, T) = 0, \quad u_2(T, C) = -3, \quad u_2(T, T) = -2$$

Finalmente, obtenemos que los pagos están representados en la siguiente tabla:

		Prisionero 2	
	$\mathbf{s}_i$	**C**	**T**
Prisionero 1	**C**	$(-1, -1)$	$(-3, 0)$
	T	$(0, -3)$	$(-2, -2)$

Tabla 1: Matriz de pagos del Dilema del Prisionero

Observamos que ambos jugadores conocen todas las estrategias disponibles, las recompensas para cada combinación de estrategias, y saben que el otro jugador también conoce todo esto. Este ejemplo ilustra un juego de información completa, ya que todas las partes del juego son conocidas y compartidas por ambos jugadores.

En este juego, la estrategia dominante para ambos jugadores es traicionar, ya que esta elección minimiza su condena independientemente de la acción del otro. Por tanto, el perfil *(T,T)* constituye un equilibrio de Nash: una vez que ambos jugadores eligen traicionar, ninguno de ellos puede mejorar su situación cambiando su estrategia de forma unilateral. Aunque el resultado *(C,C)* sería preferible para ambos en términos absolutos (solo un año de prisión), no es un equilibrio, ya que cada jugador tendría incentivos a desviarse si supiera que el otro coopera. Esto ilustra cómo la racionalidad individual puede conducir a un resultado colectivamente ineficiente, característica típica de este tipo de dilemas.

2.2. Juego Bayesiano

El concepto moderno de juego de información incompleta (o juego *bayesiano*) fue introducido por Harsanyi [1] en sus trabajos clásicos de 1967–68. Formalmente se representa como:

Definición 2.3. Un **juego de información incompleta** se define como la tupla

$$G = (I, \Theta, \Theta^0, A, p, u) \tag{2}$$

donde:

- $I = \{1, ..., n\}$ es el conjunto de n jugadores.
- $\Theta = \Theta_1 \times \cdots \times \Theta_n$ es el espacio de los tipos, donde cada $\theta_i \in \Theta_i$ para $i \in I$ codifica la información privada de cada jugador
- Θ^0 es el conjunto de posibles realizaciones de la información común, y $\theta^0 \in \Theta^0$ representa la información que todos tienen en común.
- $A = A_1 \times A_2 \times \cdots \times A_n$ es el conjunto de estrategias, siendo cada A_i el conjunto de acciones disponibles para cada jugador.
- $p(\theta^0, \theta_1, \ldots, \theta_n)$ es la distribución de probabilidad sobre los tipos de los jugadores, que modela la incertidumbre en cada estado del juego. Para un espacio de los tipos finito asumimos que $p(\theta_i) > 0$ para todo $\theta_i \in \Theta_i$.
- $u = (u_1, u_2, \ldots, u_n)$ son las funciones de pago, dependientes de los tipos y las acciones posibles:
$$u_i : A \times \Theta \times \Theta^0 \to \mathbb{R}$$

Esta representación sigue la notación empleada por Jonathan Levin [2].
En un juego de información incompleta, los jugadores conocen su propio tipo θ_i pero no los tipos de los demás jugadores. La incertidumbre sobre los tipos se modela mediante la distribución de probabilidad común p, la cual refleja su creencia inicial y puede actualizarse según las acciones observadas.

A diferencia del equilibrio de Nash clásico, que se aplica a juegos con información completa, el equilibrio bayesiano surge como una extensión natural para juegos de información incompleta. En este contexto, cada jugador forma expectativas sobre las características desconocidas a través de creencias probabilísticas y elige una estrategia que maximiza su utilidad esperada. Este concepto incorpora tanto la racionalidad como la consistencia entre estrategias y creencias, lo que lo convierte en el análogo lógico del equilibrio de Nash en situaciones de incertidumbre estructurada. Formalmente definimos:

Definición 2.4. Sea $G = (I, \Theta, \Theta^0, A, p, u)$ un juego bayesiano, una **estrategia bayesiana pura** del jugador i es una función:

$$f_i : \Theta^0 \times \Theta_i \to A_i$$

que asigna una acción a cada tipo posible del jugador i y la información común. Es decir, cada tipo de jugador actúa como si fuera un jugador distinto, seleccionando su acción en función de su información privada (su tipo) y la que comparten todos los jugadores.

Denotamos por $\mathcal{S}^{\Theta^0 \times \Theta_i}$ el conjunto de estrategias bayesianas puras del jugador i.

Definición 2.5. Sean $G = (I, \Theta, \Theta^0, p, u)$ un juego bayesiano y $f_i \in \mathcal{S}^{\Theta^0 \times \Theta_i}$ una estrategia bayesiana pura del jugador i. Un perfil de estrategias bayesianas puras $(f_1, \ldots, f_n)$ es un **equilibrio de Nash bayesiano puro** si, para todo jugador i, todo $\theta^0 \in \Theta^0$, y todo tipo $\theta_i \in \Theta_i$, se cumple que:

$$\sum_{\theta_{-i} \in \Theta_{-i}} u_i\left(f_i(\theta^0, \theta_i), f_{-i}(\theta^0, \theta_{-i}), \theta^0, \theta_i, \theta_{-i}\right) \cdot p(\theta_{-i} \mid \theta^0, \theta_i) \tag{3}$$

$$\geq \sum_{\theta_{-i} \in \Theta_{-i}} u_i\left(a_i, f_{-i}(\theta^0, \theta_{-i}), \theta^0, \theta_i, \theta_{-i}\right) \cdot p(\theta_{-i} \mid \theta^0, \theta_i) \tag{4}$$

para todo $a_i \in A_i$

En este tipo de juegos las estrategias no tienen por qué ser fijas dados θ_i y θ^0, puesto que las ganancias pueden depender también de los tipos del resto de jugadores, así que para tener en cuenta las creencias probabilísticas de los tipos del resto, definimos las estrategias mixtas:

Definición 2.6. Sea $G = (I, \Theta, \Theta^0, A, p, u)$ una **estrategia bayesiana mixta** para el jugador i se define como una función

$$\sigma_i : \Theta^0 \times \Theta \to \Delta(A_i)$$

donde $\Delta(A_i)$ es el conjunto de todas las distribuciones de probabilidad sobre A_i, es decir, el conjunto de estrategias mixtas.

Denotamos por $\mathcal{M}^{\Theta^0 \times \Theta_i}$ el conjunto de estrategias bayesianas mixtas del jugador i.

Definición 2.7. Sea $G = (I, \Theta, \Theta^0, A, p, u)$ un juego bayesiano, y sea $\sigma_i \in \mathcal{M}^{\Theta^0 \times \Theta_i}$ una estrategia bayesiana mixta del jugador i. Un perfil de estrategias bayesianas mixtas $(\sigma_1, \ldots, \sigma_n)$ es un **equilibrio de Nash bayesiano mixto** si, para todo jugador i, toda información común $\theta^0 \in \Theta^0$ y todo tipo $\theta_i \in \Theta_i$, se cumple que:

$$\sum_{\theta_{-i} \in \Theta_{-i}} \mathbb{E}_{a \sim \sigma(\theta^0, \theta)}\left[u_i(a, \theta^0, \theta_i, \theta_{-i})\right] \cdot p(\theta_{-i} \mid \theta^0, \theta_i)$$
$$\geq \sum_{\theta_{-i} \in \Theta_{-i}} \mathbb{E}_{a \sim (\sigma_i', \sigma_{-i})(\theta^0, \theta)}\left[u_i(a, \theta^0, \theta_i, \theta_{-i})\right] \cdot p(\theta_{-i} \mid \theta^0, \theta_i)$$

para toda desviación alternativa $\sigma_i' : \Theta^0 \times \Theta \to \Delta(A_i)$.

Observación sobre la notación: En la expresión

$$\mathbb{E}_{a \sim \sigma(\theta^0, \theta)}\left[u_i(a, \theta^0, \theta_i, \theta_{-i})\right],$$

la notación $a \sim \sigma(\theta^0, \theta)$ indica que el perfil de acciones $a = (a_1, \ldots, a_n)$ es una variable aleatoria cuya distribución está inducida por el perfil de estrategias mixtas $\sigma = (\sigma_1, \ldots, \sigma_n)$, condicionado a los tipos $\theta = (\theta_i, \theta_{-i})$ y a la información común θ^0.

Cada jugador j selecciona su acción aleatoriamente según la distribución $\sigma_j(\theta^0, \theta_j)$, de modo que el perfil conjunto de acciones sigue una distribución sobre el espacio $A = A_1 \times \cdots \times A_n$.

La esperanza $\mathbb{E}[u_i(\cdot)]$ representa entonces la *ganancia esperada* del jugador i, considerando tanto la aleatoriedad debida a las estrategias mixtas como la incertidumbre sobre los tipos de los demás jugadores, modelada mediante la distribución condicional $p(\theta_{-i} \mid \theta^0, \theta_i)$.

Ejemplo: Subasta a ciegas

Un ejemplo clásico de juego bayesiano, ampliamente analizado por Zamir [3], es la subasta a ciegas de primer precio, donde dos jugadores presentan una oferta sin conocer la valoración del otro jugador.

- **Conjunto de jugadores** I:

  $$I = \{1, 2\}$$

  representando a los dos participantes en la subasta.

- **Conjunto de tipos** $\Theta = \Theta_1 \times \Theta_2$: Cada jugador i tiene una valoración privada del objeto subastado $\theta_i \in \Theta_i$, dada por $\Theta_1, \Theta_2 = [0, 1] \subset \mathbb{R}$ pero no conoce la del otro jugador.

- En este caso, los jugadores no tienen ningun tipo de información común, por lo que consideramos $\Theta^0 = \{0\}$.

- **Conjunto de estrategias** $A = A_1 \times A_2$: Cada jugador hace una oferta $a_i \in A_i = [0, 1] \subset \mathbb{R}$ basada en la valaroción propia θ_i cumpliendo $a_i < \theta_i$.

- **Distribución de probabilidad** p: Las valoraciones son independientes, por lo que la distribución conjunta de tipos es

  $$p(\theta_1, \theta_2) = 1 \quad \text{para } (\theta_1, \theta_2) \in [0, 1]^2.$$

- **Funciones de pago** u: Gana quien realice la oferta mayor: si $a_i > a_j$ el jugador i se lo lleva.

  $$u_i = \begin{cases} \theta_i - a_i & \text{si } a_i > a_j, \\ \frac{1}{2}(\theta_i - a_i) & \text{si } a_i = a_j, \\ 0 & \text{si } a_i < a_j, \end{cases}$$

  Aquí, cada jugador elige su oferta a_i con conocimiento solo de su valoración privada θ_i, y debe inferir la posible oferta del oponente mediante la distribución p.

En este ejemplo, está demostrado que el único equilibrio de Nash bayesiano es el perfil de estrategias

$$f_i(\theta_i) = \frac{\theta_i}{2}, \text{ para } i = 1, 2.$$

Este resultado ha sido demostrado formalmente por Jonathan Levin [2], y refleja que en equilibrio, cada jugador ofrece una puja igual a la mitad de su valoración privada, equilibrando así el dilema entre incrementar sus opciones de ganar y mantener bajo el precio que pagará si resulta adjudicatario.

3. Ejemplo *Poker Texas Hold'em*

Siguiendo la estructura general de juegos bayesianos desarrollada por autores como Harsanyi y Zamir, podemos representar el póker Texas Hold'em como un juego con información incompleta. Utilizando la definición de juego bayesiano tenemos la siguiente tupla,

$$G = (I, \Theta, \Theta^0, A, p, u)$$

donde se interpreta cada uno de los elementos de la siguiente manera:

Conjunto de jugadores, I

Para simplificar, consideraremos un juego entre dos jugadores:

$$I = \{1, 2\}.$$

Conjunto de tipos, Θ

El tipo de cada jugador consiste en su mano privada. Sea $\mathcal{D}$ la baraja de 52 cartas. Entonces, para cada jugador i, el conjunto de tipos es:

$$\Theta_i = \{\theta_i \subset \mathcal{D} \mid |\theta_i| = 2\}.$$

La asignación conjunta de tipos para ambos jugadores es:

$$\Theta = \Theta_1 \times \Theta_2.$$

Información común, Θ^0

En este juego, la información que poseen todos los jugadores son las cinco cartas comunitarias que se revelan progresivamente durante las distintas rondas de juego (tres en la primera, una en la segunda y una en la última), el tamaño del bote, la posición relativa en la mesa, los montos de fichas de cada jugador y el historial de apuestas realizadas hasta el momento.

Acciones, A

El espacio total de las estrategias posibles, como hemos definido anteriormente, es el producto cartesiano $A = A_1 \times A_2$. El espacio de acciones posibles para cada jugador i está dado por

$$A_i = \{\textit{fold, check, call, bet, raise}\}$$

A continuación se describe brevemente el significado de cada acción en el contexto del juego del póker:

- *fold*: El jugador se retira de la mano, renunciando a cualquier posibilidad de ganar el bote. Esta acción es irreversible para el resto de las rondas.
- *check*: El jugador decide no apostar, pero sin retirarse, pasando la acción al siguiente jugador. Esta opción solo está disponible si no se ha realizado ninguna apuesta previa en la ronda.

- *call*: El jugador iguala la apuesta actual para permanecer en la mano. Requiere pagar la misma cantidad que la apuesta más alta realizada hasta el momento.
- *bet*: El jugador realiza una apuesta inicial si todavía no se ha apostado en la ronda actual. Esta acción establece una cantidad que los demás deben al menos igualar para continuar.
- *raise*: El jugador aumenta la cantidad de una apuesta previamente realizada. Obliga a los demás jugadores a igualar este nuevo monto si desean continuar en la mano.

Distribución de probabilidad, p

La función de probabilidad p asigna una probabilidad conjunta a cada posible asignación de tipos (reparto de cartas) para los jugadores. Dado el reparto sin reemplazo, la distribución es uniforme sobre todas las asignaciones válidas. Por ejemplo, si ignoramos el orden y consideramos solo las manos:

$$p(\theta_1, \theta_2) = \frac{1}{\binom{52}{2}\binom{50}{2}},$$

donde $\binom{52}{2}$ es el número de manos posibles para el primer jugador y $\binom{50}{2}$ para el segundo, al descartar las dos cartas repartidas al jugador 1.

Función de pago, u

La función de pago u_i para cada jugador i depende del resultado final del juego, determinado por:

- La mano privada del jugador, θ_i.
- Las cartas comunitarias reveladas, denotadas por $C \subset \theta^0$.
- Las decisiones y el resultado de las rondas de apuestas.

Si solo queda un jugador activo, digamos que es el jugador k, la partida termina y se le asigna el bote automáticamente:

$$u_k = B, \quad \text{y } u_i = 0 \text{ para } i \neq k.$$

En el caso que lleguen los dos jugadores al final de la partida, sea $R(\theta_i, C)$ el ranking o clasificación de la mejor mano posible de 5 cartas que forma el jugador i usando sus cartas y las cartas comunitarias. Entonces, el ganador es el jugador con la mejor mano, y el bote acumulado se asigna en consecuencia. Una forma simplificada de definir la función de pago es:

$$u_i(\theta, C) = \begin{cases} B, & \text{si } R(\theta_i, C) > R(\theta_j, C), \\ B/2, & \text{si } R(\theta_i, C) = R(\theta_j, C) \\ 0, & \text{si } R(\theta_i, C) < R(\theta_j, C), \end{cases}$$

donde:

- $\theta = (\theta_1, \theta_2)$ es el perfil de tipos de los jugadores.
- B es el bote total acumulado tras las rondas de apuestas.

Estrategia o funcion de decisión óptima, σ_i

Cada jugador i elige una acción en función de la información que posee, es decir, su tipo, la información pública que se revela progresivamente y las creencias probabilísticas sobre los tipos de los demás jugadores. Los jugadores tienen la opción de usar o bien estrategias bayesianas puras $f_i : \Theta^0 \times \Theta_i \to A_i$ o bien mixtas, $\sigma_i : \Theta^0 \times \Theta_i \to \Delta A_i$.

3.1. Ejemplo: Partida de *Texas Hold'em*

Consideremos una partida simplificada entre dos jugadores.

Sean $G = (I, \Theta, \Theta^0, A, p, u)$, donde $I = \{1, 2\}$ y $\mathcal{D}$ la baraja de 52 cartas. Los dos jugadores empiezan con un total de 100 fichas. El jugador uno pone en el bote 1 ficha y el jugador dos pone 2 fichas por ser la ciega pequeña y la grande respectivamente, (en las partidas de *Texas Hold'em* para varios jugadores, los situados en las 2 primeras posiciones aportan una cantidad de fichas pactada según la mesa antes de ver las cartas privadas). Supongamos que los dos jugadores eligen una estretegia bayesiana pura

$$f_i : \Theta^0 \times \Theta \to A_i, \text{ para } i = 1, 2.$$

Empieza la partida:

- **Ronda 1, *Preflop*:** Se reparten a cada uno sus dos cartas privadas:

  $$\theta_1 = \{A\spadesuit, K\heartsuit\}, \quad \theta_2 = \{Q\clubsuit, J\clubsuit\}.$$

  El jugador uno decidice decide aumentar la apuesta a 10 fichas, es decir $f_1(\theta^0, \theta_1) =$ *"raise"*, ahora el bote total es de 12 fichas. Después de la apuesta del rival el jugador dos responde con pagar la apuesta, $f_2(\theta^0, \theta_2) =$ *"call"* y pasar a la siguiente ronda con un bote total de 20 fichas.

- **Ronda 2, *Flop*:** Se revelan tres cartas comunitarias:

  $$\{T\diamondsuit, 9\spadesuit, 3\heartsuit\}.$$

  El jugador uno decice apostar 10 fichas más, $f_1(\theta^0, \theta_1) =$ *"bet"*. El jugador 2 vuelve a pagar la apuesta $f_2(\theta^0, \theta_2) =$ *"call"*y pasamos a la tercera ronda con un bote de 40 fichas.

- **Ronda 3, *Turn*:** Se revela la cuarta carta:

  $$\{7\clubsuit\}.$$

  El jugador uno decide no apostar, $f_1(\theta^0, \theta_1) =$ *"check"* y el jugador dos igual$f_2(\theta^0, \theta_2) =$ *"check"*. Pasamos a la última ronda con el mismo tamaño de bote que antes.

- **Ronda 4, *River*:** Se revela la última carta:

  $$\{8\spadesuit\}.$$

  De modo que:

  $$C = \{T\diamondsuit, 9\spadesuit, 3\heartsuit, 7\clubsuit, 8\spadesuit\} \subset \theta^0.$$

  El jugador uno decide no apostar, $f_1(\theta^0, \theta_1) =$ *"check"*, el jugador dos decide apostar 20 fichas, $f_2(\theta^0, \theta_2) =$ *"bet"* finalmente el jugador uno decide retirarse de la mano, $f_1(\theta^0, \theta_1) =$ *"fold"*.

- **Asignación del bote:** Puesto que el jugador dos es el único jugador activo en la partida se le asigna el bote automáticamente.

Resumen: Este ejemplo ilustra una partida de Texas Hold'em en el marco de un juego bayesiano. Cada jugador posee información privada (sus cartas iniciales) y enfrenta diversas rondas de apuestas (preflop, flop, turn y river), en las cuales la estrategia se ajusta en función del historial de apuestas y de la información pública (cartas comunitarias). La función de pago final depende del ranking de la mejor mano formada y del bote acumulado durante el juego o de que solo quede un jugador activo en la mesa, como en este ejemplo.

4. Redes Neuronales

En esta sección abordamos el uso de redes neuronales como herramienta para aproximar funciones de decisión óptimas en juegos bayesianos con información incompleta, como ocurre en el póker. En particular, nos centramos en replicar, mediante aprendizaje profundo, la estrategia del bot *Pluribus*, cuya toma de decisiones puede modelarse como una función que depende del tipo privado del jugador, de la información común disponible en la partida y de sus creencias sobre los demás jugadores.

El uso de redes neuronales en este contexto se justifica formalmente por el *Teorema de Aproximación Universal*, el cual garantiza que, bajo ciertas condiciones, una red neuronal puede aproximar con precisión arbitraria cualquier función continua definida sobre un conjunto compacto. Dado que la estrategia óptima de un jugador racional en un juego bayesiano puede expresarse como una función continua de sus entradas, el aprendizaje profundo ofrece una vía teórica y práctica para aprender dicha función a partir de datos simulados o históricos.

A lo largo de esta sección presentaremos los fundamentos teóricos que respaldan este enfoque, así como su implementación concreta utilizando modelos de tipo perceptrón multicapa (*MLP*) con la librería `scikit-learn`.

4.1. Teorema de Aproximación Universal

Uno de los pilares teóricos que justifica el uso de redes neuronales en tareas de modelado de funciones complejas es el *Teorema de Aproximación Universal*. Este resultado establece que, bajo ciertas condiciones, una red neuronal con una sola capa oculta y una función de activación no lineal es capaz de aproximar con precisión arbitraria cualquier función continua definida sobre un conjunto compacto [5].

Este teorema justifica el uso de redes neuronales en tareas de regresión o clasificación en contextos arbitrariamente complejos, como las que surgen en juegos bayesianos con información incompleta. En el contexto de este trabajo, dicho teorema proporciona la base matemática que permite considerar la estrategia del bot *Pluribus*, como una función de decisión continua, que puede ser aprendida mediante una red neuronal adecuadamente entrenada.

A continuación presentamos una primera formulación accesible del teorema, adecuada para fines aplicados, mientras que en la siguiente sección se ofrece una versión formal y su demostración rigurosa.

Teorema 4.1 (Teorema de Aproximación Universal)**.** *Sea $K \subset \mathbb{R}^n$ un compacto y $C(K)$ el conjunto de funciones continuas definidas sobre K, entonces para cualquier $\varepsilon > 0$ y toda función $f \in C(K)$, existe una red neuronal de una sola capa oculta $N : \mathbb{R}^n \to \mathbb{R}$ que aproxima la función f de manera que*

$$|N(x) - f(x)| < \varepsilon, \quad \text{para cualquier } x \in K.$$

Comentarios: De este teorema se deduce que, en principio, una red con una sola capa oculta bastaría para aproximar cualquier función. Sin embargo, no nos aparta el número de neuronas necesarias, por lo que en la práctica utilizaremos arquitecturas multicapa (MLP) para capturar mejor la complejidad de la estrategia del bot.

4.2. Demostración del Teorema de Aproximación Universal

Presentamos a continuación una formulación rigurosa del Teorema de Aproximación Universal, y su demostración basada en el trabajo de Cybenko (1989) [5]. Para ello, empezamos recordando unas definiciones y resultados clave:

Definición 4.2 (Función sigmoide)**.** Diremos que una función $\sigma : \mathbb{R} \to \mathbb{R}$ es **sigmoide** si es continua y satisface

$$\lim_{x \to -\infty} \sigma(x) = 0 \quad \text{y} \quad \lim_{x \to +\infty} \sigma(x) = 1.$$

Definición 4.3 (Conjunto denso)**.** Sea $(X, \|\cdot\|)$ un espacio normado. Un subconjunto $A \subset X$ se dice **denso en** X si, para todo $f \in X$ y todo $\varepsilon > 0$, existe $g \in A$ tal que

$$\|f - g\| < \varepsilon.$$

Es decir, cualquier elemento de X puede ser aproximado arbitrariamente bien por elementos de A.

Definición 4.4 (Álgebra de funciones)**.** Sea X un conjunto no vacío cualquiera y denotemos por $C(X)$ el espacio de las funciones reales continuas sobre X. Un subconjunto $F \subseteq C(X)$ se llama **álgebra (real) de funciones** si verifica:

1. Es cerrado bajo suma, es decir, si $f, g \in F$ entonces $f + g \in F$.
2. Es cerrado bajo el producto por escalares, es decir, si $f \in F$ y $\lambda \in \mathbb{R}$ entonces $\lambda f \in F$.
3. Es cerrado bajo producto puntual, es decir,si $f, g \in F$ entonces $f \cdot g \in F$, donde $(f \cdot g)(x) := f(x)\, g(x)$.

Si además A contiene la función constante 1, decimos que A es un *álgebra con unidad*.

Teorema 4.5 (Stone–Weierstrass)**.** *Sea $K \subset \mathbb{R}^n$ un conjunto compacto y $C(K)$ el conjunto de funciones continuas reales definidas sobre K. Si la familia $\mathcal{F} \subseteq C(K)$ es un álgebra de funciones continuas sobre K tal que*

- *$\mathcal{F}$ separa puntos: para todo par $x, y \in K$, con $x \neq y$, existe $f \in \mathcal{F}$ tal que $f(x) \neq f(y)$,*
- *$\mathcal{F}$ no se anula: existe $f \in \mathcal{F}$ tal que $f(x) \neq 0$ para al menos un $x \in K$,*

entonces $\mathcal{F}$ es densa en $C(K)$ con la norma del supremo. Es decir, para todo $f \in C(K)$ y todo $\varepsilon > 0$, existe $g \in \mathcal{A}$ tal que

$$\|f - g\|_\infty < \varepsilon.$$

La demostración de este teorema se puede ver en [7]. Ahora enunciamos la versión formal del teorema:

Teorema 4.6 (Teorema de Aproximación Universal, Cybenko 1989)**.** *Sean $K \subset \mathbb{R}^n$ un compacto, $C(K)$ el conjunto de funciones continuas sobre K con la norma del supremo y $\sigma : \mathbb{R} \to \mathbb{R}$ una función sigmoide. Entonces, para toda $f \in C(K)$ y todo $\varepsilon > 0$ existe una red neuronal de una sola capa oculta*

$$N(x) \;=\; \sum_{j=1}^{N} a_j \, \sigma\big(w_j^{\mathsf{T}} x + b_j\big), \qquad x \in K, N \in \mathbb{N}, a_j \in \mathbb{R}, w_j \in \mathbb{R}^n, b_j \in \mathbb{R}$$

tal que $\|N - f\|_\infty < \varepsilon$. Equivalentemente, el conjunto

$$\mathcal{S} \;=\; \Big\{ x \mapsto \sum_{j=1}^{N} a_j \, \sigma(w_j^{\mathsf{T}} x + b_j) \;:\; N \in \mathbb{N}, a_j \in \mathbb{R}, w_j \in \mathbb{R}^n, b_j \in \mathbb{R} \Big\}$$

es denso en $C(K)$ con la norma del supremo.

Este teorema afirma que una red neuronal con una sola capa oculta y una función de activación adecuada, como la sigmoide $\sigma(x) = \frac{1}{1+e^{-x}}$, puede aproximar con precisión arbitraria cualquier función continua definida en un conjunto compacto, por ejemplo, el cubo $[0,1]^n$.

La clave del resultado es entender que las funciones de la forma

$$x \mapsto \sigma(w^T x + \theta)$$

pueden considerarse como "bloques básicos" que, al combinarse linealmente, permiten construir funciones cada vez más cercanas a una función objetivo f. La función $\sigma(w^T x + \theta)$ actúa como un "filtro suave"que se activa en ciertas regiones del dominio. Cambiando los parámetros w y θ, estas funciones pueden cubrir todo el espacio $[0,1]^n$.

Al sumar un número suficientemente grande de estas funciones con pesos adecuados α, se puede aproximar cualquier función continua, lo que justifica el uso de redes neuronales para aprender funciones complejas.

Demostración. Sea $K \subset \mathbb{R}^n$ compacto y sea

$$\mathcal{S} \;=\; \Big\{ x \;\longmapsto\; \sum_{j=1}^{N} a_j \, \sigma\big(w_j^{\mathsf{T}} x + b_j\big) : \; N \in \mathbb{N},\; a_j \in \mathbb{R},\; w_j \in \mathbb{R}^n,\; b_j \in \mathbb{R} \Big\}.$$

Demostraremos que $\mathcal{S}$ es denso en $C(K)$ aplicando directamente el Teorema de Stone–Weierstrass; para ello basta verificar que se cumplen todas las condiciones e hipótesis del teorema.

Veamos que $\mathcal{S}$ es un álgebra de funciones de $C(K)$:

- *Cerrada bajo suma y producto por escalares:* la definición ya es una combinación lineal finita, por lo que $f, g \in \mathcal{S}$ y $\lambda \in \mathbb{R}$ implican $f + g, \lambda f \in \mathcal{S}$.
- *Cerrada bajo producto punto a punto:* si $f(x) = \sum_j a_j \sigma(w_j^{\mathsf{T}} x + b_j)$ y $g(x) = \sum_k c_k \sigma(v_k^{\mathsf{T}} x + d_k)$, entonces

$$f(x)\, g(x) \;=\; \sum_{j,k} a_j c_k \, \sigma(w_j^{\mathsf{T}} x + b_j)\, \sigma(v_k^{\mathsf{T}} x + d_k) \;\in\; \mathcal{S},$$

ya que el producto de dos sigmoides es una función continua y puede re-expresarse como combinación finita de sigmoides (por ejemplo, usando identidades polinómicas o añadiendo nuevas neuronas con pesos apropiados).

Hemos visto que $\mathcal{S}$ es un álgebra de funciones de $C(K)$, veamos ahora que $\mathcal{S}$ separa puntos de K:

Sean $x, y \in K$ con $x \neq y$. Existe $w \in \mathbb{R}^n$ tal que $w^\mathsf{T} x \neq w^\mathsf{T} y$ (escójase, por ejemplo, $w = x - y$). Puesto que σ es estrictamente monótona, podemos elegir $b \in \mathbb{R}$ de manera que $\sigma(w^\mathsf{T} x + b) \neq \sigma(w^\mathsf{T} y + b)$. Así, la neurona $x \mapsto \sigma(w^\mathsf{T} x + b) \in \mathcal{S}$ distingue x de y.

Finalmente comprobamos que $\mathcal{S}$ no se anula:

Sea $x_0 \in K$ fijo. Como la función sigmoide$\sigma : \mathbb{R} \to \mathbb{R}$ no puede ser la función nula por definición, podemos escoger $b \in \mathbb{R}$ tal que

$$\sigma(b) \neq 0$$

. Tomamos ahora $w = 0 \in \mathbb{R}^n$. Para culaquier $x \in K$ se tiene que

$$\sigma(w^T x + b) = \sigma(b) \neq 0$$

y en particular

$$\sigma(w^T x_0 + b) = \sigma(b) \neq 0$$

Por lo tanto la neurona

$$x \longmapsto \sigma(w^T x + b) \in \mathcal{S}$$

toma un valor no nulo en $x_0 \in K$, es decir $\mathcal{S}$ no se anula en K.

Concluimos que $\mathcal{S}$ es un álgebra con unidad que separa puntos de K, de modo que cumple las hipótesis del Teorema de Stone–Weierstrass. Por dicho teorema,

$$\overline{\mathcal{S}}^{\|\cdot\|_\infty} = C(K).$$

En particular, para cada $f \in C(K)$ y $\varepsilon > 0$ existe una red de una sola capa oculta $N \in \mathcal{S}$ con $\|f - N\|_\infty < \varepsilon$.

□

4.3. Arquitectura y funciones de activación

Como se ha observado en [9], el rendimiento de una red neuronal depende críticamente de su arquitectura y de las funciones de activación empleadas, ya que estos factores determinan su capacidad de aproximación y la facilidad de su entrenamiento.

A continuación se describe la arquitectura final adoptada, junto con la motivación de cada decisión de diseño.

Principios de diseño

1. **Compatibilidad con la entrada.** El vector de estado $x \in \mathbb{R}^{40}$ codifica cartas, posición, SPR, etc. (véase 5.2) y, por construcción, ya está normalizado a $[0, 1]$.

2. **Suficiente capacidad, pero sin sobreajuste.** Después de probar varios tamaños de capas y de neuronas por capa, hemos llegado a la conclusión de que dos capas ocultas de tamaño medio (64 y 32 neuronas) ofrecen un equilibrio razonable entre capacidad de representación y riesgo de sobreajuste.

3. **Funciones de activación robustas.** Hemos empleado ReLU en las capas ocultas por su sencillez, bajo coste computacional y buena propagación del gradiente. Otras opciones (sigmoide o *tanh*) satisfacen el Teorema de Aproximación Universal, pero en la práctica producen gradientes saturados.

4. **Salida probabilística.** La capa de salida aplica la función *softmax*, definida para $z = (z_1, \ldots, z_5) \in \mathbb{R}^{|A|} = \mathbb{R}^5$ como:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{5} e^{z_j}}, \quad i = 1, \ldots, 5.$$

   Esto garantiza que la salida $\hat{y} = \text{softmax}(z)$ es un vector con componentes positivas que suman uno, es decir, una distribución de probabilidad sobre las cinco acciones posibles.

En resumen, la combinación de una arquitectura con varias capas ocultas y funciones de activación no lineales permite a la red modelar con precisión la función de decisión óptima del jugador en contextos de juegos de información incompleta.

Observación 4.7. Aunque $\text{ReLU}(x) = max\{0, x\}, \quad x \in \mathbb{R}$ no es una función sigmoide ya que no está acotada,

$$\lim_{x \to +\infty} \text{ReLU}(x) = +\infty \neq 1$$

las redes con activación ReLU siguen siendo aproximadoras universales. Leshno, Lin, Pinkus y Schocken demostraron que cualquier activación continua y no polinómica confiere universalidad a una red *feedforward* [11]; ReLU cumple ambas condiciones, por lo que mantiene la capacidad de aproximar arbitrariamente bien cualquier función $f \in C(K)$.

4.4. Entrenamiento y optimización de la red

El objetivo es ajustar los parámetros θ de la red neuronal para que la política aprendida, es decir, el modelo de decisión entrenado con la base de datos disponible $\hat{y} = \pi_\theta(x)$ se aproxime a la política de referencia generada por *Pluribus*. Formulamos el problema como la minimización de la función de coste

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(\pi_\theta(x^{(i)}),\, y^{(i)}) + \lambda \|\theta\|_2^2,$$

donde:

- θ agrupa todos los parámetros de la red (pesos y sesgos),

- $\{x^{(i)}, y^{(i)}\}$ es el conjunto de entrenamiento, concretamente son los pares de entrada y salida tomados del dataset de Pluribus,
- ℓ es la función de pérdida (ver más abajo),
- $\lambda > 0$ es el coeficiente de regularización ℓ_2.

Función de pérdida: entropía cruzada

Dado que el objetivo es aproximar una distribución sobre las acciones (estrategia mixta), la función de pérdida natural es la entropía cruzada entre la distribución estimada $\hat{y} = \pi_\theta(x)$ y la distribución empírica $y^{(i)}$. Se define como:

$$\ell(\hat{y}, y) := -\sum_{k=1}^{5} y_k \log \hat{y}_k.$$

En la práctica, si y es una distribución unitaria en la acción a, la función entropía cruzada se reduce a

$$\ell(\hat{y}, y) \;=\; -\log\big(\hat{y}_a\big),$$

Este resultado se puede comprobar en [9]

Regularización

El segundo término de la función objetivo penaliza la norma cuadrada de los parámetros. La regularización ℓ_2 ayuda a prevenir el sobreajuste, especialmente en contextos con ruido, entrada de alta dimensión o datasets de tamaño limitado. En este trabajo se fija $\lambda = 10^{-4}$, valor típico en redes densas de tamaño medio.

Algoritmo de optimización

En su forma básica, el algoritmo de descenso por gradiente actualiza los parámetros en la dirección opuesta al gradiente:

$$\theta^{(t+1)} \;=\; \theta^{(t)} - \eta\, \nabla_\theta \mathcal{L}\big(\theta^{(t)}\big)$$

donde $\eta > 0$ es la tasa de aprendizaje. Calcular $\nabla_\theta \mathcal{L}$ sobre todo el conjunto de datos es costoso, por lo que se emplea la versión estocástica (SGD): que actualiza los parámetros utilizando mini-lotes de entrenamiento (*mini-batches*) en lugar de todo el conjunto de datos. La regla de actualización en este caso es:

$$\Theta^{(t+1)} = \Theta^{(t)} - \eta \nabla_\Theta \ell\big(f_\Theta(x^{(t)}), y^{(t)}\big),$$

Para mejorar la velocidad de convergencia y la estabilidad del entrenamiento, se emplea el optimizador Adam (*Adaptive Moment Estimation*), que combina las ventajas del descenso por gradiente con momentum y la normalización de tasas de aprendizaje por parámetro. Adam utiliza estimaciones de primer y segundo momento (media y varianza) de los gradientes para ajustar dinámicamente cada componente de Θ a lo largo del proceso de entrenamiento [10].

En este trabajo hemos decidido usar los siguientes hiperparámetros estándar:

- tasa de aprendizaje inicial: $\eta = 10^{-3}$,
- tamaño de batch (SGD): 128,
- coeficiente de regularización: $\lambda = 10^{-4}$,

La implementación se ha realizado en `scikit-learn` para compatibilidad con otras partes del proyecto. No obstante, se verificó que los resultados coinciden con implementaciones equivalentes en `PyTorch`.

5. Integración de Redes Neuronales con Juegos Bayesianos

El objetivo de esta sección es mostrar cómo la estrategia de un jugador en un juego bayesiano puede ser representada mediante una red neuronal entrenada sobre datos simulados por un agente óptimo (por ejemplo Pluribus). Este enfoque es coherente con el marco teórico del aprendizaje por refuerzo y el aprendizaje supervisado, donde una función objetivo se aproxima a partir de ejemplos observados [10]. Además, se discuten las implicaciones de interpretar la red resultante como una aproximación funcional a una estrategia bayesiana: si bien el modelo es determinista en su forma (parámetros fijos), su salida probabilística puede entenderse como una política mixta condicionada a la información observada.

Para aproximar la estrategia óptima de Pluribus como un problema de clasificación supervisada, definimos primero la función de decisión y luego detallamos la codificación de datos, la arquitectura de la red y el pipeline de entrenamiento.

5.1. Modelado de la función de decisión

En un juego bayesiano, la estrategia de un jugador i se concibe como una función que, dada la información disponible (su tipo privado y la información común observada), devuelve una acción o una distribución sobre acciones. Denotemos:

- Θ_i es el espacio de tipos del jugador i, que representa su información privada (por ejemplo, su mano en una partida de póker),
- Θ^0 es la iformación que tienen todos los jugadores en común del juego: secuencia de apuestas y cartas comunitarias reveladas hasta el momento,
- A_i es el conjunto de acciones disponibles, en nuestro caso: {*fold, check, call, bet, raise*}.

Formalmente, la estrategia pura ideal sería

$$s_i : \Theta^0 \times \Theta_i \longrightarrow A_i,$$

o bien, el jugador puede utilizar una estrategia mixta,

$$\pi_i : \Theta^0 \times \Theta_i \longrightarrow \Delta(A_i),$$

donde $\Delta(A_i)$ es el conjunto de todas las distribuciones de probabilidad sobre A_i (véase 2.2). En la práctica, entrenaremos una red neuronal parametrizada $f_\Theta : \mathbb{R}^d \to \mathbb{R}^{|A_i|}$que, a partir de una codificación numérica $x = \text{encode}(\theta^0, \theta_i) \in \mathbb{R}^d$, produce un vector de logits $z \in \mathbb{R}^{|A_i|}$capaz de aproximar la estrategia del jugador, $f_\Theta(x) \approx s_i(\theta^0, \theta_i)$, donde $f_\Theta(x)$ produce un vector de puntuaciones o probabilidades sobre las acciones posibles. La salida final se interpreta como una política estocástica, $\hat{\pi}(a \mid x)$, que aproxima la estrategia del jugador en cada situación de juego.

Codificación en espacio numérico

Definimos una función de codificación

$$\text{encode} : \Theta^0 \times \Theta_i \longrightarrow x \in \mathbb{R}^d,$$

que resume la información relevante del estado del juego. Ver sección 5.2 para el detalle de cada componente de x. Esta codificación debe:

- Preservar toda la información necesaria para la toma de decisión (cartas privadas, comunitarias, tamaño de pila/bote, historial, posición, SPR, etc.).
- Evitar fugas de información ("data leakage") que hagan trampa al modelo.
- Mantener dimensión fija d, mediante relleno o codificaciones apropiadas cuando cierta información aún no esté disponible (p. ej. cartas comunitarias no reveladas).

Bajo la hipótesis de que la función de decisión óptima (o cuasi-óptima) es continua en la entrada codificada y el dominio de inputs puede verse como un compacto razonable, el Teorema de Aproximación Universal garantiza la existencia de una red suficientemente amplia (ver sección 4) capaz de aproximar arbitrariamente bien dicha función. En la práctica, el entrenamiento con datos busca encontrar parámetros que hagan real esta aproximación.

En las siguientes secciones se describe con detalle la arquitectura utilizada para implementar la red neuronal así como el procedimiento de entrenamiento y evaluación del modelo.

5.2. Codificación y entrenamiento

Para aplicar el modelo descrito en la sección anterior, transformamos las entradas (θ^0, θ_i) y las salidas $a \in A_i$ a formatos numéricos compatibles con una red neuronal. Esta etapa de codificación es crítica, ya que una representación adecuada del estado del juego permite al modelo capturar patrones estratégicos relevantes.

Codificación de la entrada $x \in \mathbb{R}^d$

El vector de entrada $x \in \mathbb{R}^d$ se construye mediante la concatenación de varias representaciones codificadas que resumen la información relevante del estado de juego:

- **Cartas privadas del jugador:** cada carta se codifica como un vector de 5 componentes: un valor normalizado entre 0 y 1 (correspondiente al rango de 2 a 14) y un vector *one-hot* de dimension 4 que representa el palo (clubs, diamonds, hearts, spades). Las dos cartas generan un vector de dimensión $2 \times 5 = 10$.
- **Cartas comunitarias:** hasta cinco cartas comunitarias se codifican de la misma manera, y si no están todas reveladas (por ejemplo, en preflop o flop), se rellenan con ceros. Se obtiene un vector de dimensión fija de $5 \times 5 = 25$.

- **Calle del juego (street):** se codifica como un valor escalar normalizado en el intervalo $[0, 1]$, siendo 0 para preflop y 1 para river.
- **Posición en la mesa:** codificada como un entero entre 1 y 6 (según el asiento), normalizado al intervalo $[0, 1]$.
- **Relación stack-bote (SPR):** se codifica como un escalar $\text{SPR} = \frac{\text{stack}}{\text{pot}}$ en el momento de la acción. Se normaliza al intervalo $[0, 1]$.
- **Número de jugadores activos:** escalar que indica cuántos jugadores siguen activos en la mano, normalizado al intervalo $[0, 1]$.
- **Dinero ya invertido por el jugador:** representa cuánto ha apostado el jugador en la mano hasta ese momento. Se usa como un escalar continuo normalizado al intervalo $[0, 1]$.

Estas representaciones se concatenan para formar un vector de entrada $x \in \mathbb{R}^{40}$ de dimensión fija. No se incluye información explícita sobre la última acción realizada en la mesa para evitar *data leakage* o sesgos artificiales que puedan sobreestimar la capacidad predictiva del modelo.

Codificación de la salida $y \in A_i$

El conjunto de acciones posibles se modela como un problema de clasificación multiclase. Las acciones consideradas son:

$$A = \{\textit{fold}, \textit{call}, \textit{raise}, \textit{bet}, \textit{check}\}$$

Cada acción se representa mediante una codificación entera, tratada internamente por la función de pérdida de clasificación.

Conjunto de entrenamiento

Los datos de entrenamiento se extraen de registros reales del bot Pluribus. Cada muestra corresponde a una decisión del bot, acompañada del contexto de la mano. Cada muestra se representa como un par (x, y), donde:

- x es el vector codificado que representa la información parcial del jugador,
- y es la acción ejecutada en ese contexto.

Se construye un conjunto de entrenamiento de tamaño m, dividido en subconjuntos de entrenamiento y validación (80 %–20 %). Para evitar problemas derivados del desequilibrio entre clases (por ejemplo, pocas acciones de `raise`), se emplean técnicas como:

- Reponderación de clases mediante `class_weight='balanced'`,
- Sobremuestreo de acciones minoritarias si es necesario.

Tarea de aprendizaje

La red neuronal recibe el vector $x \in \mathbb{R}^d$ y predice una distribución de probabilidad sobre las acciones posibles. Se utiliza la entropía cruzada como función de pérdida y se optimiza mediante el algoritmo Adam. El modelo se entrena siguiendo un enfoque de clasificación multiclase supervisada, encuadrado dentro del aprendizaje bayesiano por observación de conducta.
Este planteamiento convierte el problema en una tarea clásica de clasificación multiclase supervisada, encuadrada en el contexto más amplio del aprendizaje bayesiano por observación de conducta.

5.3. Arquitectura y función de pérdida

Para aproximar la función de decisión s_i, modelamos un clasificador multiclase mediante una red neuronal de tipo perceptrón multicapa (MLP), tal como se describió en la sección 4. La arquitectura ha sido diseñada para equilibrar expresividad, eficiencia computacional y capacidad de generalización.

La red implementa una función $f_\Theta : \mathbb{R}^d \to \mathbb{R}^{|A_i|}$, donde:

- $\mathbb{R}^d$ es el espacio de entrada, que representa el vector codificado $x = (\theta^0, \theta_i)$,
- $\mathbb{R}^{|A_i|}$ es el espacio de salida, correspondiente a las acciones posibles del jugador.

Como se expuso en la sección 4.3, empleamos una red MLP(40–64–32–5) con ReLU en ocultas y softmax en salida. En este experimento concreto, los hiperparámetros elegidos son:

- Tasa de aprendizaje inicial $\eta = 10^{-3}$, con reducción al estancarse la validación.
- Regularización ℓ_2 con $\lambda = 10^{-4}$.
- Tamaño de batch: 128.
- Número máximo de épocas: 100, con parada temprana basada en pérdida de validación.
- Ponderación de clases en la entropía cruzada para mitigar desequilibrios (peso mayor para acciones raras).

$$
\begin{aligned}
& x \in \mathbb{R}^{40} \\
& h^{(1)} = \mathrm{ReLU}\big(W^{(1)}x + b^{(1)}\big), && W^{(1)} \in \mathbb{R}^{64\times 40} \\
& h^{(2)} = \mathrm{ReLU}\big(W^{(2)}h^{(1)} + b^{(2)}\big), && W^{(2)} \in \mathbb{R}^{32\times 64} \\
& z = W^{(3)}h^{(2)} + b^{(3)}, && W^{(3)} \in \mathbb{R}^{5\times 32} \\
& \hat{y} = \mathrm{soft\,max}(z) \in \Delta^4.
\end{aligned}
$$

Donde $\Delta^4 = \{p = (p_1, ..., p_5) \in \mathbb{R}^5 | p_i \geq 0 \text{ y } \sum_1^5 p_i = 1\}$.

La función de pérdida es entropía cruzada regularizada (ver la sección 4.4). Cualquier variación experimental (p. ej. incluir dropout del 0.2) se documenta al describir los resultados en la sección 6.

Esta arquitectura y función de entrenamiento permiten aproximar con precisión la política del bot Pluribus en un entorno bayesiano con información incompleta.

5.4. Aproximación bayesiana y generalización

Aunque el modelo f_Θ entrenado es una red neuronal determinista, su comportamiento puede interpretarse como una aproximación funcional a una estrategia bayesiana. En efecto, el jugador observa un vector de información parcial $x = (\theta^0, \theta_i)$, a partir del cual debe inferir la acción óptima teniendo en cuenta la incertidumbre sobre los tipos de los demás jugadores.

La red neuronal aprende esta correspondencia a partir de datos generados por Pluribus, que actúa siguiendo una estrategia óptima o cuasi-óptima en contextos de información incompleta. Al entrenarse sobre estos datos, la red replica de forma empírica la función de decisión s_i que maximiza la utilidad esperada bajo incertidumbre.

Este enfoque puede interpretarse como una forma de *aprendizaje bayesiano indirecto*, en el sentido de que el modelo no infiere explícitamente distribuciones de probabilidad sobre los estados ocultos, pero sí aproxima directamente la acción que maximiza el valor esperado, condicionado a la información observada.

Adicionalmente, técnicas como la regularización ℓ_2 y el uso de optimizadores adaptativos como Adam contribuyen a mejorar la capacidad de generalización del modelo a situaciones no vistas, evitando el sobreajuste. En un enfoque más avanzado, podría introducirse inferencia bayesiana aproximada sobre los pesos de la red, mediante métodos como Monte Carlo Dropout o redes bayesianas, para cuantificar la incertidumbre de las predicciones.

En conjunto, el modelo entrenado puede considerarse como una aproximación práctica y eficiente a la estrategia bayesiana del jugador, en el sentido formal descrito por Harsanyi y Zamir [3].

6. Resultados del Entrenamiento y Discusión

Tras definir la codificación de estados y entrenar un modelo de red neuronal utilizando la arquitectura y los parámetros descritos previamente, se evaluó su rendimiento sobre un conjunto de test no visto, así como mediante validación cruzada sobre el conjunto de entrenamiento y se analiza la matriz de confusión para entender los patrones de error.

Evaluación en test

La precisión global obtenida fue de 0,8497. A continuación, se presenta el informe de clasificación desglosado por acción:

Acción	**Precision**	**Recall**	**F1-score**	**Soporte**
bet	0.54	0.53	0.54	220
call	0.72	0.75	0.73	339
check	0.85	0.91	0.88	504
fold	0.94	0.93	0.93	1595
raise	0.76	0.72	0.74	376
Macro avg	0.76	0.77	0.76	3034
Weighted avg	0.85	0.85	0.85	3034

Tabla 2: Informe de clasificación por acción en el conjunto de prueba.

La red alcanza una precisión del 84.97 %, lo que indica una buena capacidad para reproducir la política de Pluribus en la mayoría de los estados.

- Las acciones más frecuentes, como *fold* (f1 = 0.93) y *check* (f1 = 0.88), son también las más fáciles de predecir. Su frecuencia elevada permite al modelo capturar de forma fiable los patrones asociados a jugadas pasivas o conservadoras.
- *Call* y *raise* presentan también buenos resultados (ambas con f1-score cercano a 0.74), lo que sugiere que el modelo distingue correctamente situaciones en las que Pluribus mantiene la mano o adopta una actitud agresiva.
- La clase *bet* es la que presenta más dificultades (f1 = 0.54). Esto puede atribuirse tanto a su frecuencia moderadamente baja como a la ambigüedad contextual: muchas de sus instancias se dan en situaciones intermedias, tácticas o semi-bluff, donde la línea óptima depende de matices difíciles de codificar.

Validación cruzada

Con el objetivo de comprobar si nuestro conjunto de entrenamiento estaba sufriendo sobreajuste respecto a los datos de entrada, se aplicó validación cruzada de 5 particiones (*5-fold cross-validation*) sobre el conjunto de entrenamiento para estimar la capacidad de generalización del modelo. Los resultados obtenidos fueron:

- **Accuracy medio:** 0.831
- **Accuracies individuales:** [0.823 0.834 0.834 0.823 0.839]

La ligera brecha entre el 83 % medio en validación y el 85 % en prueba confirma que no hay sobreajuste apreciable y que el modelo generaliza bien.

Matriz de confusión

La Figura 1 muestra la matriz de confusión, donde las filas corresponden a las acciones reales y las columnas a las acciones predichas. La diagonal principal refleja los aciertos, mientras que las celdas fuera de la diagonal indican confusiones entre pares de acciones.

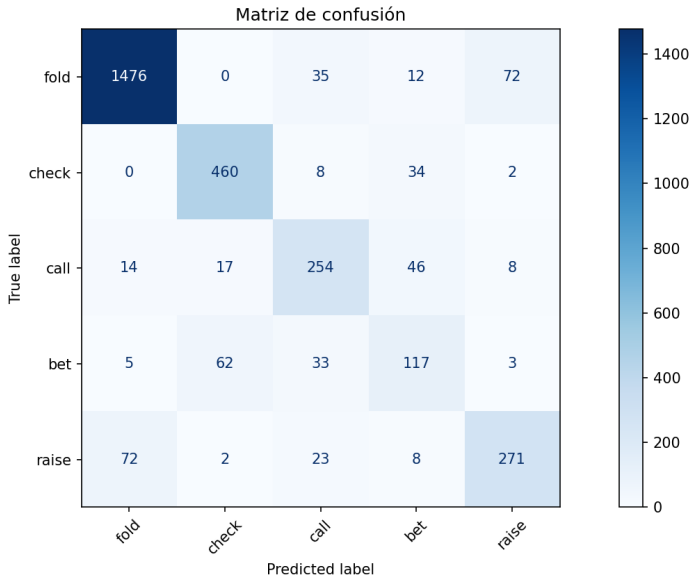

Figura 1: Matriz de confusión en el conjunto de prueba.

- La matriz muestra que el error más frecuente del modelo es no distinguir en algunas situaciones entre *raise* y *fold* y con un total de 144 imprecisiones, 72 en cada sentido. El segundo error más frecuente es elegir *check* cuando el bot juega *bet* y viceversa, con 62 y 34 fallos respectivamente, observamos que en este caso el modelo tiende a elegir la estrategia más pasiva.
- Tiene sentido que el modelo cometa estos errores, no solo por el bajo número de datos disponibles, sino porque pueden ser situaciones muy sutiles y complicadas de diferenciar, y dado que el bot Pluribus usa una estrategia mixta su estrategia puede ser un poco más flexible.

Posibilidades de mejora

- Mejorar el *preprocesado del historial de apuestas*, incorporando más contexto secuencial (por ejemplo, tamaño del último *raise*, posición del agresor, etc.).

- Aplicar *sobremuestreo* o *ponderación de clases* para mitigar el desequilibrio en clases como *bet* y *raise*.

Conclusión final

El modelo MLP(40–64–32–5) logra reproducir de forma robusta la estrategia de Pluribus, con una precisión global del 85 % y métricas F1 superiores al 0.70 en cuatro de las cinco acciones. Las decisiones mayoritarias (*fold*, *check*) se capturan con gran fiabilidad (F1 $\geq$ 0,87), lo que confirma que la red ha internalizado los patrones fundamentales de juego defensivo. Para acciones agresivas, el rendimiento es notable en *raise* y en *call* (F1 $\approx$ 0,74). El único punto claramente mejorable es *bet* (F1 $\approx$ 0,54).

La validación cruzada (5-fold) corrobora la capacidad de generalización de la precisión media del 83.1 % solo tiene una sutil brecha con el 85 % alcanzado en la prueba, por lo que no se detecta sobreajuste relevante. La matriz de confusión nos revela que las acciones que más le cuesta distinguir son *fold* y *raise*.

En conjunto, estos resultados demuestran que la red entrenada constituye una aproximación bayesiana aceptable de la política óptima del bot, reproduciendo con buena fidelidad su comportamiento esperado bajo incertidumbre. Esto la convierte en una herramienta válida tanto para tareas de análisis y visualización de decisiones como para su uso práctico como oponente de entrenamiento en entornos simulados. Además, su eficiencia computacional y capacidad de generalización permiten integrarla en bucles de aprendizaje más amplios, donde pueda interactuar con agentes humanos o automatizados en tiempo real.

Anexo: Código Fuente

En este anexo se incluye el código fuente desarrollado para la implementación del modelo. La estructura consta de tres módulos principales: el script de entrenamiento y evaluación, la codificación de los estados del juego, y el parser para extraer datos desde los logs del bot Pluribus.

Archivo: `main.py`

Listing 1: Script principal: entrenamiento y evaluación

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split,
    cross_val_score
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score,
    confusion_matrix, ConfusionMatrixDisplay

from encodingestados2 import encode_state
from lecturafichero1 import parse_logs

# === CONSTANTES ===
LOG_PATH = "pluribus_logs/*.txt"

# === 1. Cargar los datos ===
print("Leyendo logs de Pluribus...")
df = parse_logs(LOG_PATH)
print(f"Total decisiones de Pluribus: {len(df)}")

# === 2. Limpieza y preparacion ===
df['action'] = df['action'].str.lower().str.strip()
df['action'] = df['action'].replace({
    'folds': 'fold',
    'calls': 'call',
    'raises': 'raise',
    'bets': 'bet',
    'checks': 'check'
})

acciones_validas = ['fold', 'call', 'raise', 'bet', 'check']
df = df[df['action'].isin(acciones_validas)]

# === 3. Funcion para calcular SPR ===
def compute_spr(row):
    pot = row.get('pot_size', 0)
    stack = row.get('stack', 0)
    if pot <= 0:
        return 0.0
    return round(stack / pot, 2)

df['spr'] = df.apply(compute_spr, axis=1)

# === 4. Codificar X e y ===
def encode_row(row):
    hole = [row['hole1'], row['hole2']]
    board = row['board'] if isinstance(row['board'], list) else []
    street_map = {'Preflop': 0, 'Flop': 1, 'Turn': 2, 'River': 3}
```

```
    street_idx = street_map.get(row['street'], 0)
    position = row.get('seat', 0) - 1  # asiento de 1 a 6 -> 0 a 5
    spr = row['spr']
    players_active = row.get('num_players_active', 6)
    invested = row.get('invested_by_pluribus', 0)
    return encode_state(hole, board, street_idx, position, spr,
        players_active, invested)

print("Codificando jugadas...")
X = np.vstack(df.apply(encode_row, axis=1).values)
y = df['action'].values


# === 5. Entrenar y evaluar modelo ===


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)

model = MLPClassifier(hidden_layer_sizes=(64, 32), max_iter=500,
    random_state=42)

print("Realizando validacion cruzada (5-fold)...")
scores = cross_val_score(model, X_train, y_train, cv=5)
print("Accuracy medio validacion cruzada:", scores.mean())
print("Accuracies individuales:", scores)
print("Entrenando red neuronal...")

model.fit(X_train, y_train)

print("Evaluando el modelo...")
y_pred = model.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred))


action_to_int = {
    'fold': 0,
    'check': 1,
    'call': 2,
    'bet': 3,
    'raise': 4
}

y_test_int = [action_to_int[y] for y in y_test]
y_pred_int = [action_to_int[y] for y in y_pred]
cm = confusion_matrix(y_test_int, y_pred_int, labels=[0, 1, 2, 3,
    4])
disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=['
    fold', 'check', 'call', 'bet', 'raise'])
disp.plot(cmap='Blues', xticks_rotation=45)
plt.title('Matriz de confusion')
plt.tight_layout()
plt.savefig("confusion_matrix.png", dpi=300)
plt.show()
```

Archivo: encodingestados.py

Listing 2: Codificación de estados del juego

```
import numpy as np

def card_to_features(card_str):
    """Convierte una carta como 'As' a un vector con valor y palo
        one-hot"""
    ranks = {'2':2, '3':3, '4':4, '5':5, '6':6, '7':7, '8':8, '9'
        :9,
             'T':10, 'J':11, 'Q':12, 'K':13, 'A':14}
    suits = {'c':0, 'd':1, 'h':2, 's':3}

    if len(card_str) != 2:
        return np.zeros(5)  # vector nulo si error

    rank_char, suit_char = card_str[0], card_str[1]
    value = (ranks.get(rank_char.upper(), 0)-2)/12
    suit_vector = np.zeros(4)
    suit_index = suits.get(suit_char.lower())
    if suit_index is not None:
        suit_vector[suit_index] = 1
    return np.array([value] + list(suit_vector))

def encode_cards(cards,max_cards):
    """Codifica una lista de cartas (pueden ser privadas o
        comunitarias)"""
    encoded = [card_to_features(card) for card in cards]
    while len(encoded) < max_cards:  # se rellenan hasta 7 con
        ceros (hole+board)
        encoded.append(np.zeros(5))
    return np.concatenate(encoded)

def encode_state(hole_cards, board_cards, street, position, spr=
    None, num_players_active=None, invested_by_pluribus=None):
    """
    Devuelve el vector de estado codificado para la red neuronal
    - hole_cards: lista con 2 cartas propias, ej. ['As', 'Kc']
                                10
    - board_cards: lista con cartas comunitarias
                                          25
    - street: int (0=preflop, 1=flop, 2=turn, 3=river)
                                       1
    - position: int (0=UTG, ..., 5=SB, 6=BB), rango depende del
        numero de jugadores     1
    - spr: stack-to-pot ratio (float o None)
                                                  1
    - num_players_active: int (1 a 6), normalizado
                                               1
    - invested_by_pluribus: float, dinero invertido (normalizado)
                             1
    """
    hole_encoded = encode_cards(hole_cards,2)
    board_encoded = encode_cards(board_cards,5)

    street_vec = (np.array([street]))/3
    position_vec = np.array([position])/6
    spr_vec = np.array([spr if spr is not None else 0.0])
    players_vec = np.array([num_players_active / 6.0 if
        num_players_active is not None else 0.0])
    invested_vec = np.array([invested_by_pluribus / 10000.0 if
        invested_by_pluribus is not None else 0.0])
```

```
    return np.concatenate([hole_encoded, board_encoded, street_vec,
        position_vec, spr_vec, players_vec, invested_vec])
```

Archivo: `lecturafichero.py`

Listing 3: Lectura y procesamiento de logs

```
import re
from glob import glob
import pandas as pd

# Expresiones regulares
HAND_START = re.compile(r"^PokerStars Hand #\d+:")
SEAT_LINE = re.compile(r"Seat (\d+): (\w+) \((\d+) in chips\)")
DEALT_LINE = re.compile(r"Dealt to Pluribus \[([2-9TJQKA][shdc])
    ([2-9TJQKA][shdc])\]")
ACTION_LINE = re.compile(r"^(\w+): (folds|checks|calls(?: \d+)?|
    bets \d+|raises \d+ to \d+)")
BOARD_LINE = re.compile(r"\*\*\* (FLOP|TURN|RIVER) \*\*\*
    \[([^\]]+)\]")
SUMMARY_LINE = re.compile(r"\*\*\* SUMMARY \*\*\*")


def parse_logs(path_pattern):
    hands = []
    for path in glob(path_pattern):
        with open(path, 'r') as f:
            lines = f.readlines()

        current_hand = []
        for line in lines:
            if HAND_START.match(line):
                if current_hand:
                    hands.append(current_hand)
                current_hand = [line.strip()]
            else:
                current_hand.append(line.strip())
        if current_hand:
            hands.append(current_hand)

    print(f"Total manos detectadas: {len(hands)}")
    return _extract_pluribus_actions(hands)



def _extract_pluribus_actions(hands):
    records = []

    for hand in hands:
        players = {}
        board = []
        street = 'Preflop'
        hole_cards = ('??', '??')
        pot = 0
        last_action = 'none'
        folded = set()
        invested=0
        pluribus_stack=0
```

```
for i, line in enumerate(hand):
    # Extraer asientos y stacks
    m = SEAT_LINE.match(line)
    if m:
        seat, name, stack = m.groups()
        players[name] = {'seat': int(seat), 'stack': int(
            stack)}
        for n in name:

        if name == "Pluribus":
            pluribus_stack = int(stack)
            if seat == 1:
                invested+=50
            elif seat == 2:
                stack-=100
                invested+=100

        continue

    # Cartas privadas
    m = DEALT_LINE.match(line)
    if m:
        hole_cards = m.group(1), m.group(2)
        continue

    # Board
    m = BOARD_LINE.match(line)
    if m:
        street = m.group(1).capitalize()
        board = m.group(2).split()
        continue

    # Accion
    m = ACTION_LINE.match(line)
    if m:
        actor, action_raw = m.groups()
        amount=0
        if actor in folded:
            continue

        # Sumar al bote si la accion tiene cantidad
        if action_raw.startswith("bets"):
            amount = int(action_raw.split()[1])
            pot += amount
        elif action_raw.startswith("calls"):
            try:
                amount = int(action_raw.split()[1])
                pot += amount
            except:
                pass
        elif action_raw.startswith("raises"):
            amount = int(action_raw.split()[-1])
            pot += amount
        elif action_raw == 'folds':
            folded.add(actor)

        # Guardar accion de Pluribus
        if actor == 'Pluribus':
            act = action_raw.split()[0]
            if act in ['posts', 'Uncalled', 'collected']:
                continue
```

```
                    records.append({
                        'hole1': hole_cards[0],
                        'hole2': hole_cards[1],
                        'board': board,
                        'street': street,
                        'action': act,
                        'seat': players.get('Pluribus', {}).get('
                            seat', -1),
                        'stack': pluribus_stack,
                        'num_players_active': 6 - len(folded),
                        'pot_size': pot,
                        'invested_by_pluribus': invested,
                    })
                    invested += amount
                    pluribus_stack -= amount

            if SUMMARY_LINE.match(line):
                break

    print(f"Manos con decisiones de Pluribus extraidas: {len(
        records)}")
    return pd.DataFrame(records)
```

Referencias

[1] John C. Harsanyi, *Games with Incomplete Information Played by "Bayesian" Players, Parts I–III*, Management Science, 1967–1968. Disponible en: `http://www.dklevine.com/archive/refs41175.pdf`

[2] Jonathan Levin, *Games of Incomplete Information*, Lecture Notes, Stanford University. Disponible en: `https://web.stanford.edu/~jdlevin/Econ%20203/Bayesian.pdf`

[3] Shmuel Zamir, *Bayesian Games: Games with Incomplete Information*, Notas de curso. Disponible en: `http://www.ma.huji.ac.il/~zamir/documents/Bayesian`

[4] John von Neumann, *Biografía y aportes a la teoría de juegos*, Disponible en: `https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/neumann.html`

[5] G. Cybenko. *Approximation by superpositions of a sigmoidal function.* Mathematics of Control, Signals, and Systems, 1989, 2 (4), pp.303-314. ff10.1007/BF02551274ff. ffhal-03753170f

[6] K. Hornik, M. Stinchcombe, and H. White, *Multilayer feedforward networks are universal approximators*, Neural Networks, **2**(5), 359–366, 1989.

[7] W. Rudin, *Principios de Análisis Matemático*, Tercera edición, McGraw-Hill, 1991. (Original: *Principles of Mathematical Analysis*, 3rd ed., 1976)

[8] Charles Fefferman, *Neural Nets*, Disponible en: `https://www.math.stonybrook.edu/~bishop/classes/math533.S21/MachineLearning/Fefferman_NeuralNets.pdf`

[9] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016.

[10] Richard S. Sutton y Andrew G. Barto, *Reinforcement Learning: An Introduction*, Disponible en: `https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf`

[11] M. Leshno, V. Y. Lin, A. Pinkus, S. Schocken, *Multilayer feedforward networks with a nonpolynomial activation function can approximate any function*, Neural Networks, **6**(6), 861–867, 1993.

[12] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, Proc. ICLR 2015.