Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales

Ajedrezzinni Latentinni: autoencoders para interpretar el espacio latente del ajedrez

Tesis de Licenciatura en Ciencias de Datos

Julián Garbulsky

Director: Juan Pablo Pinasco

Buenos Aires, 2025

AJEDREZZINNI LATENTINNI: AUTOENCODERS PARA INTERPRETAR EL ESPACIO LATENTE DEL AJEDREZ

En este trabajo exploramos representaciones comprimidas de posiciones de ajedrez mediante autoencoders, una clase de redes neuronales no supervisadas. A partir de una base de datos pública de partidas de ajedrez, filtramos posiciones jugadas entre jugadores de alto nivel y las representamos en formato binario (usando una variante del bitboard). Entrenamos un autoencoder con el objetivo de reconstruir estas posiciones, y medimos la calidad de la reconstrucción tanto con el error cuadrático medio como con una métrica específica del dominio que contabiliza cuántas casillas del tablero fueron mal reconstruidas.

Para estudiar la interpretabilidad de las representaciones latentes, aplicamos análisis de componentes principales (PCA) y visualizamos cómo las aperturas de ajedrez, así como ciertas estructuras como el enroque y los peones centrales, se reflejan en las primeras componentes principales. Encontramos que algunas de estas características estratégicas o posicionales se correlacionan con direcciones particulares del espacio latente.

Finalmente, evaluamos la robustez del autoencoder aplicándolo a posiciones significativamente distintas a las del entrenamiento, como las jugadas por principiantes o las generadas artificialmente mediante simetrías. En ambos casos, observamos un desempeño de reconstrucción menor, lo que sugiere que el modelo captura regularidades propias de partidas de nivel intermedio a alto.

Palabras claves: Ajedrez, Autoencoders, Aprendizaje no supervisado, Representaciones latentes, PCA, Interpretabilidad, Reconstrucción.

AJEDREZZINNI LATENTINNI: AUTOENCODERS FOR INTERPRETING THE LATENT SPACE OF CHESS

In this work, we explore compressed vector representations of chess positions using autoencoders, a class of unsupervised neural networks. Our goal is twofold: to obtain a compact encoding that captures the essential structure of a chess position, and to understand to what extent strategic or positional features are preserved in the latent space.

We begin by preprocessing a public Lichess dataset, filtering for standard-rated games with a minimum time control and players rated above 2100. Chess positions are encoded using a variant of the standard bitboard representation: we represent each square of the board using a one-hot encoding over 13 channels—one for each of the 12 piece types and one additional channel for empty squares. This results in a 832-dimensional binary vector for each position. This representation has the advantage of allowing direct interpretation of the model's outputs, which are real-valued, by choosing the most activated channel for each square.

We train an autoencoder on a large sample of midgame positions to minimize reconstruction error. The encoder compresses the input vectors into a lower-dimensional latent representation, while the decoder attempts to reconstruct the original position. The model's performance is evaluated using both the mean squared error and a domain-specific metric based on the number of incorrectly reconstructed squares.

To study the structure of the latent space, we apply Principal Component Analysis (PCA) to the compressed vectors and visualize the projections along the first components. We observe that chess openings, castling maneuvers, and pawn formations often correspond to distinguishable regions in the PCA plane, suggesting that the autoencoder has captured meaningful chess regularities.

We further examine the model's robustness by testing it on positions drawn from out-of-distribution sources. These include positions from beginner-level games and artificially transformed positions obtained via the board symmetry of vertical reflections. In both settings, the model's reconstruction error increases, reinforcing the idea that the autoencoder learns typical patterns from high level chess and not simply the position of each single piece.

In summary, we demonstrate that autoencoders can learn to represent and reconstruct chess positions with high accuracy while preserving structural features in a low-dimensional space, and we highlight how visualization and domain-specific decoding choices can enhance interpretability in unsupervised settings.

Keywords: Chess, Autoencoders, Unsupervised learning, Latent space, PCA, Interpretability, Reconstruction, Out-of-distribution analysis, One-hot encoding.

AGRADECIMIENTOS

El primer agradecimiento es para todos. El otro día en la defensa de tesis y en los posteriores sanguchitos la pasé genial con todos ustedes. Disfruté mucho también el proceso de pensar qué poner en las diapositivas para hacer reír a cada grupo de personas, con algunas referencias escondidas involucradas. Sigo sin poder creer cuántos éramos, y me quedó claro en el momento en que fui a buscar una porción de chocotorta y ya habían desaparecido las dos.

También a los muchos de ustedes que me ayudaron a preparar la presentación escuchando ensayos y aportando ideas buenísimas sobre cómo contar las cosas que se hicieron en la tesis: Bruno Glecer, Massi, Pili y Nahue, Conejillos (Mateo, Lu, Maggy, Marti, Chiari, Valen, Pedro Raigorodsky, Cami Pinat, Coni), Outer Wilds (Adro, Ilu, mi viejo), Bruno Giordano, Ale, Joaco Bermejo y obviamente Juan Pablo.

Sigo con la persona más importante del proceso de la tesis: Juan Pablo Pinasco. No solo por coparse desde el primer día a ser mi director sino también por lo bien que la pasé en el medio pensando ideas, manijeando posibles proyectos y yéndonos por las ramas charlando de otras cosas que nos hicieron reír. Con él nos habíamos conocido en la facu varios años atrás y en el medio ya habíamos tirado ideas de posibles proyectos que combinen aprendizaje automático y ajedrez. Me encantó finalmente haberlo hecho juntos.

El siguiente es para toda la familia. Hay miles de cosas para decir, pero hoy voy a ir con algunas bien específicas. A mi viejo por hacer que me guste mucho el ajedrez desde chiquito. A mi vieja por haberme ayudado a decorar las chocotortas que comimos después de la defensa como las figuras 4.12 y 4.13. A Lele por ser la persona más creativa del planeta a la hora de hacerme reír, además de por decidir escuchar mi sabio consejo sobre la dirección en la que tenía que girar realmente la tuerca si queríamos cambiar la rueda rota del auto yendo a Tandil. Del lado de mi vieja, a Abu y el Tío Loco por ser las personas que más llenan mi vida de Cindor y de anécdotas delirantes. A Vale, Martín, Ludmi, Joán y Carola por haber introducido el tereré en mi vida y ser los directos responsables de la cantidad de litros que tomé desde que empecé la facu, y en esta ocasión a Ludmi especialmente por haberme ayudado a elegir el título de la tesis. Y del lado de mi viejo a todos por lo bien que me la hacen pasar en los viajes y lo mucho que me hacen reír todos los días (hay una referencia escondida en esta tesis para que busquen ustedes). Pero esta vez a algunos especialmente. A Guidín por ser el primero que me habló de la nueva carrera de Ciencia de Datos que estaba por aparecer. A Tein por pasar en muy poco tiempo de ser el primo al que le enseño ajedrez a ser el primo que me enseña ajedrez y por haber tirado ideas copadas para la tesis desde el principio. Y a Cande por subirme la heladera.

A los Avetta no solo por venirse a Buenos Aires especialmente para la defensa, sino también por las ganas con las que siempre quieren recibirme en San Nicolás, que toquemos temas de Coldplay en la guitarra y que les diga qué otra cosa necesito que se pueda resolver con la máquina de coser.

A las peronas (de la facu y de afuera) con las que más compartimos momentos o temas que nos apasionan: A CMC por ser mis amigos de toda la vida. A Ale por la infinidad de proyectos y actividades que hicimos juntos desde muy chiquitos. A Mati Bergerman

por las ganas que tiene siempre de conocer todos los detalles de cómo funcionan las cosas y preguntar por lo que estoy haciendo hasta entenderlo en profundidad y aportar ideas geniales. A Valen por compartir la pasión de conocer personas que hagan cosas que nos dan mucha intriga y llenarlos a preguntas, en un episodio de Desayuno Podcast o en la vida en general. A Bruno Glecer por ser la persona que me hace sentir que charlando con él se puede entender cualquier cosa, y también por ser la persona con la que no podemos hacernos chistes porque a esta altura ya se nos ocurren los mismos remates bien específicos al mismo tiempo. A Massi porque al día de hoy sigo sin entender cómo hizo, porque con los años nunca me dejó de pasar de aprender algo de matemática (cada vez más avanzado) y en algún momento darme cuenta de "che esto es lo que me había contado Massi cuando estábamos haciendo el CBC", y también es la persona con la que cuando hago matemática mejor me hace entender en castellano qué es conceptualmente lo que estamos haciendo. A Juli1 por ser la que me hace entender las ideas más abstractas de la matemática manejándolas como si fueran de lo más cotidiano, y también por ser la única persona con la que nos reímos tanto del humor más absurdo (ver Figura 0.1).

Fig. 0.1: si no te reíste no sos Juli1.

A Cami Mildiner por pasar de ser la persona a la que le cuento en qué consiste la carrera de Matemática a ser la persona a través de quien conozco a más otras personas de la facu, por los niveles de manija que maneja para organizar actividades sociales como Fulbo y Conitos. Al igual que el Chino Cribioli en su tesis, agradecimiento para él y Bruno Giordano por la manera tan única en la que nos hacemos reír. Son ellos los que estando juntos en Córdoba me hicieron conocer muchas de las estructuras de humor que ahora me divierten todos los días (pero bueno, igual me lo tengo que fumar). A Pedro Raigorodsky por los altos niveles en sangre que tiene de pasión por la matemática, y por ser con quien contarnos de un proyecto que tenemos se puede transformar automáticamente en una videollamada todas las semanas (como lo fue con "BienAI!"). A Gabi Sac porque a pesar de ser el chico con el que en ExpC2014 casi no llegué a cruzar una palabra, después nos fuimos enterando de cada vez más cosas que compartíamos (OMA, electrónica, AwesomeMath, la carrera, etc.) hasta terminar siendo él con quien fundamos el Taller de Ingenio, haciéndome cumplir por primera vez mi sueño de dar un taller de matemática recreativa. A Mati Saucedo por ser uno de los más directos responsables de que haya terminado estudiando en Exactas, contagiando a su manera muy única su pasión por resolver problemas como profe en la secundaria, en Exactas y como amigo, y porque al día de hoy cada vez que lo escucho explicar algo, su manera de hacerlo sigue siendo siempre más clarísima de lo que puedo

recordar. A Juampi De Rasis por ser de mis amigos que más me orientaron cuando entré a la carrera de Matemática, siempre adelantándome lo más geniales que se ponían las cosas cuando uno sigue generalizando. A Dina por coparse tanto a hacer experimentos de física desde que nos conocemos, ya sea haciendo malabares dentro de un avión acelerando en la pista de despegue o juntándonos a hacer un espejo parabólico para hacer un radiotelescopio aunque solo termine siendo una máquina de encandilarnos con el sol. A Juli3 por convencer a Bruno de que el Outer Wilds sí me iba a gustar a pesar de ser ficción, y ponerlo en práctica juntándonos los tres regularmente para que me vieran jugar y en el medio yo no pueda dejar de pensar en el juego por la intriga con la que me quedaba. A Diego Fernández Slezak porque desde que lo conocí cuando estaba en el CBC que siempre tiene ganas de contarme aplicaciones copadas de las cosas que él conoce y que yo estoy por aprender, y por coparse a ser mi tutor de la carrera de Datos y la paciencia con la que me ayuda a entender qué trámites tengo que hacer cuándo. A Dylan Fridman por compartir desde hace tanto la manera tan estratégica de pensar cuál es el próximo paso en la investigación que estemos haciendo (por ejemplo Proyecto Cuadrado) o cuál es el próximo acorde en la canción que estemos componiendo. A Santi Aranguri por las ganas que tiene siempre de pensar las cosas. A Lauti Borrovinsky por las juntadas para preparar charlas y tocar música.

A la gente que no puedo cruzarme en el pasillo y salir sin aprender algo nuevo de matemática o de datos: Chanu (con algún problemón y su respectiva solución elegante involucrados), Fran Valdés (con alguna sugerencia de en qué orden atípico cursar las materias involucrada), Juli4 (con algún dato de Eurovisión involucrado), Zenón (con topología y memes involucrados), Rocco (con ideas de estadística y maneras de organizarse bien en la vida involucradas), Lucho Cassini (con principios filosóficos involucrados, y probablemente preparar otro final juntos), Lula Chechic (con neurociencia y tocar un temón de Miranda en la guitarra involucrados), Aye y Lucas Vitali (con divulgación, grafos y comida involucrados), Gasti Zabala (con datos curiosos sobre lo que sea involucrados), Pollo (con tremendas notas en la guitarra involucradas), Pedro Sánchez Terraf (con conjuntos y memes bizarros involucrados).

Sección Futbol: A todos los que son parte del Fulbo de los findes en el poli y a todo el equipo de Conitos FC, por lo bien que me la hacen pasar pateando la pelota, siendo de los mejores ejemplos de por qué a la facu me gusta llamarla "El Club". Y ya que estamos en tema, al Dibu por la atajada más importante de nuestra generación.

A los que me ayudaron especialmente con la tesis: Al pibe random que me mandó un mensaje por Instagram sin conocernos pero resultó ser Feli Marelli, que además de divertirnos en varias videollamadas durante la pandemia fue el que más me hizo tener ganas de hacer programas de inteligencia artificial que entiendan el ajedrez (y las ideas que habíamos charlado se terminaron transformando en el disparador de esta tesis). A Gabo Mindlin por ser el que más me contagió su pasión por los autoencoders en ese curso de la UMA, y por recibirme con tantas ganas en su laboratorio al principio del cuatri para poder hacerle preguntas y pedirle opiniones e ideas que fueron claves para poder hacer esta tesis. A Hernán Grecco por la buena onda que tiene desde el día que lo conocí, las ganas de juntarnos en la facu a charlar sobre física, y por hacerme ver que mi tesis tenía un gran valor que yo desconocía: según él son este tipo de proyectos de interpretabilidad los que van a hacer que varias herramientas de inteligencia artificial se terminen entendiendo mejor que nunca. A Sofi Roitman por tipear la primera letra de esta tesis. A Pablo Mislej

por coparse a ser jurado de esta tesis y a recibirme en su oficina un tiempo antes para conocernos y charlar de temas que nos copan a los dos.

Agradecimiento para Lichess por tener abierta y gratuita su base de datos, disponible para que los que queremos hacer proyectos que involucran partidas de ajedrez podamos hacerlos realidad.

Hay tantas personas a las que quiero mencionar en los agradecimientos que muy probablemente me esté olvidando de varios de ustedes sin querer, así que el último agradecimiento va para todos ustedes (pero si sos uno de ellos escribime y te mando el tuyo personalizado).

Índice general

1. INTRODUCCIÓN

Hace un tiempo conocí los autoencoders y desde el primer instante supe que quería hacer algún proyecto en el que estén invloucrados. Esta tesis es ese proyecto.

Mi introducción a los autoencoders fue el video [1], en el que los usaban para describir fotos de caras de personas en pocas variables. Me sorprendió que un método tan simple pueda lograr algo tan abstracto como representar una cara, no como una lista de píxeles y sus activaciones, sino con variables como largo del pelo, inclinación de la cabeza, género, etc.

Cuando pensamos en que la tesis fuera de la forma "entrenemos un autoencoder con algo en particular y veamos cómo aprende conceptos" no fue nada difícil decidirnos porque ese algo en particular fueran posiciones de ajedrez.

El objetivo principal de esta tesis es explorar cómo un modelo de autoencoder puede representar posiciones de ajedrez en un espacio latente de baja dimensión (estos conceptos, y otros que mencionemos en la introducción, están definidos en detalle más adelante), y qué tipo de información semántica logra capturar esta representación. Para eso, construimos y entrenamos un autoencoder sobre un conjunto de datos compuesto por posiciones reales extraídas de partidas jugadas en línea, y luego analizamos su comportamiento mediante distintas herramientas estadísticas y visuales.

Generamos el conjunto de datos a partir de una porción de la base pública de partidas de Lichess correspondiente a mayo de 2019. Filtramos exclusivamente las partidas estándar (ajedrez clásico) en las que ambos jugadores tuvieran al menos 2100 puntos de Elo y el ritmo de juego fuera de al menos 180 segundos por jugador. De estas partidas extrajimos posiciones intermedias, y representamos cada una como un vector binario mediante codificación *one-hot* de un bitboard extendido de 13 canales, uno por cada tipo de pieza (seis por color) más uno adicional para representar las casillas vacías. Cada posición quedó representada como un vector binario de dimensión 832 (13×64). Además, eliminamos posiciones duplicadas para evitar fuga de datos (data leakage) y dividimos los datos en conjuntos de entrenamiento y testeo.

Construimos un modelo de autoencoder entrenado para minimizar el error de reconstrucción entre la entrada y la salida, pasando en el medio por un cuello de botella de dimensión más baja (20 específicamente). Una vez entrenado el modelo, evaluamos la calidad de la reconstrucción tanto cuantitativamente usando el error cuadrático medio (ECM), como estructuralmente calculando el número de casillas mal reconstruidas en cada tablero, es decir, aquellas donde la pieza reconstruida no coincide con la original.

Posteriormente, analizamos la estructura del espacio latente aprendido. Para esto, aplicamos análisis de componentes principales (PCA) sobre las representaciones latentes de un conjunto de posiciones no vistas por el modelo, y visualizamos el espacio proyectándolo a planos generados por dos componentes principales. Primero analizamos qué proporción de la varianza explicaban las primeras componentes, para evaluar qué tan comprimible era el espacio latente. Luego, generamos visualizaciones coloreadas por distintos atributos semánticos de las posiciones: apertura, ubicación de los reyes, posición de peones centrales, entre otros. Esto nos permitió interpretar el significado de distintas direcciones del

espacio latente. Por ejemplo, observamos que la primera componente estaba fuertemente correlacionada con el estado de los enroques de los jugadores, y que otras componentes parecían estar relacionadas con la estructura de peones centrales.

También evaluamos el desempeño del modelo ante posiciones que diferían notablemente de las del conjunto de entrenamiento. Para ello, aplicamos el mismo pipeline de evaluación sobre un conjunto de posiciones jugadas por jugadores de Elo bajo (a lo sumo 1000) con poco tiempo en el reloj (a lo sumo 180 segundos). Comprobamos que el error de reconstrucción, tanto en términos de ECM como de número de casillas mal reconstruidas, era más alto en este conjunto, como era de esperar. Esto sugiere que el autoencoder no aprendió a respresentar posiciones guardando la información de la ubicación de cada pieza, sino captando regularidades propias de partidas de nivel intermedio o alto, que no siempre se respetan en partidas de jugadores principiantes.

Por último, exploramos la simetría "blancas-negras" del espacio latente evaluando la reconstrucción de posiciones reflejadas verticalmente, con colores de piezas intercambiados. Observamos que el error de reconstrucción de estas versiones reflejadas también es mayor que el de las posiciones originales, lo cual indica que, aunque simétricas desde un punto de vista geométrico, estas posiciones no son estadísticamente equivalentes en el corpus de entrenamiento. Esto nos proporcionó evidencia empírica de que la estadística de posiciones en partidas reales de ajedrez no es completamente simétrica bajo reflexiones verticales.

En resumen, este trabajo no sólo mostró que es posible representar posiciones de ajedrez de forma comprimida preservando buena parte de la información, sino que también evidenció que el espacio latente aprendido refleja propiedades semánticas del juego y responde de manera coherente ante cambios estructurales o contextuales. Esta tesis cae en la intersección entre aprendizaje no supervisado y juegos complejos, y propone herramientas para visualizar y entender el contenido de modelos entrenados en dominios ricos y estructurados como el ajedrez.

1.1. Organización de la tesis

En el desarrollo de esta tesis abordamos cada una de esas etapas. El trabajo se estructura de la siguiente manera:

En el capítulo 2 se introducen los conceptos necesarios para comprender el trabajo, incluídos autoencoders, análisis de componentes principales y nociones básicas de ajedrez. En el capítulo 3 se describe la metodología seguida, incluyendo la codificación númerica de posiciones de ajedrez utilizada, las herramientas de programación implementadas y el pipeline de desarrollo. En el capítulo 4 se presentan los resultados obtenidos, con visualizaciones e interpretabilidad sobre el espacio latente aprendido. Finalmente, en el capítulo 5 se discuten las conclusiones y posibles extensiones del trabajo.

2. PRELIMINARES

2.1. Autoencoders

Esta es una explicación informal e intuitiva sobre qué son los autoencoders a través de un ejemplo. Para una lectura más detallada ver [8].

Imaginemos que tenemos un conjunto de fotos de caras de personas. Una foto es algo que tiene mucha información: para cada píxel que contiene hay un valor de activación entre 0 (negro) y 1 (blanco). Pero si uno le pide a una persona que describa una foto de una cara, lo va a hacer muy bien diciendo muy pocas cosas (por ejemplo el largo del pelo, el género, la distancia entre los ojos, etc). ¿Se puede hacer un programa que aprenda a describir en pocas variables una foto de una cara? La respuesta es sí, y esta es una manera.

Hacemos una red neuronal que reciba como entrada un vector de $\mathbb{R}^m$ (en nuestro ejemplo, un vector formado por las activaciones de los m píxeles de una foto), que tenga una capa oculta de n neuronas con n mucho más chico que m, y que la salida también tenga tamaño m como la entrada. Es decir, esta red neuronal, haga lo que haga, recibe fotos como entrada y genera fotos en la salida, como muestra el esquema e la figra 2.1.

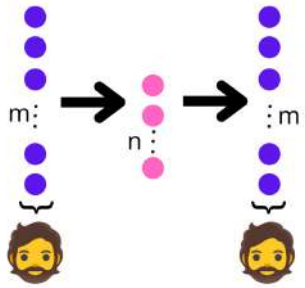

Fig. 2.1: Esquema de red neuronal que recibe y devuelve fotos.

¿Qué queremos que aprenda la red? Que cada vez que recibe en la entrada una foto de una cara genere en la salida exactamente la misma foto (o una lo más parecida posible, como se detalla más adelante). Esto se puede hacer con muchas fotos de caras como conjunto de entrenamiento (aprendizaje supervisado). La idea fundamental de hacer todo esto es que cuando la red neuronal lo aprende, inevitablemente está logrando comprimir toda la información de la foto de una cara en solo los n números de la capa del medio. Con esto conseguimos lo que queríamos, que era poder describir una foto entera de una cara con poca información.

Si llamamos $X = \mathbb{R}^m$ al espacio de entrada y de salida del autoencoder y $Z = \mathbb{R}^n$ al espacio correspondiente a la capa oculta del medio, entonces lo que estamos buscando son dos funciones, cada una de las cuales vive en una familia parametrizada: una función *codificadora* $E_\phi : X \to Z$ (parametrizada por el conjunto ϕ de coeficientes de una red neuronal) y una *decodificadora* $D_\theta : Z \to X$ (parametrizada por el conjunto θ de coeficientes de otra red neuronal). En este trabajo ambas familias de funciones van a ser redes neuronales con arquitecturas específicas. Si $x \in X$, a $z = E_\phi(x) \in Z$ lo llamamos la

codificación de x, y a $x' = D_\theta(z) \in X$ lo llamamos la *decodificación* de z. Al espacio Z lo llamamos *espacio latente*.

Cuando se entrena al autoencoder lo que se busca es que la salida sea idéntica a la entrada. Para esto, se busca que se minimice tanto como sea posible una función que mide la diferencia entre ellas, generalmente llamada *función de pérdida* o *error* . Una de las más comunes, que es la que vamos a usar en este trabajo es

$$L(\phi, \theta) = \frac{1}{N} \sum_{i=1}^{N} ECM(x^{(i)}, x'^{(i)}) \tag{2.1}$$

donde $\{x^{(1)}, ..., x^{(N)}\} \subset X$ es el conjunto de entrenamiento (y N su tamaño), $x'^{(i)} = D_\theta(E_\phi(x^{(i)}))$, y ECM es el error cuadrático medio, definido como

$$ECM(x, y) = \frac{1}{m} \sum_{j=1}^{m} (x_j - y_j)^2 \tag{2.2}$$

para $x, y \in \mathbb{R}^m$. Lo que buscamos es

$$\arg\min_{\phi, \theta} L(\phi, \theta) \tag{2.3}$$

La manera en la que estamos haciendo una reducción de dimensión es la siguiente. Nuestros datos, que pertenecen a $X = \mathbb{R}^m$, provienen de alguna distribución desconocida. Esta distribución puede tener correlaciones entre las distintas coordenadas, por lo cual es esperable que el autoencoder pueda lograr comprimir toda la información relevante en solamente las n coordenadas del espacio latente Z.

Teniendo esto en mente, ahora el autoencoder tiene otro uso: si nos olvidamos de la entrada y de la función codificadora y nos quedamos solo con el espacio latente, la función decodificadora y la salida, podemos a partir de puntos arbitrarios $z \in Z$ calcular sus imágenes en X a través de D_θ y conseguir así datos sintéticos que siguen la misma distribución que los reales (o lo más parecida posible, según qué tan buena sea la reconstrucción).

2.2. Interpretabilidad y Análisis de Componentes Principales (PCA)

El objetivo de esta tesis es hacer interpretabilidad en el espacio latente de un autoencoder que va a ser entrenado con posiciones de ajedrez (detalles en la sección 3), es decir, buscar entender el significado de distintas regiones del espacio en el que se codifican los datos.

Para analizar el espacio latente nos conviene aplicar la técnica de análisis de componentes principales (PCA), ver [6], [9] para una descripción técnica. Esencialmente, PCA calcula nuevos ejes en el espacio latente que son ortogonales y se ordenan de manera decreciente de acuerdo a la varianza de los datos que explican.

A continuación se describen los pasos necesarios para aplicar PCA a un conjunto de datos pertenecientes al espacio latente. Sea $A \in \mathbb{R}^{k \times n}$ una matriz de datos, donde cada una de las k filas es una muestra y cada una de las n columnas una variable del espacio latente.

1. **Estandarización de los datos**: centrar y escalar cada variable. Esto implica restar la media y dividir por el desvío estándar de cada columna:

$$\tilde{A}_{ij} = \frac{A_{ij} - \mu_j}{\sigma_j}$$

donde $\mu_j = \frac{1}{n}\sum_{i=1}^{n} A_{ij}$ y σ_j es el desvío estándar de la columna j.

2. **Cálculo de la matriz de covarianza**: una vez que los datos están centrados y escalados, se calcula la matriz de covarianza:

$$\Sigma = \frac{\tilde{A}^T \tilde{A}}{k-1}$$

Esta matriz captura las correlaciones lineales entre variables.

3. **Descomposición espectral (autovalores y autovectores)**: se calculan los autovalores $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq 0$ y sus correspondientes autovectores $v_1, v_2, \ldots, v_n$ de la matriz Σ. Los autovectores representan las *componentes principales* del espacio de datos y los autovalores indican cuánta varianza explica cada una.

4. **Selección de componentes**: se eligen algunos autovectores de los hallados en el paso anterior. Esto define una base ortonormal en un subespacio menor dimensión

5. **Proyección de los datos**: se proyectan todos los datos al subespacio de menor dimensión para obtener una descripción del espacio en menos variables.

En lo que sigue de esta tesis la selección de componentes va a ser hecha siempre con dos vectores, para poder visualizar fácilmente la información.

2.3. Ajedrez

Esta sección consiste en una explicación de los conceptos básicos de ajedrez necesarios para poder entender los detalles de esta tesis.

El ajedrez es un juego de estrategia de información completa en el que dos jugadores se enfrentan sobre un tablero de 8×8 casillas, alternando sus movimientos por turnos. Cada jugador controla un conjunto de dieciséis piezas, entre las cuales hay seis tipos distintos (ver figura 2.2). Las piezas son el rey, la dama (también llamada reina), dos torres, dos alfiles, dos caballos y ocho peones.

Fig. 2.2: Tipos de piezas de ajedrez. De izquierda a derecha: rey, dama, torre, alfil, caballo, peón.

Las piezas blancas se enfrentan a las piezas negras, y el jugador que controla las blancas mueve primero, partiendo siempre desde la misma posición inicial (ver figura 2.3).

Fig. 2.3: Tablero de ajedrez con las piezas en su configuración inicial.

Cada tipo de pieza tiene reglas específicas sobre cómo puede desplazarse por el tablero. Por ejemplo, los caballos se mueven en forma de "L", las torres en línea recta horizontal o vertical, los alfiles en diagonal, y la dama puede moverse en cualquier dirección (vertical, horizontal y diagonal). Los peones avanzan una casilla hacia adelante (o dos si es su primer movimiento y el jugador lo desea), y capturan en diagonal. Sólo se permite mover piezas propias, y en caso de que una pieza del adversario ocupe una casilla alcanzable, puede ser capturada al mover una pieza propia a esa casilla. (Ver figura 2.9). El ganador es el primero en capturar el rey del oponente.

Fig. 2.4: Peón

Fig. 2.5: Caballo

Fig. 2.6: Alfil

Fig. 2.7: Torre

Fig. 2.8: Dama

Fig. 2.9: Posición de ajedrez con los posibles movimientos de algunas piezas blancas (marcadas con fondo verde) y puntos que indican las posibles casillas de destino.

Una de las jugadas especiales del juego es el *enroque*, que consiste en un movimiento simultáneo del rey y una de las torres, en el que el rey se desplaza dos casillas hacia la torre y esta salta por encima del rey, colocándose justo al lado opuesto. El enroque está sujeto a varias condiciones, entre ellas, que no haya ninguna pieza ocupando alguna casilla que se encuentre en el camino entre el rey y la torre (ver figura 2.13).

Fig. 2.10: Blancas justo antes de enrocar.

Fig. 2.11: Blancas justo después de jugar enroque corto (lado derecho).

Fig. 2.12: Blancas justo después de jugar enroque largo (lado izquierdo).

Fig. 2.13: Ejemplo de enroque.

Para describir las posiciones en ajedrez, se utiliza una notación estandarizada conocida como notación algebraica. En ella, cada casilla del tablero se identifica mediante una letra (de la `a` a la `h`) que indica la columna, siendo `a` la de la izquierda desde el punto de vista del jugador blanco, y un número (del 1 al 8) que indica la fila. Por convención, las blancas ocupan inicialmente las filas 1 y 2, y las negras las filas 7 y 8. Para indicar qué pieza ocupa una casilla, se antepone una letra mayúscula que representa la inicial del tipo de pieza (`K` para rey, `Q` para dama, `R` para torre, `B` para alfil, `N` para caballo y `P` para peón). Si la pieza es negra, se respresenta con la misma letra pero en minúscula. Por ejemplo, `Nf3` indica que hay un caballo blanco en la casilla `f3`, mientras que `nf3` indica que hay un caballo negro en esa misma posición. La notación usada para describir jugadas en una partida es similar.

Al comienzo de una partida, existen patrones de movimientos frecuentes que se han estudiado extensamente y que reciben el nombre de *aperturas* o *defensas* (ver ejemplos en la figura 2.18). Estas secuencias iniciales buscan posiciones convenientes para el juego, en las que se persiguen conceptos estratégicos como el desarrollo eficiente de las piezas, el control del centro del tablero o la preparación el enroque. Se las suele clasificar en *abiertas*, *semiabiertas*, *cerradas* y *semicerradas* según el movimiento de los peones centrales, que son los que comienzan en las columnas `d` y `e`.

Fig. 2.14: Ruy López: `e4`, `e5`, `Nf3`, `Nc6`, `Bb5`.

Fig. 2.15: Siciliana: `e4`, `c5`.

Fig. 2.16: Defensa francesa: `e4`, `e6`.

Fig. 2.17: India de rey: `d4`, `Nf6`, `c4`, `g6`.

Fig. 2.18: Ejemplos de aperturas.

A lo largo de esta tesis se trabajará con partidas extraídas de la base de datos pública de Lichess, un servidor de ajedrez libre y popular. Las partidas allí se registran en formato PGN (Portable Game Notation), una notación de texto que incluye entre otras cosas el nombre de los jugadores, el tiempo de juego, la fecha, el resultado, y la secuencia completa de jugadas. A continuación se muestra un ejemplo de partida extraída de dicha base de datos:

```
[Event "casual blitz game"]
[Site "https://lichess.org/HMXmrrQe"]
[Date "2024.12.31"]
[Round "-"]
[White "teoccc"]
[Black "JuliGarbulsky"]
[Result "1-0"]
[GameId "HMXmrrQe"]
[UTCDate "2024.12.31"]
[UTCTime "23:30:45"]
[WhiteElo "1592"]
[BlackElo "1487"]
[Variant "Standard"]
[TimeControl "300+0"]
[ECO "C62"]
[Opening "Ruy Lopez: Steinitz Defense"]
[Termination "Time forfeit"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 d6 4. d4 exd4 5. Nxd4 Bd7 6. Nxc6 Bxc6
7. Bxc6+ bxc6 8. O-O Rb8 9. Nc3 g6 10. b3 Bg7 11. Bb2 Nf6 12. Re1 Nd7
13. Qf3 Ne5 14. Qe2 O-O 15. f4 Nd7 16. e5 Re8 17. Qc4 Rb6
18. exd6 cxd6 19. Rxe8+ Qxe8 20. Rf1 Qe3+ 21. Kh1 Qc5 22. Qxc5 Nxc5
23. Nd1 Rb7 24. Re1 Bxb2 25. Nxb2 f5 26. Re8+ Kf7 27. Rh8 Kg7
28. Re8 Kf7 29. Re2 Re7 30. Rxe7+ Kxe7 31. g3 Ne4 32. c4 d5
33. cxd5 cxd5 34. Kg2 Kd6 35. Kf3 Kc5 36. Nd3+ Kd4 37. Ne5 Kc5
38. Ke3 d4+ 39. Kd3 Kd5 40. Nc4 Nf2+ 41. Ke2 Ng4 42. h3 Ne3
43. Nxe3+ dxe3 44. Kxe3 a5 45. g4 h6 46. gxf5 gxf5 47. Kd3 Ke6
48. Kc4 Kd6 49. b4 Kc6 50. b5+ Kb7 51. a4 Ka7 52. Kc5 Kb7 53. b6 h5
54. h4 Kb8 55. Kc6 Ka8 56. Kb5 Kb8 57. Kxa5 Kc8 58. Ka6 Kb8 59. a5 Ka8
60. b7+ Kb8 61. Kb5 Kc7 62. a6 Kb8 63. Kc6 Ka7 64. Kc7 1-0
```

Cada línea con corchetes contiene metadatos de la partida, y la secuencia de jugadas aparece luego, indicando qué pieza se mueve y a qué casilla.

Para este trabajo los metadatos relevantes van a ser WhiteElo y BlackElo (los puntajes que describen los niveles de habilidad de los jugadores [3]), TimeControl (que contiene la

información del tiempo total con el que cuentan los jugadores para la partida) y Opening (que contiene el nombre correspondiente a la apertura o defensa jugada).

3. METODOLOGÍA

3.1. Codificaciones de tableros de ajedrez

Dado que queremos hacer que la entrada y la salida de nuestra red neuronal sean posiciones de ajedrez, el primer paso es pensar cómo codificar una posición de ajedrez como un vector de números. Existen múltiples maneras de hacerlo, pero en este trabajo nos enfocamos en la más usada, llamada *bitboard* (ver por ejemplo [12], [4], [5], [11]), y le hacemos modificaciones para conseguir otra que llamamos *one hot B* (definida más adelante). Para otras posibles codificaciones ver [7], [10]. Aclaración: no confundir el término "codificación" refiriéndose a la manera de representar una posición como un vector de números con el mismo término usado para hablar de dónde cae en el espacio latente un punto a través de la función codificadora (sección 2.1). La ambigüedad está resuelta por el contexto en el que se usa la palabra.

La primera codificación, ampliamente utilizada en trabajos relacionados con aprendizaje automático aplicado al ajedrez, es la conocida como *bitboard*. Esta representación consiste en construir un vector de longitud $12 \times 64 = 768$, resultado de concatenar 12 vectores de 64 entradas cada uno. Cada uno de esos 12 vectores corresponde a uno de los 12 tipos posibles de pieza (6 tipos por cada color: rey, dama, torre, alfil, caballo y peón). La i-ésima coordenada de uno de estos vectores vale 1 si en la casilla i del tablero se encuentra el tipo de pieza correspondiente al canal en cuestión, y 0 en caso contrario. Así, por ejemplo, si hay un caballo blanco en la casilla `f3`, el canal correspondiente a los caballos blancos tendrá un 1 en la posición `f3`, y el resto de los canales tendrá un 0 en esa posición. Esta representación es *esparsa*, ya que un tablero puede contener como máximo 32 piezas, por lo que a lo sumo 32 de las 768 coordenadas son iguales a 1; el resto son ceros. También es *redundante*, ya que si una casilla contiene una pieza, entonces necesariamente todos los otros canales tienen un 0 en esa posición.

Una limitación de esta codificación surge al utilizarla como *salida* de una red neuronal. Dado que las activaciones de las neuronas suelen tomar valores reales entre 0 y 1, hay que diseñar un procedimiento para interpretar el resultado como una posición de ajedrez. Una manera posible es definir qué pieza que hay en la i-ésima casilla como el canal que tiene mayor activación en ella, pero esto daría lugar a un tablero que tiene piezas en todas sus casillas (no tiene ninguna vacía). Esto último se puede solucionar definiendo un umbral que una activación debe superar para que la casilla no se considere vacía (Ejemplo en la figura 3.4).

Fig. 3.1: Umbral alto.

Fig. 3.2: Umbral medio.

Fig. 3.3: Umbral bajo.

Fig. 3.4: Posición generada por una salida hipotética de un autoencoder, reconstruida para distintos umbrales.

Si bien ese procedimiento cumple lo que necesitamos, tiene un umbral arbitrario, que su valor puede cambiar en la reconstrucción de una posición.

Por este motivo decidimos utilizar una codificación alternativa, que denominamos *one hot B* [1], y que se basa en una variante directa del bitboard. En lugar de tener 12 canales (uno por tipo de pieza), agregamos un canal adicional que indica si una casilla está vacía. Así, el tablero se representa con $13 \times 64 = 832$ coordenadas. Al igual que antes, cada uno de los 13 vectores de 64 posiciones representa un canal asociado a un tipo de pieza o al estado "vacío". Para cada casilla del tablero, exactamente una de las 13 coordenadas correspondientes vale 1 y las demás valen 0, indicando el contenido de la casilla. Es decir, cada casilla del tablero se representa como un vector *one hot* de dimensión 13. Esta representación, aunque también redundante (ya que el canal de casillas vacías puede inferirse a partir de los demás), tiene la ventaja de permitir una interpretación directa de la salida de la red neuronal: basta con tomar, para cada casilla, el canal con mayor activación y asignar a esa casilla el contenido correspondiente. Este mecanismo evita el problema del umbral y garantiza que cada casilla tenga una única interpretación.

Es importante mencionar que en algunos trabajos la representación *bitboard* también es llamada *one hot* (ver por ejemplo [12]), lo cual puede llevar a confusión. En este trabajo vamos a usar solamente *one hot B*, refiriéndonos siempre a la codificación que utiliza 13 canales y garantiza que cada casilla esté representada por un único 1 entre 13 opciones posibles.

A modo de ejemplo, en la figura 3.5 se puede observar la representación *one hot B* de una posición.

[1] La llamo *one hot Bergerman*, que es el apellido de mi amigo al que se le ocurrió esta codificación mientras comíamos una pizza (gracias Mati!).

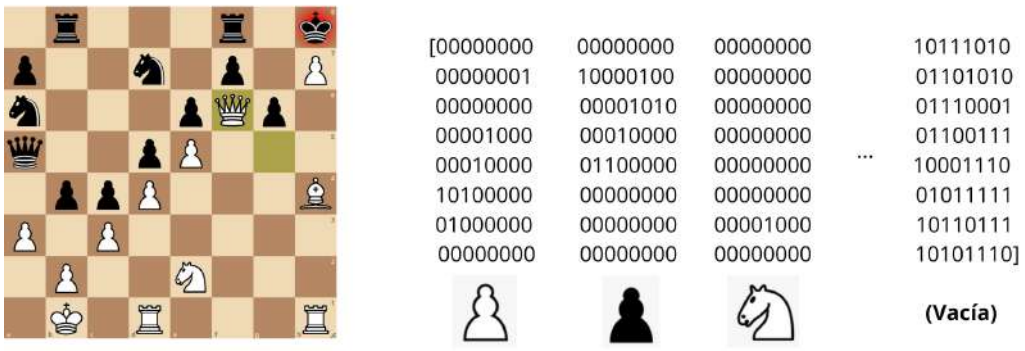

Fig. 3.5: Ejemplo de una posición de ajedrez y su codificación *one hot B*.

A modo de observación, este manera de reconstruir posiciones a partir de un vector numérico podría dar lugar a tableros con piezas que formen una posición que no sea válida en el ajedrez (por ejemplo, una que tenga dos reyes blancos).

Esta elección de codificación fue fundamental para poder reconstruir tableros de manera coherente a partir de la salida del autoencoder, y constituye uno de los aspectos clave del diseño metodológico de este trabajo.

3.2. Herramientas

Para el desarrollo de este trabajo usamos un conjunto de herramientas de software que permitieron la implementación, entrenamiento y análisis de modelos de autoencoders aplicados a posiciones de ajedrez. Estas fueron las principales:

- Lenguaje de programación: Python, en el IDE Spyder de Anaconda. Específicamente: Python 3.12.2 64-bit — Qt 5.15.2 — PyQt5 5.15.10 — macOS 13.4 (arm64), y Spyder 5.5.1. Elegimos Python por su amplia adopción en la comunidad científica y de aprendizaje automático, y por su ecosistema de librerías específicas para cada etapa del trabajo.
- Manipulación de datos: NumPy versión 1.26.4. NumPy fue fundamental para representar y transformar datos en vectores y matrices, incluyendo las codificaciones de los tableros.
- Ajedrez: python-chess (Versión 1.11.2). La librería python-chess la usamos para manipular posiciones en notación FEN, generar tableros y convertir entre representaciones del juego y estructuras numéricas. Fue central para implementar las funciones de codificación y decodificación de posiciones.
- Redes neuronales: TensorFlow (2.19.0) y Keras. Para definir, entrenar y evaluar los autoencoders usamos TensorFlow junto con su interfaz de alto nivel Keras. Keras

permitió una implementación clara y modular de las redes, funciones de activación y optimización, y métricas de reconstrucción.

- Visualización: Matplotlib (versión 3.9.2). La usamos para generar los gráficos.
- Conjunto de datos: Lichess. Usamos datos públicos de partidas provenientes de la base de datos de Lichess [2] bajo la licencia Creative Commons CC0 license. Las partidas las filtramos según criterios de tiempo y nivel de los jugadores (ver Sección 3.2.1), para su posterior codificación.

Estas herramientas nos permitieron implementar de forma eficiente un pipeline completo, desde la preparación de datos hasta la visualización de resultados, en un entorno abierto, reproducible y de bajo costo computacional.

3.2.1. Pipeline de desarrollo

En esta sección describimos el pipeline completo de desarrollo, dividido en etapas: recolección y preprocesamiento de datos, codificación de las posiciones, arquitectura del autoencoder y entrenamiento del modelo.

Recolección y filtrado de datos

Los datos fueron descargados de la base de datos abierta de Lichess, específicamente del archivo que contiene todas las partidas jugadas durante mayo de 2019. Este archivo contiene más de 35 millones de partidas, muchas de las cuales no son útiles para nuestro análisis: algunas fueron jugadas por principiantes (jugadores de bajo Elo) y otras con poco tiempo en el reloj (partidas tipo *bullet*). Este tipo de partidas suele tener "ruido" y no tantos patrones propios del ajedrez, por ejemplo, errores groseros por apuros de tiempo o falta de experiencia.

Para enfocarnos en partidas de mayor calidad, filtramos el dataset conservando solo aquellas en las que ambos jugadores tienen un Elo de al menos 2100 y donde el tiempo inicial en el reloj es de al menos 180 segundos por jugador. Luego del filtrado, nos quedamos con aproximadamente medio millón de partidas.

Selección de posiciones

Cada partida de ajedrez contiene varias posiciones (una por cada jugada), pero decidimos seleccionar solo una por partida para el entrenamiento del modelo. Elegimos la posición resultante luego de que ambos jugadores hayan realizado 10 jugadas (es decir, tras 20 movimientos en total), siempre que la partida no haya terminado antes. Este punto de la partida representa un buen equilibrio: por un lado, ya se hicieron suficientes jugadas como para sea poco probable que la posición también haya sido alcanzada en otra partida de la base de datos; y por otro lado, la distribución de las piezas todavía es más organizada que en etapas más avanzadas del juego.

De este modo, seleccionamos 100.000 posiciones. Luego, eliminamos las posiciones repetidas para evitar *data leakage* entre los conjuntos de entrenamiento y validación. Esto nos dejó un total cercano a 90.000 posiciones únicas.

Codificación de las posiciones

Cada posición fue representada como un vector usando la codificación que vamos a llamar*one hot B*, (ver sección 3.1) debido a su simplicidad y capacidad de reconstrucción de posiciones a partir de vectores con valores no discretos.

Arquitectura del autoencoder

La arquitectura del autoencoder fue diseñada como una red neuronal densa y simétrica respecto de la capa central. Fijadas las dimensiones de la entrada y de la salida (que coinciden entre ellas y a su vez con la del vector de la codificación) y la cantidad de neuronas en la capa latente, construimos el codificador colocando capas ocultas cuyos tamaños se reducen progresivamente, a la mitad del anterior, hasta alcanzar el de la dimensión latente. Luego, el decodificador repite esta estructura en orden inverso. Por ejemplo, para pasar de 832 valores en la entrada a 20 neuronas en la capa latente, usamos capas ocultas de tamaños 416, 208, 104, 52, y 26.

Todas las capas son densas (totalmente conectadas), con función de activación ReLU, excepto la última capa del codificador y la última del decodificador, donde se utiliza la función sigmoide. Esto permite obtener salidas acotadas entre 0 y 1, lo cual es deseable dado que las entradas están compuestas sólo por ceros y unos, y que puede resultar conveniente que el espacio latente sea acotado.

Entrenamiento del modelo

El conjunto de 90.000 posiciones fue dividido aleatoriamente en un conjunto de entrenamiento (85 %) y uno de validación (15 %). Aunque el autoencoder se entrena de forma supervisada (entrada y salida son iguales), el objetivo es que aprenda una representación latente significativa de los datos (no supervisada).

El entrenamiento se realizó con Keras, usando el optimizador Adam, un tamaño de batch de 256 y la métrica de error cuadrático medio. Para elegir la cantidad de épocas de entrenamiento, analizamos la evolución del error en función de la cantidad de *epochs*, utilizando como referencia un autoencoder con codificación *one hot B* y 20 neuronas latentes. En la figura 3.6 se observa que el error en el conjunto de validación se estabiliza alrededor de las 125 épocas, por lo que esta cantidad se adoptó como valor fijo para todos los entrenamientos posteriores.

Elección de la dimensión latente

Con el pipeline definido, exploramos distintos valores posibles para la cantidad de neuronas en la capa latente. Existe un compromiso entre fidelidad de reconstrucción y capacidad de compresión: pocas neuronas pueden no ser suficientes para representar la información, mientras que muchas permiten reconstruir bien las entradas pero sin lograr una reducción significativa de la dimensionalidad.

Para evaluar este balance, entrenamos modelos con diferentes tamaños de la capa latente y comparamos su error de reconstrucción (ver figura 3.7). Realizamos esta comparación para la codificación *one hot B*. En función de los resultados, seleccionamos una dimensión latente de 20 neuronas como compromiso adecuado.

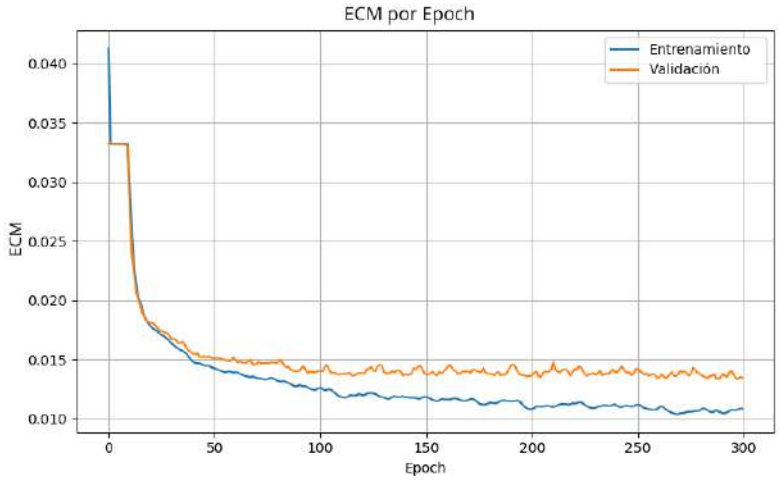

Fig. 3.6: Error (ECM) en el conjunto de entrenamiento y de validación del autoencoder en función de la cantidad de épocas.

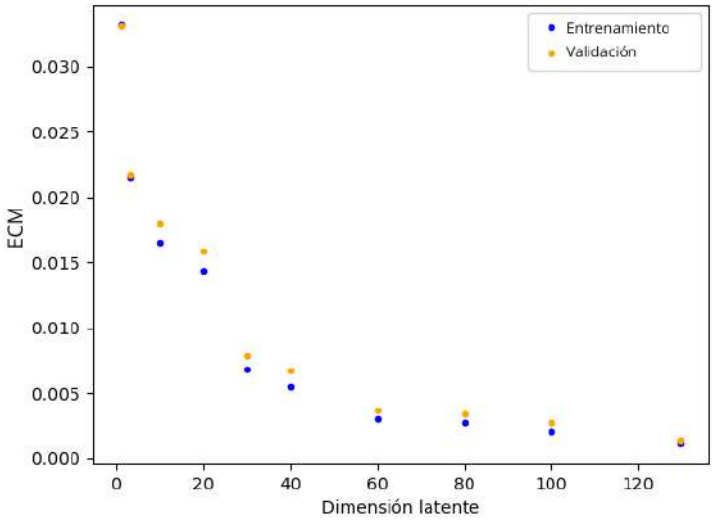

Fig. 3.7: Error (ECM) en conjuntos de entrenamiento y de validación del autoencoder en función de la cantidad de neuronas latentes.

Evaluación final

Finalmente, evaluamos el modelo ya entrenado sobre un conjunto de testeo independiente (no utilizado ni en entrenamiento ni en validación), proveniente de las 500 mil posiciones ya filtradas por calidad de juego. Este conjunto contiene aproximadamente 80.000 posiciones y nos permite estimar la capacidad real del autoencoder para generalizar a datos nuevos. Como métricas, utilizamos por un lado, nuevamente el error cuadrático medio, y por otro lado, la fracción de casillas del tablero mal reconstruidas (ver sección 4.1).

4. RESULTADOS

En este capítulo presentamos los resultados obtenidos con el autoencoder entrenado para representar posiciones de ajedrez. Dividimos los análisis en tres secciones: primero evaluamos la capacidad de reconstrucción del modelo, luego exploramos la estructura del espacio latente en busca de interpretabilidad, y finalmente analizamos el desempeño del modelo ante posiciones que se alejan de la distribución del entrenamiento.

4.1. Métricas de reconstrucción

Evaluamos la calidad de las reconstrucciones utilizando dos métricas distintas sobre el conjunto de testeo:

- Error cuadrático medio (ECM), ver ecuación (2.2).

- Número de casillas mal reconstruidas (NCMR): dada una salida del autoencoder, reconstruimos la posición de ajedrez correspondiente con el procediminto definido en la sección 3.1. Consideramos que una casilla está mal reconstruida si su estado no coincide con el de la posición original. Calculamos entonces el número de casillas mal reconstruidas sobre un conjunto de posiciones y tomamos el promedio. Se pueden ver ejemplos de esta métrica en la figura 4.7.

Ambas métricas reflejan distintas formas de error: mientras que el ECM captura errores pequeños en la representación numérica, la segunda métrica es más estricta desde el punto de vista ajedrecístico y da una mejor intuición de qué tan bien reconstruida está la posición.

Los resultados obtenidos en el conjunto de testeo fueron:

- ECM: 0.01466.

- NCMR: 7.55 (calculado con 1.000 posiciones de las del conjunto de testeo, para mayor velocidad computacional).

Fig. 4.1: Posición original 1.

Fig. 4.2: Posición original 2.

Fig. 4.3: Posición original 3.

Fig. 4.4: Posición reconstruida 1 (8 casillas incorrectas).

Fig. 4.5: Posición reconstruida 2 (3 casillas incorrectas).

Fig. 4.6: Posición reconstruida 3 (16 casillas incorrectas).

Fig. 4.7: Ejemplos de posiciones de ajedrez originales y sus respectivas reconstrucciones.

4.2. Estructura del espacio latente

Para estudiar la estructura interna del espacio latente generado por el encoder, aplicamos Análisis de Componentes Principales sobre las codificaciones del conjunto de testeo. Esta técnica permite proyectar los vectores latentes en componentes ortogonales ordenadas según la varianza que explican.

El gráfico de varianza acumulada indica que una fracción importante de la varianza total puede explicarse con pocas componentes principales (figura 4.8). Las primeras tres componentes explican el 52 % de la varianza total, y las primeras nueve el 94 %.

Lo que sigue es hacer análisis de interpretabilidad para buscar si las componentes principales tienen significados específicos para las posiciones de ajedrez que describen.

4.2.1. Distribución por aperturas

Proyectamos las posiciones sobre las dos primeras componentes principales y coloreamos los puntos según la apertura jugada en la partida a la que pertenece cada una. La información de la apertura jugada está incluida en la base de datos de Lichess.

Observamos una cierta separación entre aperturas, lo que sugiere que el modelo ha captado aspectos distintivos del tipo de juego generado por distintas líneas de apertura (figura 4.9).

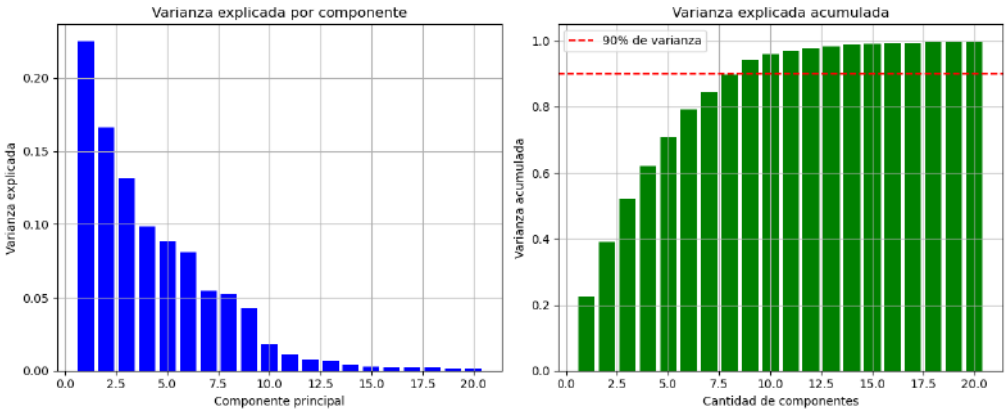

Fig. 4.8: Gráficos de varianza explicada por cada componente principal y de varianza total acumulada.

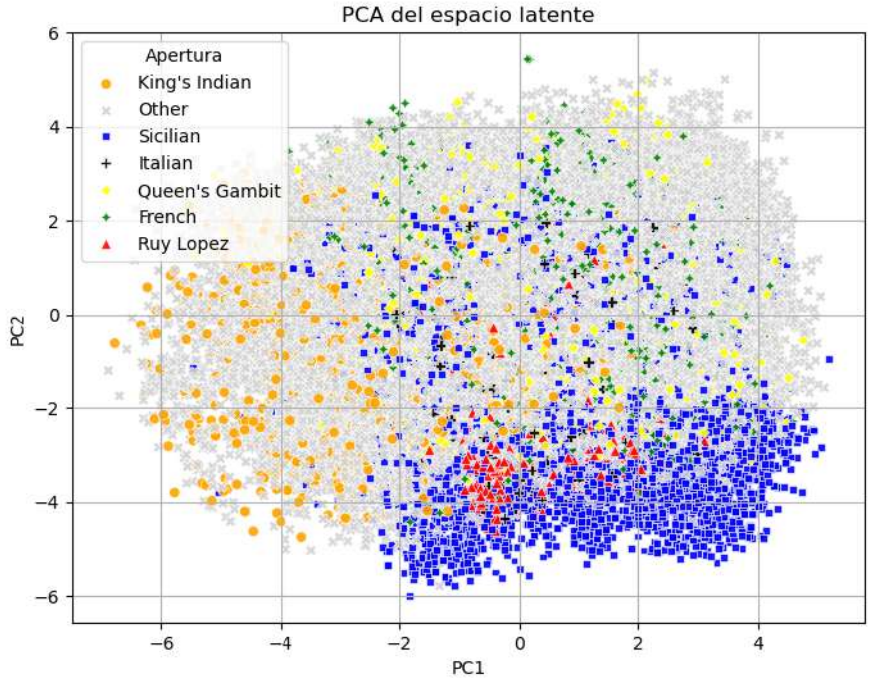

Fig. 4.9: Primeras dos componentes principales de las posiciones codificadas en el espacio latente, coloreando por la apertura jugada en sus correspondientes partidas.

También proyectamos al plano generado por la primera y tercera componente principal (figura 4.10), y al plano generado por la segunda y tercera (figura 4.11).

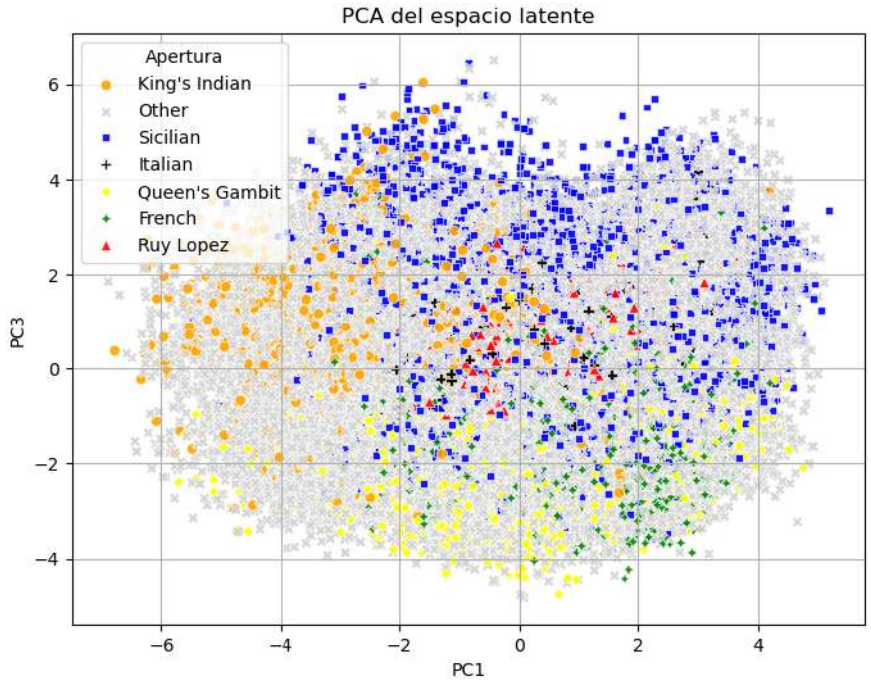

Fig. 4.10: Primera y tercera componente principal de las posiciones codificadas en el espacio latente, coloreando por la apertura jugada en sus correspondientes partidas.

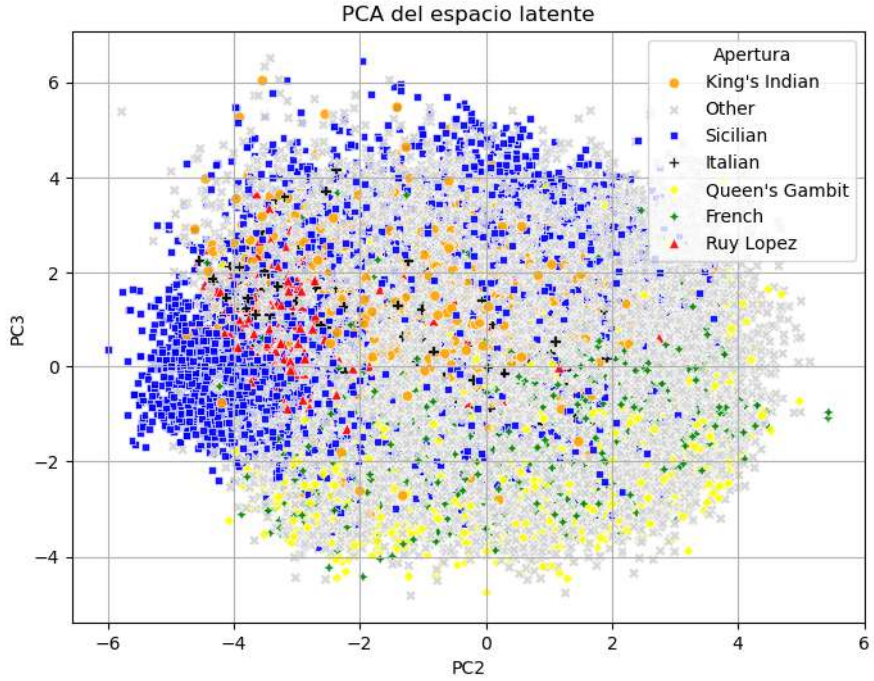

Fig. 4.11: Segunda y tercera componentes principales de las posiciones codificadas en el espacio latente, coloreando por la apertura jugada en sus correspondientes partidas.

4.2.2. Significado de las componentes

Exploramos visualmente el significado de las componentes principales coloreando las posiciones según la presencia de determinadas piezas o estructuras.

Primera componente (PC1). Parece estar relacionada con el estado de los enroques. Las posiciones suelen caen en regiones distintas según si los reyes están o no enrocados (figuras 4.12 y 4.13). Mirando ambos gráficos se puede ver que esta componente tiene regiones especializadas en las cuatro combinaciones posibles de blancas y negras enrocadas y no enrocadas (con enroque corto, que es el que se obtiene cuando los reyes dejan de estar en su columna inicial que es la `e`, y pasan a estar en la columna `g`).

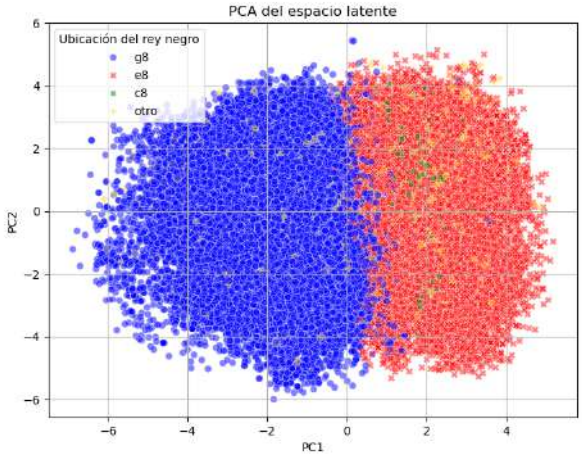

Fig. 4.12: Primeras dos componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el rey negro

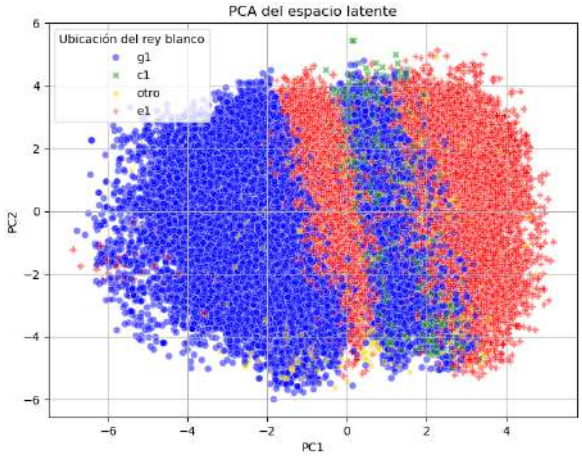

Fig. 4.13: Primeras dos componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el rey blanco

Segunda y tercera componente. Estas componentes capturan características relacionadas con la estructura de peones centrales, es decir, los peones blancos y negros de las columnas `d` y `e`. En las figuras 4.14, 4.15, 4.16 se puede ver el efecto que tienen estas dos componentes en la ubicación del peón negro de la columna `e`, mientras que el de la componente principal es casi nulo. Coloreamos cada punto según dónde está el peón en la posición correspondiente, y la categoría"otro" significa que en las tres casillas analizadas hay o bien más de un peón del color en cuestión o bien ninguno. Algo similar ocurre con los otros tres peones centrales, con la excepción de que para el peón blanco de la columna `e` sí parece ser relevante la primera componente principal.

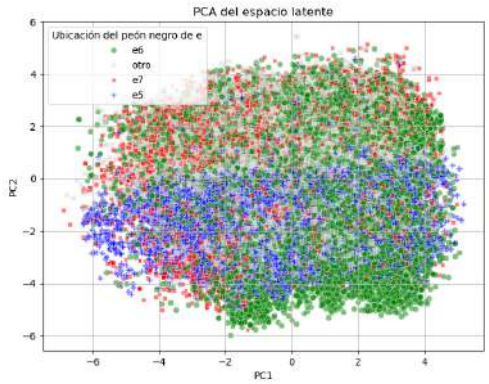

Fig. 4.14: Primeras dos componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón negro de la columna e.

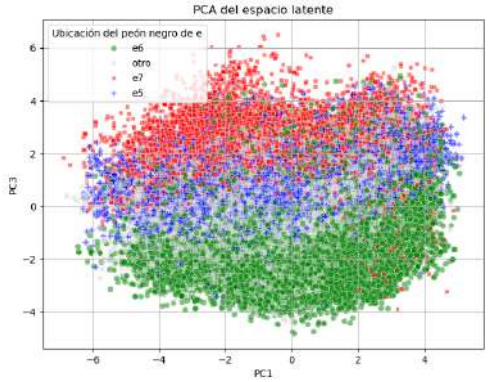

Fig. 4.15: Primera y tercera componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón negro de la columna e.

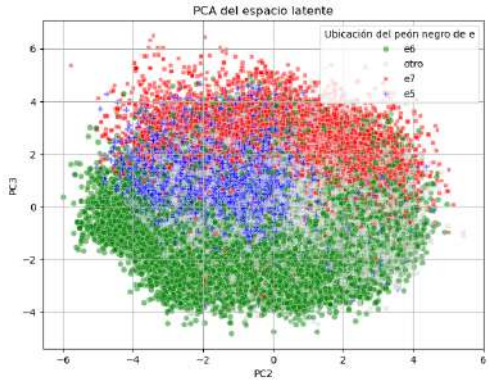

Fig. 4.16: Segunda y tercera componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón negro de la columna e.

Peón negro de la columna `d` (figuras 4.17, 4.18 y 4.19):

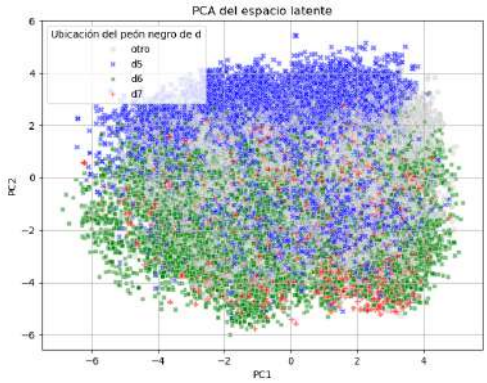

Fig. 4.17: Primeras dos componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón negro de la columna d.

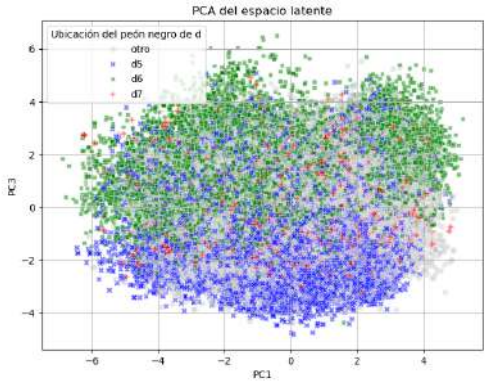

Fig. 4.18: Primera y tercera componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón negro de la columna d.

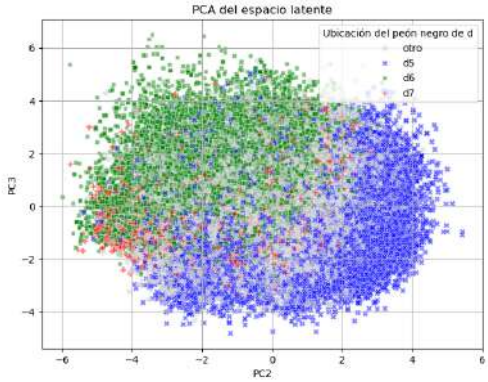

Fig. 4.19: Segunda y tercera componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón negro de la columna d.

Peón blanco de `e` (figuras 4.20, 4.21 y 4.22):

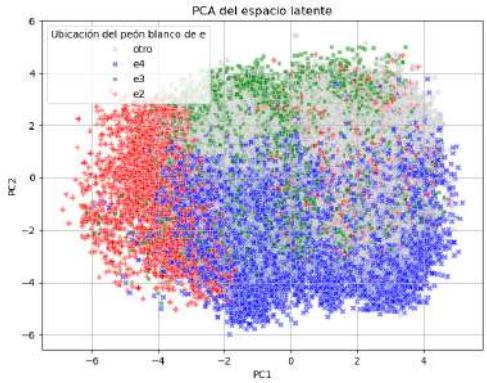

Fig. 4.20: Primeras dos componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón blanco de la columna e.

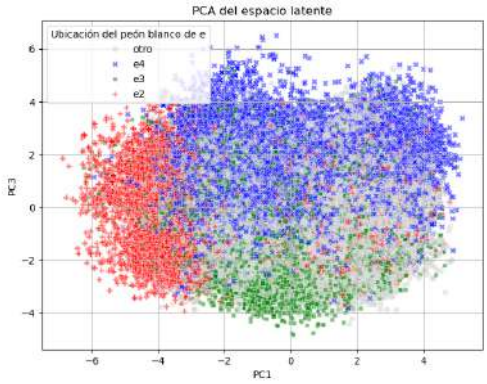

Fig. 4.21: Primera y tercera componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón blanco de la columna e.

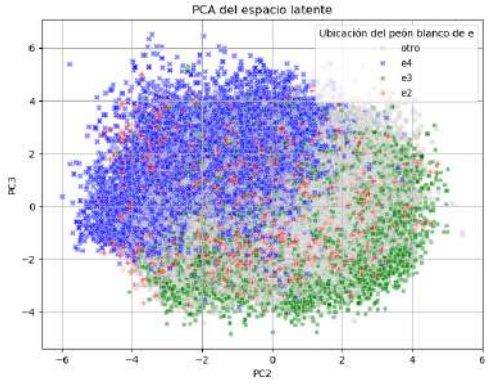

Fig. 4.22: Segunda y tercera componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón blanco de la columna e.

Peón blanco de `d` (figuras 4.23, 4.24 y 4.25):

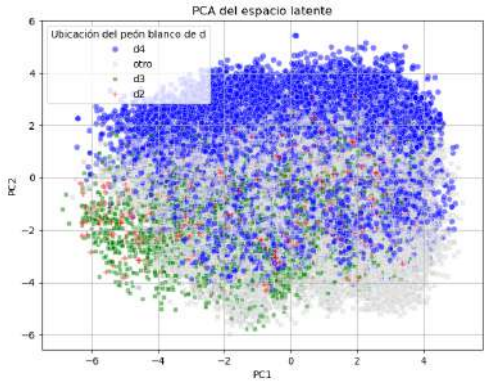

Fig. 4.23: Primeras dos componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón blanco de la columna d.

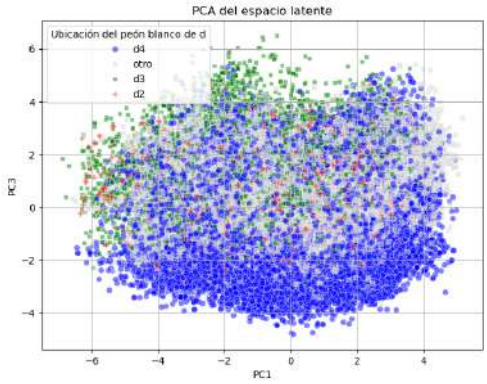

Fig. 4.24: Primera y tercera componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón blanco de la columna d.

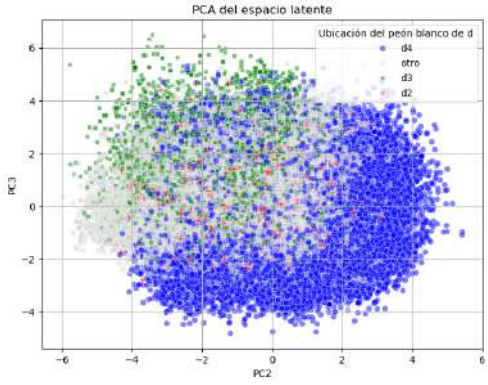

Fig. 4.25: Segunda y tercera componentes principales de las posiciones codificadas en el espacio latente, coloreando según la casilla en la que está el peón blanco de la columna d.

Séptima componente. Se asocia con la presencia de un caballo blanco en la casilla f3 (casilla muy común para un caballo blanco durante la apertura de una partida). Ver figura 4.26.

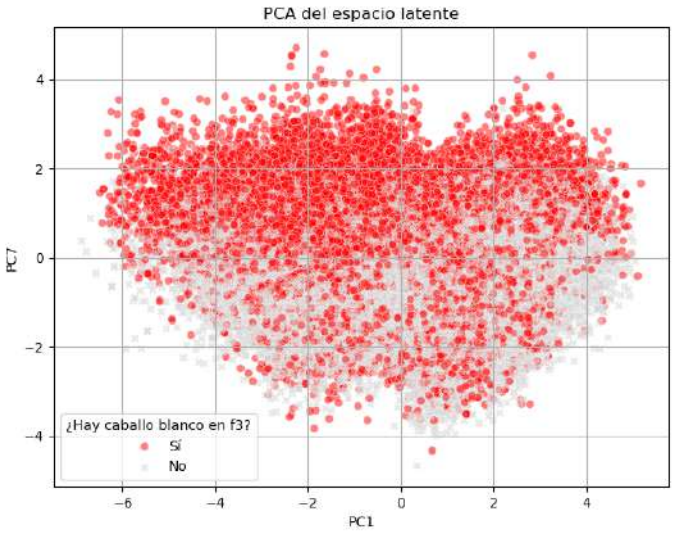

Fig. 4.26: Primera y séptima componentes principales de las posiciones codificadas en el espacio latente, coloreando según si hay o no un caballo blanco en la casilla f3.

4.3. Evaluación en posiciones fuera de distribución

Una pregunta crítica que podemos hacernos es cuál es la manera en la que el autoencoder logra reducir la dimensión de posiciones de ajedrez: ¿aprendió de patrones comunes del juego o solamente está guardando de alguna manera la información de la ubicación de cada pieza?

Para responder a esta pregunta, realizamos experimentos para ver cómo responde el autoencoder ante datos que siguen una distribución distinta a la del conjunto de entrenamiento.

4.3.1. Partidas de jugadores con bajo Elo

Recolectamos posiciones provenientes de partidas entre jugadores con Elo menor o igual a 1000 y con un control de tiempo de a lo sumo 180 segundos. Estas partidas están considerablemente fuera de la distribución de las usadas en el entrenamiento. En la figura 4.33 se pueden ver algunas de estas posiciones y sus respectivas reconstrucciones.

Las métricas de reconstrucción en este caso fueron:

- ECM: 0.02090.
- NCMR: 10.97 (calculado con 1.000 posiciones).

Como se puede ver, el modelo reconstruye peor estas posiciones que las del conjunto de testeo proveniente de la misma distribución que el de entrenamiento (el ECM aumentó

en un 43 % y el NCMR en un 45 %). Esto sugiere que el autoencoder aprendió una representación más ajustada al estilo de juego de partidas de alto nivel, y no específicamente la información de la ubicación de cada pieza en el tablero.

Fig. 4.27: Posición original 1.

Fig. 4.28: Posición original 2.

Fig. 4.29: Posición original 3.

Fig. 4.30: Posición reconstruida 1 (14 casillas incorrectas).

Fig. 4.31: Posición reconstruida 2 (9 casillas incorrectas).

Fig. 4.32: Posición reconstruida 3 (19 casillas incorrectas).

Fig. 4.33: Ejemplos de posiciones originales de ajedrez de Elo bajo (≤ 1000) y poco tiempo en el reloj (≤ 180 segundos) y sus respectivas reconstrucciones.

4.3.2. Posiciones reflejadas verticalmente

También analizamos el desempeño del modelo ante posiciones generadas artificialmente mediante un proceso de reflexión vertical del tablero e intercambio de colores (ver figura 4.36). La idea es que una posición de blancas se convierte en una posición de negras (y viceversa) con las piezas en simetría vertical.

Para este tipo de posiciones, usando las mismas dos métricas obtuvimos

- ECM: 0.01747, que fue calculado con posiciones generadas intercambiando colores y reflejando verticalmente las del conjunto de testeo original (respecto del cual aumentó 19 %). Para ver que la diferencia es significativa (ver figura 4.37), realizamos un test a nivel aproximado α la hipótesis $H_0 : \mu \leq \mu_0$ (donde μ la esperanza del ECM para este tipo de posiciones y μ_0 la de las originales). Suponemos que el número de muestras de cada distribución (N=83595) es suficientemente grande, y usamos como estadístico $T = \sqrt{N}\frac{\bar{X}-\mu_0}{s}$, donde X es el conjunto de muestras de ECMs de posiciones reflejadas

Fig. 4.34: Posición de ajedrez.

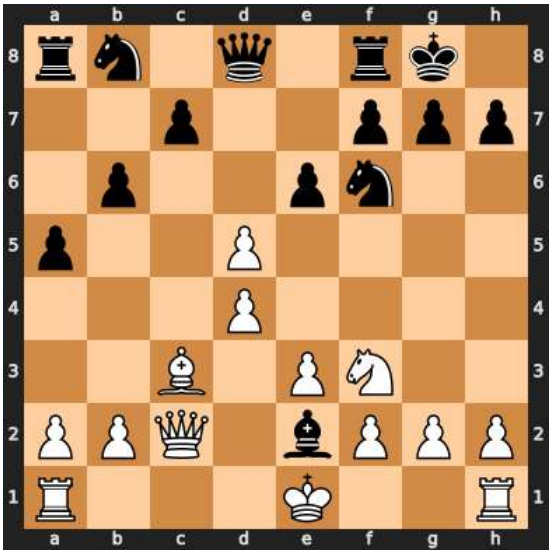

Fig. 4.35: Posición de ajedrez reflejada verticalmente y con colores invertidos.

Fig. 4.36: Ejemplo de una posición y la que resulta de reflejarla verticalmente e invertir colores.

verticalmente, $\bar{X}$ su promedio, y s el desvío estándar muestral de los ECM de las posiciones originales. Obtuvimos $T = 129,31$, por lo que rechazamos la hipótesis.

- NCMR: 9.17, que fue calculado con posiciones generadas con el mismo proceso de recién a partir del conjunto de las 1.000 originales para esta métrica (respecto del cual aumentó 21 %).

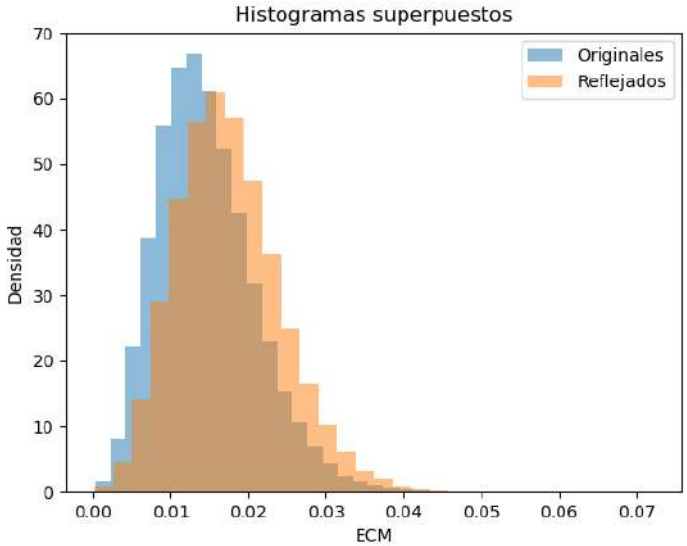

Fig. 4.37: Histogramas de las distribuciones de ECM de ambos conjuntos de posiciones.

Este resultado sugiere que las posiciones de ajedrez no siguen una distribución simétrica respecto del reflejo vertical, a pesar de ser un juego que en sus reglas y configuración inicial sí lo es (con la úncia excepción de que las blancas juegan primero).

4.3.3. Posiciones de otras etapas de la partida

Como el autoencoder fue entrenado usando solamente posiciones correspondientes a la movida 20 de cada partida, es esperable que el error de reconstrucción también aumente si lo medimos con posiciones correspondientes a otra etapa del juego. Esto puede verse en la figura 4.38. El gráfico muestra el ECM de conjuntos de 100 mil posiciones corresondientes a cada etapa del juego, excepto por el primer punto (el de abajo a la izquierda) que se correponde con el ECM original calculado en la sección 4.1. Es razonable que el error aumente a medida que la etapa de la partida con la que se lo mide se aleja de las 20 jugadas, porque las posiciones suelen ser cada vez más diferentes (por ejemplo, tienen cada vez menos piezas en el tablero). Ver figura 4.45.

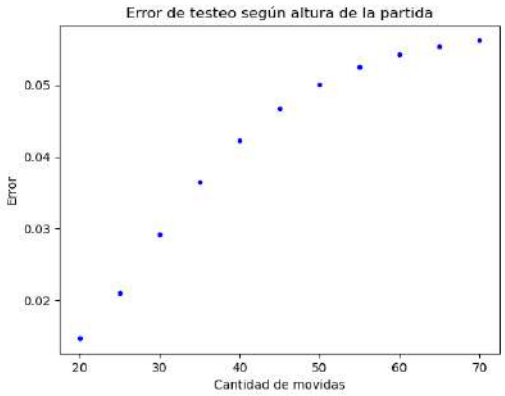

Fig. 4.38: Error de reconstrucción del autoencoder (ECM) medido con conjuntos de posiciones correspondientes a distintas etapas del juego.

Fig. 4.39: 20 movidas.

Fig. 4.40: 30 movidas.

Fig. 4.41: 40 movidas.

Fig. 4.42: 50 movidas.

Fig. 4.43: 60 movidas.

Fig. 4.44: 70 movidas.

Fig. 4.45: Distintas etapas de una misma partida, contando movidas desde la posición inicial.

5. CONCLUSIONES

En este trabajo entrenamos y analizamos un autoencoder para representar posiciones de ajedrez en un espacio latente de baja dimensión. Mostramos que el modelo logra una reconstrucción razonable de las posiciones originales y que su representación interna captura estructuras relevantes del juego, como el estado del enroque y la disposición de los peones centrales.

Además, exploramos la performance del modelo ante posiciones atípicas, como aquellas jugadas por jugadores principiantes; o posiciones generadas reflejando verticalmente e invirtiendo colores de otras ya existentes. En ambos casos, observamos una degradación en la calidad de reconstrucción, lo cual sugiere que el autoencoder ha aprendido patrones característicos del ajedrez de alto nivel.

Estos resultados muestran que es posible construir representaciones comprimidas de posiciones de ajedrez que preservan información estratégica y táctica, abriendo la puerta a futuras aplicaciones en análisis automático de partidas, generación de contenido y aprendizaje no supervisado en juegos.

Bibliografía

[1] Computer Generates Human Faces, CodeParade, YouTube, 2017. https://youtu.be/4VAkrUNLKSo?si=TJFrx0qbF0jkJFnB

[2] Lichess Database. Lichess, 2025. https://database.lichess.org/

[3] Chess rating systems. Lichess, 2025. https://lichess.org/page/rating-systems

[4] Caulfield, Tristan. Undergraduate Dissertation: Acquiring and Using Knowledge in Computer Chess. University of Bath, Department of Computer Science, 2004. https://purehost.bath.ac.uk/ws/portalfiles/portal/529627/CSBU-2004-17.pdf

[5] Caulfield, Tristan, and Joanna J. Bryson. Chess by Imitation.

[6] Ezequiel Cribioli, Evolutive EigenGame: Resolviendo PCA con un algoritmo evolutivo.Tesis de Licenciatura en Ciencias Matemáticas, Departamento de Matemática, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2022. https://cms.dm.uba.ar/academico/carreras/licenciatura/tesis/2022/Cribioli%20-%20tesis.pdf

[7] Gayen, Sutanu. Chess Endgame Classifier using Machine Learning. (2012). https://www.cse.iitk.ac.in/users/cs365/2012/submissions/sutanug/cs365/projects/report.pdf

[8] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron (2016). 14. Autoencoders. Deep learning. Adaptive computation and machine learning. Cambridge, Mass: The MIT press. ISBN 978-0-262-03561-3.

[9] Jolliffe, I. T. (2002). Principal Component Analysis. Springer Series in Statistics. New York: Springer-Verlag. doi:10.1007/b98835. ISBN 978-0-387-95442-4.

[10] Kapicioglu, Berk, et al. Chess2vec: learning vector representations for chess. arXiv preprint arXiv:2011.01014 (2020). https://arxiv.org/pdf/2011.01014

[11] Maesumi, Arman. Playing chess with limited look ahead. arXiv preprint arXiv:2007.02130 (2020). https://arxiv.org/pdf/2007.02130

[12] David, Omid E., Nathan S. Netanyahu, and Lior Wolf. Deepchess: End-to-end deep neural network for automatic learning in chess. Artificial Neural Networks and Machine Learning–ICANN 2016: 25th International Conference on Artificial Neural Networks, Barcelona, Spain, September 6-9, 2016, Proceedings, Part II 25. Springer International Publishing, 2016. https://arxiv.org/pdf/1711.09667

Chau chau chauuuuuuuuuuu