"Aprendizaje de Redes MLP Basado en Monte Carlo Tree Search para el Ajedrez Cuántico"

Autor/Autores:

1. Piero Marcelo Pastor Pacheco 20210836
2. Fabrizio Gabriel Gómez Buccallo 20212602
3. Jean Paul Tomasto Cordova 20202574
4. Jesus Mauricio Huayhua Flores 20196201
5. Pablo Eduardo Huayanay Quisocala 20193484

Este proyecto fue realizado con la finalidad de implementar una red neuronal capaz de usar los conceptos de la física cuántica aplicada a los tableros de ajedrez.

1 Introduccion

1.1 ¿Qué es el ajedrez cuántico?

El ajedrez cuántico es una variante del ajedrez que incorpora conceptos de la mecánica cuántica, como la superposición y el entrelazamiento. Las piezas no ocupan posiciones concretas en el tablero; pueden existir en diferentes lugares simultáneamente, creando una dinámica compleja para los jugadores involucrados.

1.2 Objetivo del trabajo académico

Se busca desarrollar una inteligencia artificial (IA) capaz de jugar este juego tan particular, que además maneje los conceptos de la mecánica cuántica involucrados, para jugar de manera competitiva esta variante tan extraña de ajedrez.

1.3 Aplicaciones

Debido a su manejo de decisiones a nivel cuántico, puede ser útil en áreas donde se necesita tomar decisiones en sistemas cuánticos, además de permitir una comprensión menos compleja de los algoritmos cuánticos y modelos de inteligencia artificial en contextos de alta complejidad.

2 Trabajos Relacionados

2.1 DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess

Este trabajo describe el desarrollo de DeepChess, un modelo de red neuronal profunda que aprende a evaluar posiciones de ajedrez sin conocimiento previo de las reglas del juego. El entrenamiento se realiza utilizando una gran base de datos de partidas de ajedrez, combinando preentrenamiento no supervisado con entrenamiento supervisado para comparar posiciones y elegir la más favorable. DeepChess es relevante para el desarrollo de un modelo de red neuronal para ajedrez cuántico porque demuestra la viabilidad de entrenar un modelo desde cero, sin reglas predefinidas, lo cual es aplicable a la naturaleza compleja y diferente de las reglas del ajedrez cuántico. [David et al., 2016]

2.2 Monte Carlo Tree Search, Neural Networks & Chess

El documento analiza la combinación de Monte Carlo Tree Search (MCTS) y redes neuronales convolucionales (CNN) con la finalidad de crear un motor de ajedrez que pueda competir contra otros motores como Stockfish. Inspirado en AlphaZero, se utiliza MCTS para explorar posiciones y CNN para evaluar la calidad de las jugadas. Este enfoque es relevante para un modelo de ajedrez cuántico porque MCTS podría ser una herramienta útil para manejar la complejidad de las posibles configuraciones cuánticas, mientras que las CNN pueden ayudar a evaluar posiciones no tradicionales en el tablero. [Steinberg and Jarnagin, 2021]

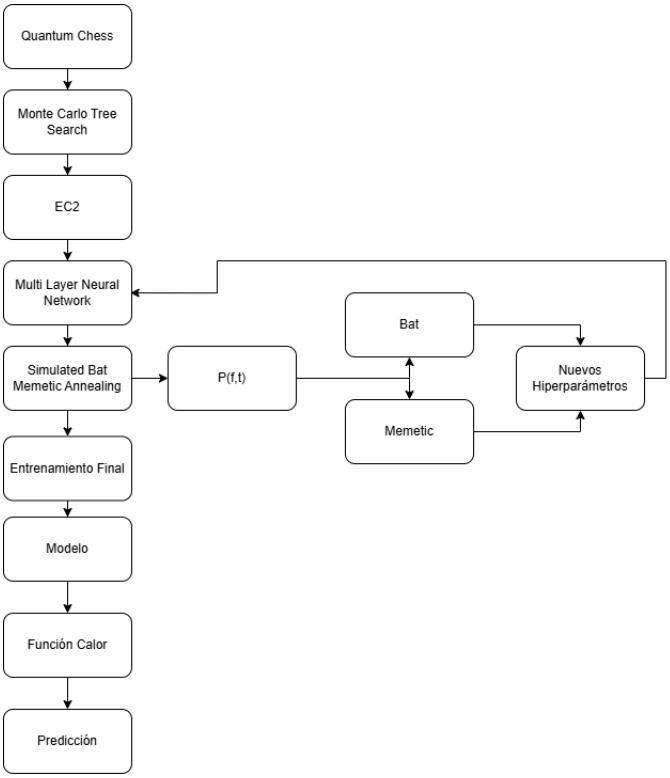
2.3 Aprendizaje por refuerzo y ajedrez: Alpha Zero

AlphaZero, es un modelo basado en aprendizaje por refuerzo que se entrena jugando millones de partidas contra sí mismo que utiliza utilizando redes neuronales y búsqueda de Monte Carlo con la finalidad de evaluar posiciones (states) y tomar decisiones sobre cada una de las jugadas que debe realizar. Sin necesidad de depender de datos preexistentes, aprende estrategias avanzadas y no convencionales, optimizando su rendimiento mediante la mejora continua en base a experiencias previas, lo que le permi-

tió superar a programas tradicionales como Stockfish en ajedrez. [Ezquerra, 2022]

3 Metodología

Con el fin de resolver las necesidades del ajedrez cuántico, se requiere el uso de una red neuronal que sea capaz de interpretar y utilizar los conceptos del ajedrez cuántico.
Para poder jugar de manera correcta, coherente y desafiante al ajedrez cuántico, se necesita una red neuronal que pueda calcular movimientos a partir de un estado inicial. Dicho estado es un tablero común y corriente con las piezas en él. Para alcanzar este objetivo, se ha planteado el siguiente gráfico, que representa el pipeline mediante el cual se obtendrá la data, para usarse posteriormente en el entrenamiento de la red neuronal, considerando también el ajuste de hiperparámetros mediante un algoritmo metaheurístico.



A continuación, una breve descripción de cada componente desarrollado para la implementación final del juego.

3.1 Quantum Chess

Sobre él se juegan las partidas, también se le puede llamar entorno de juego. Tiene implementadas tanto las reglas del juego tradicional, como las reglas de la mecánica cuántica para hacerlo un juego más interesante. Además, ya contiene el tablero y las piezas necesarias para poder desarrollar todas las simulaciones posibles, y también para que pueda ser jugado por una persona común.
Los movimientos especiales con mecánica cuántica que se emplearon, son los siguientes:

3.1.1 Movimiento

Para poder trabajar el movimiento convencional que implica cambiar la posición de una pieza entera, o una superposición de esta misma a otra, se utilizó la compuerta iSWAP, que se debe aplicar entre los dos qubits que intercambiarán valores.
Esta compuerta intercambia el valor de un qubit con otro al colapsar el circuito completo, lo cual se debe a la matriz que representa la compuerta matemáticamente.

$$U_{iSwap} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Se utiliza esta porque es más sensible a los patrones de memoria y puede aprender de manera más segura los patrones del circuito y así se evitan swaps inefectivos que dejarían el juego injugable por la aleatoriedad.

3.1.2 Split

Para poder realizar una superposición entre dos posiciones, se debe de mantener el uso de la compuerta previamente analizada, pero esto sería únicamente para uno de los dos destinos que implican el split. El otro destino debe verse evaluado mediante sqrt(iSWAP), entre este y el mismo origen.
Esta raíz sobre la compuerta, efectúa una superposición que implica que más adelante cuando se realice el colapso, existirán un 0.5 de probabilidades que aparezca en una posición u otra. Cuantos más splits haya, se irán reduciendo las probabilidades de manera lineal al respecto de cada split o slide realizado.

$$U_{\sqrt{i\text{SWAP}}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{i}{\sqrt{2}} & 0 \\ 0 & \frac{i}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3.1.3 Slide

El slide implica que se realice un movimiento atravesando a una pieza y pasando sobre ella como si no estuviera. Para poder hacer uso de esta regla, la pieza que será obviada, debe de encontrarse en alguna superposición. Y la compuerta que maneja este proceso, es el Controlled-iSWAP, y se tendrá que aplicar como controlador a la casilla que

será atravesada, para sí en caso sea cero se tome en cuenta ese movimiento como válido, mientras que se la pieza controladora se mantuvo ahí al colapsar; el movimiento se invalide por completo.

$$U_{\text{slide}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & i & 0 & 0 & 0 & 0 & 0 \\ 0 & i & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

3.1.4 Split-Slide

Para poder realizar un split-slide, se optó por utilizar una combinación de compuertas. Es decir, se realizará un Controlled-iSWAP entre la posición y su destino con slide. Luego, en caso de que no haya otro slide, se realizará un $\sqrt{iSWAP}$ hacia la dirección sin mayor problema. En caso de que sea un slide en la misma dirección, se realizará un $\sqrt{iSWAP}$, pero tomando como inicio el qubit donde se realizó inicialmente el Controlled-iSWAP. Y, por último, en caso de que se vaya en dos direcciones distintas y ambas requieran slide, a la segunda se le realizará un $\sqrt{Controlled - iSWAP}$ desde el mismo qubit de origen.

3.1.5 Merge

Es solo un movimiento que, a nivel lógico, permite que se encuentre en esa posición, mas no a nivel matemático, porque luego, cuando se realice el colapso, el circuito generado simulará todo y, como ese resultado comenzó a moverse junto, no importa de dónde llegue; lo que importa es que, por un lado u otro, se mantendrá el camino. La compuerta utilizada será un Controlled-iSWAP, para que si un resultado tiene 0, el otro se lo envíe, y si este tiene 0, su contraparte se lo envíe.

3.2 Montecarlo Tree Search

Cumple la función de jugar de manera aceptable el juego. No obstante, se tendrá que limitar tanto su ramificación como su longitud de simulación, debido a la cantidad exorbitante de movimientos posibles a partir de un estado.
El algoritmo de monte carlo utilizará como función objetivo la menor cantidad de movimientos para poder ganar la partida. No obstante, también se hará uso de una heurística, esto porque aparte de guiar el camino para la solución; como muy probablemente el árbol no pueda llegar a simular hasta tener el coste total, entonces la heurística al ser optimista llevará por caminos según su propiedad.

$h1(n)$ = Puntaje del jugador en base al valor de cada pieza.

c = Coste (Cantidad de jugadas hasta el jaque mate)

1. **Admisibilidad**: La heurística resulta admisible porque relaja el problema. Ya no busca realizar jaque mate que puede resultar complejo, comenzará a buscar la mayor puntuación que implica solo comer las más valiosas posibles.
2. **Consistencia**: Es consistente, porque conforme se tengan más sucesores, estos siempre resultan más sencillos debido a la cantidad de puntaje reducido.

En base a lo que pueda jugar el algoritmo, se le pondrá como contrincante otro igual para que así simulen partidas; y como el algoritmo buscará el mejor movimiento posible, entonces de esta manera se nos proveerá de data útil para la red neuronal, cosa que realizando partidas con personas resultaría tanto más costoso, como no necesariamente valioso, porque estas personas pueden cometer errores graves. Se simularán algunas partidas, pero también se generarán estados aleatorios a los que el algoritmo MCTS tendrá que darles solución, y en base a esta se generarán los datasets.

3.3 Dataset Generado

El dataset que se genere a partir de las partidas de Monte Carlo, consistirá en dos vectores. El primero será el tablero de ajedrez que tendrá identificada cada pieza con un número y un signo, además de un valor que indicará a quién le toca jugar.
Si bien el algoritmo MCTS está desarrollado en un archivo *.ipynb*; la generación y la simulación de todas las jugadas para la generación de datos, se realizan en una instancia de Amazon Linux en la plataforma AWS (Amazon Web Services), con un archivo *.py*.

3.3.1 Instancia EC2

Las especificaciones de la instancia generada son:

- **Nombre:** MCTS.
- **SO:** Linux.
- **Distribución:** Amazon Linux.
- **AMI:** Amazon Linux 2023 AMI 2023.6.20241121.0 x86_64 HVM kernel6.1
- **Arquitectura:** 64 bits (x86).
- **Tipo de instancia:** c5ad.large - 2 vCPU (3.3 GHz) - 4 GiB RAM - 75 GiB SSD NVMe.
- **Almacenamiento:** 20 GiB - gp3.

Se seleccionó la distribución de Amazon Linux, ya que resulta más liviana en comparación a distribuciones más conocidas y populares como Ubuntu. Además que está optimizada para operaciones de cómputo grandes en el servicio EC2.
El tipo de instancia, se seleccionó bajo el mismo criterio de soporte para un cómputo pesado, esto bajo las limitaciones del laboratorio de AWS Academy.

3.3.2 Entrada

Pieza	ID	Color Blanco	Color Negro	Valor en Vector
Torre	82	1	-1	Identificador * Color
Caballo	75	1	-1	Identificador * Color
Alfil	66	1	-1	Identificador * Color
Reina	81	1	-1	Identificador * Color
Rey	69	1	-1	Identificador * Color
Peon	80	1	-1	Identificador * Color
Espacio en Blanco	0	0	0	0

Y con estos valores, el tablero que se tenga en base a strings se convertirá en un arreglo numérico basado en el diccionario previamente establecido.
Con respecto al output para predecir, se utilizará de igual manera un vector. Este indicará:
$v = (\text{Movimiento}, \text{Origen1}, \text{Origen2}, \text{Destino1}, \text{Destino2}, \text{Coronación})$
Y los valores establecidos para cada uno de los parámetros del vector son:

3.3.3 Salida

3.3.3.1 Movimiento

Movimiento	Valor
Basico	1
Split	2
Swap	3

Para el movimiento no se toma en cuenta el slide, ya que, está implícito dentro de cualquier movimiento, menos el merge.

3.3.3.2 Orígenes y destinos

Valor	Necesita movimiento	No necesita
Origen 1	Valor de 0 a 63	-
Origen 2	Valor de 0 a 63	-1
Destino 1	Valor de 0 a 63	-
Destino 2	Valor de 0 a 63	-1

3.3.3.3 Coronación

Pieza	Valor
No hay Coronación	0
Torre	1
Reina	2
Alfil	3
Caballo	4

La data no llevará un tratamiento especial, ya que, como es generada por el computador, esta ya está completa y es fiable por el algoritmo que la generó

3.4 Multi Layer Neural Network with Output Heads

Tras conseguir de las simulaciones con Monte Carlo datos de entrenamiento. Estos se procederán a utilizar como input a la red neuronal.
Para el proceso de entrenamiento se tomarán los datos con un 25 % que será para el test, mientras que un 75 % para el train. De esta forma se podrá ver si las jugadas que realizará el modelo tienen relación con lo que aprendió y si realmente está trabajando bajo la misma lógica de Monte Carlo para llegar a las mismas conclusiones sobre qué movimientos realizar. Prácticamente es una regresión con respecto a MCTS.
La red neuronal multicapa se encargará de mejorar el proceso de aprendizaje, ya que, pasará por capas de neuronas, para así ajustar a cada parámetro de salida necesario de manera más óptima. Todo esto para que finalmente, en la última capa se le asignen distintos cabezales de salida (uno para cada output del vector planteado); se toma esta metodología para poder dar predicciones con valores de entre cero y uno para cada componente del vector resultado, aplicando softmax a cada cabezal independiente del otro (para eso el uso de cabezales de salida). Esto es porque como el resultado de la predicción no necesariamente es buena o mala respecto a la entrada, si se le asignara a predecir valores directamente existiría un sesgo en base a la distancia entre los resultados en el espacio; cosa que no aplica para el caso porque no necesariamente existirá un resultado mejor o peor, solo lo más aproximado a como jugaría MCTS.
La cantidad de neuronas por capas, y todos las funciones de activación por estas, se inicializarán aleatoriamente, pero según la bibliografía estudiada para el apartado de la profundidad, ya luego se irán ajustando con el uso del Bat Algorithm y Memetic Algorithm usando la metodología del Simulated Annealing Algorithm.
Finalmente, cuando se compruebe que la red puede predecir movimientos del algoritmo Monte Carlo, entonces se pondrá a prueba en juegos y matrices de confusión para ver su funcionamiento.

3.5 Hybrid Algorithm

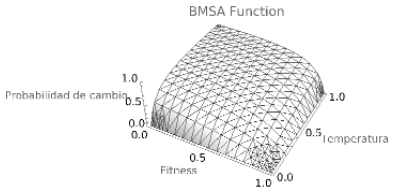
Se plantea un algoritmo híbrido que combina tres metaheurísticos para conseguir la mejor optimización de los hiperparámetros (neuronas por capa). De estos tres, se tomará uno que hará el papel de "switcher"para ocasionar el intercambio del uso entre los otros dos; así se logrará conseguir lo mejor del Bat Algorithm y el Memetic Algorithm.

3.5.1 Simulated Bat Memetic Annealing (SBMA) Algorithm

Este metaheurístico se encargará, mediante una variable de calor, ir intercambiando entre los algoritmos. Además, se tendrá en cuenta el fitness que genera cada uno de estos, ya que, su objetivo es de; si un algoritmo está dando muy buenos resultados, entonces lo normal sería que se mantenga mejorando mediante el ya seleccionado. Pero mediante la lógica del Simulated Annealing, y su variable de calor; para darle más explosividad y asegurar la mejora absoluta de resultados, se realizará el intercambio. Esto para evitar sesgos por el uso constante de un mismo algoritmo bajo sus propios parámetros optimizados.

La función que realizará de calcular la probabilidad de intercambio es $P(f,T) = 1 - \frac{1}{1+\alpha \times T \times (1-f)}$, donde:

- P = Probabilidad de salto.
- f = Fitness actual.
- T = Temperatura actual.
- α = Constante reguladora para ajustar limites. Usamos el valor 10.



Por otro lado ambos metaheurísticos tendrán la misma función de fitness; esta función es $Fo(m,t) = \frac{m}{1+\alpha t}$

- Fo = Fitness.
- m = 1-MAE del modelo
- t = Tamaño normalizado en MB del modelo.
- α = Proporción de importancia del tamaño de la red. Implementado con 0.05.

El algoritmo en cuestión presenta la siguiente estructura:

Algorithm 1 SBMA

$G : (\forall i : 0 <= i < N_{configuraciones} : bounds[i][0] < configuraciones[i] < bounds[i][1])$
Require: $Pre : N_{configuraciones} = N_{bounds} \wedge dimension \geq 1$
$proc\ SBMA(value\ N_{configuraciones} :\ integer$
$value\ dimension :\ integer$
$value\ T :\ float$
$value\ DR :\ float$
$value\ MaxIter :\ integer$
$value\ N_{generaciones} :\ integer$
$value\ threshold :\ integer$
$value\ test\_size :\ float$
$value\ alpha :\ float$
$value\ gamma :\ float$
$value\ T_{casamiento} :\ float$
$value\ T_{mutacion} :\ integer$
$value\ P_{casamiento} :\ integer)$
$best_fitness\ :=\ 0$
$fitness\ :=\ 0$
$algoritmo\ :=\ random(0,1)$
$\alpha = 10$
$i\ :=\ 0$
$\{invariant :\ fitness\ best_fitness\ G\}$
$\{bound :\ MaxIter\}$
$\{invariant :\ fitness\ best_fitness\ G\}$
while do $i < MaxIter$
  if $algoritmo = 1$ **then**
    $configuracion, fitness, configuraciones, c\_fitness := MemeticAlgorithm(alpha, gamma)$
  else if $algoritmo = 0$ **then**
    $configuracion, fitness, configuraciones, c\_fitness := BatAlgorithm(T\_casamiewnto, T\_mutacion, P\_casamiento)$
  end if
  if $fitness \geq best\_fitness$ **then**
    $best\_fitness := fitness,\ best\_configuracion := configuracion$
  end if
  if then $fitness \geq best\_fitness$
    $best\_fitness :=\ fitness,\ best\_configuracion\ :=\ configuracion$
  end if
  $probabilidad\ :=\ 1 - \frac{1}{1+\alpha \times T \times (1-fitness)}$
  if $random() < probabilidad \rightarrow$ **then**
    if $algoritmo = 1 \rightarrow algoritmo\ := 0$ **then**
      $algoritmo = 0 \rightarrow algoritmo := 1$
    end if
  end if
  $if best_fitness thresholdbreakfi$
  $T := T \times DR$
  $i := i + 1$
end while
$postcondition : 0 < best_{fitness} \leq 1$
$retbest\_fitness, best\_configuracion$
Ensure: $Post : 0 < best_{fitness} \leq 1$

Otra forma de entenderlo es:

Algorithm 2 SBMA

Require: $Q : N_{configuraciones} = N_{Bounds} \wedge dimension \geq 1$
Ensure: $R : 0 < best_{fitness} \leq 1$
1: $Q \Rightarrow wp(SBMA, R)$

3.6 Bat Algorithm

Se basa en la ecolocalización de los murciélagos para así poder llegar a la solución más óptima, esto porque mediante la técnica previamente mencionada se llaman unos a otros y comienzan a viajar a ciertos puntos para conseguir la mejor solución. En este caso sería la cantidad de

neuronas por capa. Este algoritmo proporcionará la explotación al SBMA, como únicamente se encarga de ajustar velocidades y puntos de cada murciélago; mejorará ese conjunto de soluciones lo más posible en base a un mejor local.

3.7 Memetic Algorithm

Se basa en la evolución muy similar a los algoritmos genéticos evolutivos. Con la diferencia en que, para mantener una población mínima y constante, siempre se elimina a toda la mayoría de esta bajo cierto nivel de aleatoriedad, pero siempre dando mayor prioridad de eliminación a los que presentan peores resultados. Se mantiene la mutación y reproducción entre cromosomas. Este algoritmo proporcionará la exploración al SBMA, como va generando nueva población y deshaciéndose de la peor, si bien mantiene explotación por ese lado; su punto fuerte es la exploración en la búsqueda de nuevos posibles individuos.

3.8 Greedy Search

Se programó el Greedy Search para poder comparar el desempeño del algoritmo híbrido con este, ya que, se considera un estándar en la optimización de hiper parámetros. No obstante, se modificó la inicialización de datos, ya que, no partirá de entradas dadas por el usuario, sino las generará aleatoriamente para proporcionarle algo más de explotación, y ya luego se ajustará sin problemas probando las combinaciones respectivas.

3.9 Despliegue

El uso de la red neuronal será únicamente sus pesos para poder predecir a partir de un tablero dado, que movimiento realizar. Para esto, se realizará un despliegue dentro de Python para que se pueda jugar con un tablero elaborado en Pygame; la metodología de juego será por turnos donde empezará el jugador con las blancas, y la red neuronal será las negras, o como lo seleccione el usuario.
Además se plantea el uso de calor en las predicciones para poder variar en los resultados, y además de no jugar siempre igual; en caso haya una predicción errónea que contraiga un movimiento imposible, se volverá a predecir hasta llegar a una solución permitida. En caso se falle muchas veces, se procederá a tomar una decisión aleatoria, pero se irán guardando componentes coherentes para que la aleatoriedad únicamente afecte a ciertas partes del vector movimiento, y que no sea una decisión completamente aleatoria.

3.10 Función de calor

La función de calor lo que hará será generar probabilidades para poder tomar otras posibles soluciones, similarmente buenas que la mejor. Esto porque la probabilidad de que se seleccione una de ellas, es directamente proporcional, al porcentaje que predijo la red neuronal. Para cada probabilidad de seleccionar un resultado en el arreglo de estos, se le aplicará la función:

$P(L_i, T) = \frac{e^{\frac{L_i}{T} - max(\frac{L}{T})}}{\sum_{i=1}^{n}(e^{\frac{L_i}{T} - max(\frac{L}{T})})}$, donde:

- **P** = Nueva probabilidad
- **L** = Todas las predicciones originales de la red
- L_i= Indica una predicción de todas
- **T** = Temperatura

4 Experimentación y Resultados

Mediante la presente sección se busca demostrar que tan eficaz y eficiente puede resultar la red neuronal para responder a las diversas situaciones presentadas durante el ajedrez cuántico.
Por ser un proyecto dedicado a lo que son entornos de juego, dentro de sus simulaciones propias la red neuronal creerá que está tomando decisiones correctas, mientra que en ambientes reales esto puede cambiar considerablemente dependiendo de como se le presenten nuevas situaciones a la red neuronal de las cuales tendrá que aprender para poder mejorar. Terminando con estas aclaraciones, se procederá a ver lo referido a todo lo referido a los experimentos realizados.

4.1 Setup Experimental

4.1.1 Datos usados

Para el presente trabajo, hay que recalcar algunos detalles. En este caso específico, los datos usados para los experimentos detallados a continuación han tenido que ser generados por computadora, sometiendo al entorno de juego a experimentos usando tres tipos de casos con el fin de obtener datos suficientes y necesarios para entrenar la red neuronal que será la encargada de jugar al ajedrez cuántico. Con esta finalidad, se han definido dos tipos de jugadores: **Random Player** (es un jugador que toma decisiones completamente aleatorias, no hace un análisis altamente profundo y representa a los jugadores muy novatos en el ajedrez, permitiendo una generación de datos altamente veloz) y **Montecarlo Player** (representa a jugadores más experimentados haciendo un análisis profundo de las situaciones de juego; cabe resaltar que, debido al análisis que realizará, el procesado de las jugadas será mucho más lento que en el tipo de jugador previamente mencionado). Finalmente, las formas en las cuales se han conseguido los datos han sido las siguientes:

1. Random Player vs Random Player:

2. Ramdom Player vs Montecarlo Player:

3. Montecarlo Player vs Montecarlo Player:

Con la finalidad de optimizar la ejecución y la obtención de datos, se ha tenido que realizar estas ejecuciones de manera independiente, en diversas líneas de ejecución. Además, debido al tamaño de las ejecuciones, después de haber terminado una simulación, se han tenido que eliminar los datos en caché de las simulaciones previas a medida que avanzan, para poder realizar las simulaciones con mayor eficacia.

Referido a esto, se usaron dos formas de obtener los datos. La primera requirió el propio trabajo de Google Colaboratory para generar datos; sin embargo, las limitaciones propias del sistema impiden una generación amplia, que era lo requerido para el trabajo presentado. Por lo tanto, se recurrió al uso de instancias de Amazon Web Services (AWS) para optimizar y llevar a cabo esta tarea con mayor rapidez, aprovechando que el sistema de Amazon tiene mayor potencia para generar datos y mantenerse ejecutando cuando los jugadores Montecarlo simulan partidas entre sí, ya que requieren un alto consumo de memoria que recientes actualizaciones de Google Colaboratory han restringido a los usuarios.

Por último, cabe resaltar que los datos obtenidos se han almacenado en un archivo CSV, lo que permitirá cargar los datos en la red neuronal.

4.1.2 Métricas de evaluación

4.1.2.1 Perdida (Loss)

La función de pérdida que será utilizada es **çategorical crossentropy"**. Esta misma función se le asignó a cada cabezal; no obstante, como convergen todos en la capa anterior, debido a que comparten todas las capas menos la de salida, se mantendrá el análisis de pérdida tomando las pérdidas como un todo y no como predicciones independientes.

No se asignarán pesos a cada pérdida, ya que todos los componentes del vector tienen el mismo valor, y se deberán analizar como una única predicción. No se tomará un promedio de pérdidas, porque si sucede un error en algún cabezal, es necesario que esta afecte rotundamente a la pérdida total, cosa que, si se promediaba, su error no sería tan influyente para el entrenamiento, y requerimos su magnitud absoluta. Es por esto que la función final de pérdida será:

$$L_{\text{total}} = \sum_{i=1}^{n} -\frac{1}{m}\left(\sum_{j=1}^{m}\sum_{k=1}^{c} y_{jk}\log(\hat{y}_{jk})\right)$$

donde:

$$\begin{aligned} n &= \text{Número de salidas} \\ m &= \text{Número de muestras} \\ c &= \text{Número posible de resultados} \\ y_{jk} &= \text{Resultado real} \\ \hat{y}_{jk} &= \text{Resultado predicho} \end{aligned}$$

4.1.2.2 Error Absoluto Medio(MAE)

Esta métrica será utilizada para determinar el rendimiento de la red neuronal; lo que se realizó fue asignar esta función a cada uno de los cabezales. De esta manera, como se tendrá un MAE por cabezal a la hora de evaluar el modelo, realizaremos un promedio para conocer el desempeño general de la red neuronal empleada. Se hizo selección de esta métrica porque mide qué tan cerca están las predicciones de los valores reales, sin requerir una coincidencia exacta. Justamente no se pretende que se tenga algo muy cercano a lo real, porque debido a la cantidad de posibles resultados resulta improbable; por ello, cuanto menor sea el MAE, implica que se tienen muy buenos resultados.

4.1.2.3 Algoritmos de optimización

Ya que se requiere hacer un análisis de grandes cantidades de jugadas, se requiere tener algoritmos que permitan hallas la mejor posible (optimizando el resultado). Como parte de la investigación y experimentación, se comparará el proceso de mejora del fitness de cada uno de los algoritmos (Greedy Search y SBMA). Esto se realizará guardando el mejor fitness, que se representa por la función presentada en la sección de metodología, por cada iteración de mejora que realice cada uno de los optimizadores. Luego con estos datos se realizará una gráfica para realizar las evaluaciones de sus desempeños. Ambos funcionarán bajo la misma cantidad de capas, y la misma función de fitness, para que ambos estén bajo la misma métrica, finalmente esta se calculará mediante la metodología de Train-Test.

4.1.3 Configuración experimental

Para la generación del dataset, se hará uso de la instancia en AWS mencionada previamente en la metodología, además de sesiones en Google Colaboratory, utilizando la CPU.

Por otro lado, para la optimización de hiperparámetros se utilizará, de igual manera, el último software mencionado. Sin embargo, para evitar problemas con la memoria RAM, debido a la cantidad de posibles soluciones y modelos generados, se utilizó una v2-8 TPU, ya que está proporcionaba 334.56 GiB de memoria RAM.

Una vez conseguidos los hipermetamorfosis optimizados, se procederá a comparar, mediante validación cruzada, el

desempeño de la red utilizando SGD y Adam como optimizadores. Dado que la cantidad de modelos es menor, se hará uso de la sesión de tipo T4 GPU para otorgar velocidad en el entrenamiento.
Conseguido el mejor optimizado, y manteniendo la misma sesión, se realizará el entrenamiento final del modelo ya seleccionado.

4.1.4 Escenarios probados

4.1.4.1 Escenario 1: Redes aleatorias

Descripción: Inicialmente se tienen una cantidad aleatoria de neuronas por capa, esta cantidad de capas se define inicialmente con 6; y los límites de neuronas por capas es de [0; 1024].
Objetivo: Inicializar todo para comenzar el ajuste.

4.1.4.2 Escenario 2: Red ajustada

Descripción: Una vez se haya culminado con la optimización de hiper parámetros ya se tiene una red con un MAE adecuado, y un peso igualmente regulado.
Parámetros:

1. Neuronas por capa: [561; 9; 176; 293; 1008; 95]
2. Fitness: 0.8765378294944639

Objetivo: Tener ya una configuración para la red definida.

4.1.4.3 Escenario 3: Optimizador seleccionado

Descripción: Con la red ya elegida se procedió a seleccionar un optimizador entre SGD y Adam basándose en los resultados de una cross validación.

Parámetros:

1. Optimizador Seleccionado: SGD
2. 1-MAE: 90.854

Objetivo: Tener un optimizador para ya conseguir un modelo completo para el entrenamiento.

4.1.4.4 Escenario 4: Entrenamiento y resultados

Descripción: Una vez que la red ya está preparada, se genera el modelo final y se entrena. Para luego ser evaluado
Parámetros: MAE–>8.899 %
Objetivo: Entrenar el modelo y analizar la minimización del MAE.

4.1.5 Estrategia de validación

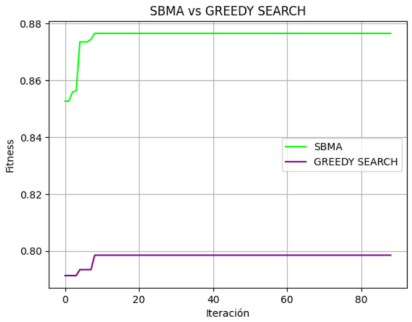
Para validar en los algoritmos de optimización se utilizó *Train-Test*; por otro lado, para la evaluación de los optimizadores de la *MLP*, se utilizó *Cross Validation* (en este caso, para hallar el resultado total se utilizó un promedio). En la división de datos, esta fue de un 25 % para pruebas, y 75 % para entrenamiento.

4.2 Experimentación

4.2.1 Resultados:

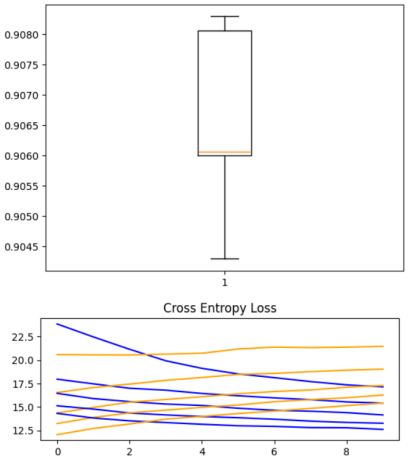
4.2.1.1 Resultados SBMA vs Greedy Search

Con el fin de evaluar el desarrollo del experimento por iteración entre cada uno de los algoritmos de optimización.

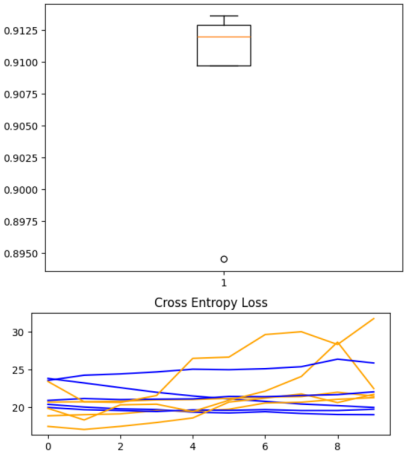


4.2.1.2 Resultados SGD vs Adam

Adam: Los resultados son de: 1-MAE: mean=90.654 std=0.148, Cantidad de Folds = 5

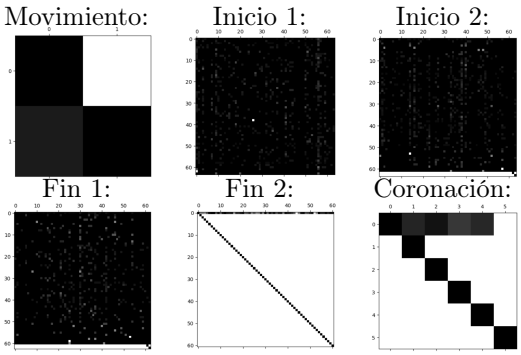


Se puede observar según el boxplot la media de 1-MAE es de 90.654 %, además según las curvas de cross entropy loss, al usar Adam, si bien no hay overfitting, hay mucha diferencia entre las pérdidas al momento de realizar los test y entrenamientos.
SGD: Los resultados son de: 1-MAE: mean=90.854 std=0.714, Cantidad de Folds = 5.



Según la mediana del boxplot, se tiene un 91 % de 1-MAE; por lo que se tienen mejores resultados con este optimizado. Sumado a esto, las curvas si bien hay ocasiones en las que se disparan las de pruebas, las demás si mantienen una mejor relación con sus contrapartes de entrenamiento

4.2.2 Resultados finales



5 Conclusiones

Importancia del Diseño Experimental:
Los escenarios probados permiten identificar cómo los diferentes parámetros y configuraciones influyen en el desempeño del modelo. Esto destaca la necesidad de un diseño experimental cuidadoso para obtener conclusiones robustas.
Impacto de las Funciones de Loss en el Aprendizaje Multisalida: La estrategia de asignar funciones de pérdida específicas a cada cabezal en tu modelo multisalida permite capturar con precisión las características de cada componente del sistema (movimientos, posiciones iniciales y finales, y promoción de piezas).
Contribución al Campo de Ajedrez Cuántico:
Este proyecto no solo desarrolla una red neuronal funcional para un problema altamente complejo, sino que también sienta las bases para investigaciones futuras en el cruce entre inteligencia artificial y computación cuántica.

Eficiencia del SBMA en la Optimización de Hiper Parámetros:
El uso del algoritmo SBMA (Simulated Bat Memetic Annealing) permitió explorar de manera eficiente un espacio de búsqueda complejo para la arquitectura de la red neuronal. Su capacidad de balancear la explotación y la exploración resultó en configuraciones de hiper parámetros que mejoran tanto la precisión como la eficiencia computacional. A pesar de trabajar con datos ruidosos y limitaciones computacionales, el algoritmo mantuvo un desempeño consistente, lo que refuerza su robustez como herramienta de optimización en problemas de gran escala.

6 Sugerencias

1. Principalmente, se podría mejorar desde el lado de recursos computacionales para obtener más datos y mejor calidad en el dataset. Para ello, se puede explorar el uso de infraestructura computacional más potente, como clústeres de GPU de alto rendimiento.

2. Se podría evaluar cómo la simulación de datos obtenidos por Montecarlo afectan a la red neuronal. Por un lado, cómo el número de iteraciones y el método de muestreo (uniforme, estratificado) afectan la calidad del dataset y, por ende, el desempeño de la red neuronal.

3. Por otro lado, se podría evaluar el uso de funciones de pérdida basadas en equilibrios de Nash para encontrar estrategias óptimas.

7 Implicaciones Éticas

1. Seguridad de Datos: Respecto a este punto, no se trabaja con datos personales de ninguna persona. Por el contrario, la data que debe manejarse para el Trabajo Académico es una data que no implica riesgos para la seguridad de nadie.

2. Sesgo: Se priorizan movimientos que puedan acabar en el menor tiempo posible las partidas llevadas a cabo por la IA desarrollada en el documento.

8 Repositorio

Link al repositorio de github link.

9 Declaración y contribución

1. Piero Marcelo Pastor Pacheco: Implementación de reglas de juego cuántica en el tablero, implementa-

ción de la red neuronal, interpretación de resultados, realización de conclusiones

2. Fabrizio Gabriel Gómez Buccallo: Implementación del tablero de ajedrez con las reglas de un ajedrez común, interpretación de resultados, realización de conclusiones,realización de los experimentos.

3. Jean Paul Tomasto Cordova: Búsqueda de información y corrección de errores en el tablero creado para el trabajo académico, interpretación de resultados, realización de conclusiones

4. Jesus Mauricio Huayhua Flores: Redacción de documento, interpretación de resultados, realización de conclusiones.

5. Pablo Eduardo Huayanay Quisocala: Implementación del tablero de ajedrez, interpretación de resultados, realización de conclusiones

10 Referencias

Referencias

[David et al., 2016] David, O. E., Netanyahu, N. S., and Wolf, L. (2016). Deepchess: End-to-end deep neural network for automatic learning in chess. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9887 LNCS:88 – 96. Cited by: 49; All Open Access, Green Open Access.

[Ezquerra, 2022] Ezquerra, M. S. (2022). Aprendizaje por refuerzo y ajedrez: Alpha zero.

[Steinberg and Jarnagin, 2021] Steinberg, M. D. and Jarnagin, Z. R. (2021). Monte carlo tree search, neural networks & chess. Master's thesis, Northeastern University, Khoury College of Computer Sciences.