Métodos de Aprendizaje Automático
Obligatorio 1

Bruno Garate & Nicolás Izquierdo & Guillermo Siriani

1. Diseño del jugador

1.1. Regressor ANN

El jugador propuesto debería tomar todas las jugadas posibles y evaluar la que llevase al mejor tablero posible. Es decir, para cada jugada posible se evaluaría el tablero resultante a partir de una *función de evaluación estática* que dado una matriz T que represente el tablero, devolvería un valor asociado a la probabilidad de ganar la partida.

El método de evaluación sería entonces una *Regressor Artificial Neural Network*. La entrada sería un vector de 64 valores representando el estado de los 64 escaques del tablero.

$$t(t_0, t_1, t_2, ..., t_{64})$$

$$t_i = \begin{cases} 1 & \text{ficha propia en } T_i \\ 0 & \text{escaque vacío en } T_i \\ -1 & \text{ficha del contrincante en } T_i \end{cases}$$

Y la salida sería un escalar $V(t)$ asociado a la estimación de la red de la posibilidad de victoria, derrota o empate. La victoria, empate y derrota se codificaría de la siguiente forma:

$$V(t) = \begin{cases} 1 & \text{Victoria} \\ 0 & \text{Empate} \\ -1 & \text{Derrota} \end{cases}$$

Mediante backpropagation se aplicaría el aprendizaje sobre la red. Se experimentó con neuronas ReLU o Rectifier Linear Unit. Estas utilizan como función de activación una función rectificadora:

$$f(x) = max(0, x)$$

Se empleo la implementación de redes neuronales feedforward regressor de la libería scikit de Machine Learning.

1.2. $TD(\lambda)$ vs dinámica vs Monte Carlo

Una de las decisiones que afectan el aprendizaje es la forma en que se responsabilizan las jugadas individuales por el resultado del partido. Lo que afecta esta decisión es la salida esperada de la red para cada evaluación del tablero realizada en el entrenamiento.

Para la última jugada n del partido, la salida esperada en la neurona de salida de la red es el valor final de la partida

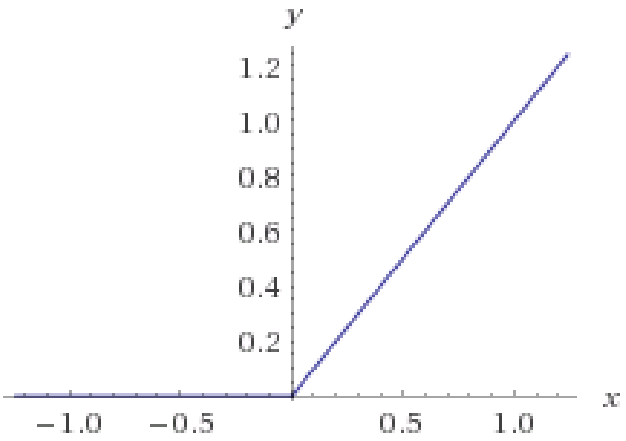

Figura 1: Función rectificadora $f(x) = max(x, 0)$

$$o_{esperada}(t_n) = V_{partida}$$

Para las otras jugadas se consideraron otros enfoques. Se puede argumentar que la red, en una jugada muy alejada del final de la partida no tiene porque predecir el resultado final de la partida.

Un extremo de esta consideración, visto como *programación dinámica*, es asumir que la red debe ser capaz de predecir únicamente el valor del siguiente valor del tablero, es decir un turno más tarde.

$$o_{esperada}(t_i) = V(t_{i+1})$$

Otro extremo, empleado en las técnicas de Monte Carlo, se encuentra en considerar que cada evaluación del tablero debe ser capaz de predecir el resultado final de la partida.

$$o_{esperada}(t_i) = V_{partida}$$

Una forma intermedia empleada en aprendizajes por refuerzos, que emplea *Time Differences* para evaluar las jugadas intermedias, mediante *Eligibility Traces*, llamada $TD-\lambda$. Es el método aplicado en el programa TD-Gammon por Tesauro.

Lo aplicamos responsabilizando de forma exponencialmente decreciente a las decisiones de los resultados futuros.

$$o_{esperada}(t_n) = V_{partida}$$

$$o_{esperada}(t_i) = V(t_i) + \lambda(V(t_i) - V(t_{i+1}))$$

$$con\lambda \in [0, 1]$$

Los casos extremos $\lambda = 0$ y $\lambda = 1$ pertenecen a las visiones de programación dinámica y Monte Carlo respectivamente.

1.3. Minmax y otras extensiones

Minmax

Para mejorar la tasa de partidas ganadas por parte de nuestro jugador se optó por usar el algoritmo de toma de decisiones *Minmax*, idea obtenida del curso de Programación Lógica dictado el semestre anterior. Este algoritmo, creado por John Von Neumann consiste en elegir el mejor movimiento posible, considerando que el oponente elegirá su mejor movimiento posible. Minmax puede ser aplicado a todo tipo de juego que pueda ser definido con una serie de reglas y premisas. Con ellas es posible saber, dado un punto en el juego, cuales son los movimientos posibles propios y del adversario.

Para evaluar el estado de un tablero y por lo tanto elegir la jugada que más favorezca al jugador actual, se genera un árbol con todas las jugadas posibles propias, seguidas sucesivamente por las jugadas posibles del oponente y nuevamente las propias. Esto bien podría seguir hasta que finalice el juego o hasta una profundidad predefinida en la implementación del algoritmo. Luego se procede a evaluar los tableros del último nivel u hojas, usando una función de evaluación previamente conocida. Se calcula el valor del resto de los nodos únicamente si se conoce el valor de sus hijos, es decir se evalúa desde las hojas hacia la raíz.

El método de evaluación de los nodos es el siguiente: Si se trata de un turno del oponente, el valor de ese nodo será el mínimo valor de sus nodos hijos. En caso de que sea el turno del jugador actual, será el máximo valor de sus nodos hijos. La explicación está en que el algoritmo supone que el oponente siempre realizará la jugada que, según la función de evaluación, más lo beneficie, es decir la que más perjudique al jugador actual. Por lo tanto tratará de escoger el tablero con menor evaluación. A su vez el jugador que aplica el algoritmo tratará de hacer lo opuesto y siempre buscar el tablero con mayor evaluación.

El supuesto principal es que el valor del tablero elegido, es el mismo que el valor del tablero al que se llegará si el oponente actúa como está previsto.

Alpha-Beta Prunning

Una de las desventajas de este algoritmo es la cantidad de tiempo que se invierte en construir el árbol de jugadas en el caso que las jugadas posibles sean demasiadas o se utilice un árbol de gran altura. El método que se utilizó para contrarrestar esta desventaja es el denominado Alpha-Beta Prunning, el cual mediante podas puede disminuir considerablemente la cantidad de nodos del árbol, mejorando así la performance del algoritmo.

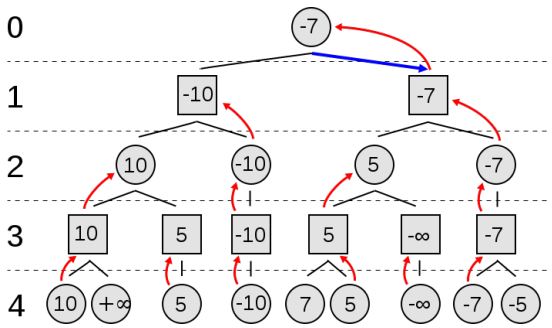

Figura 2: Árbol de evaluación MinMax

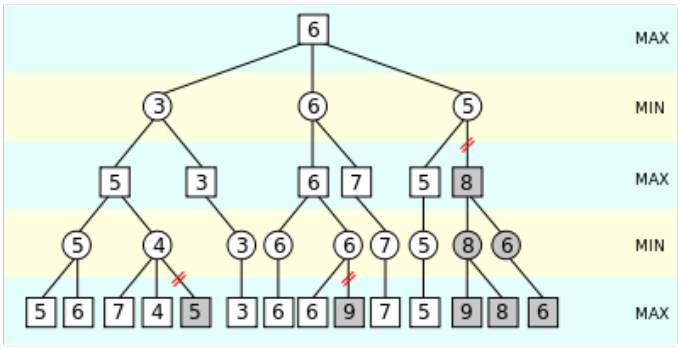

Figura 3: Poda Alfa-Beta

La poda alfa-beta utiliza dos parámetros que describen los límites sobre los valores que aparecen a lo largo de cada camino. Alfa es el valor de la mejor opción hasta el momento a lo largo del camino de la maximización, esto implica que se trata de la elección del valor más alto. Mientras que beta es el valor de la mejor opción hasta el momento a lo largo del camino de la minimización, esto implicará por lo tanto la elección del valor más bajo.

Esta búsqueda alfa-beta va actualizando el valor de los parámetros según se recorre el árbol. El método realizará la poda de las ramas restantes cuando el valor actual que se está examinando sea peor que el valor actual de α o β para la maximización o minimización, respectivamente.

El desarrollo del algoritmo en pseudocódigo será el siguiente:

Algorithm 1

```
1:  procedure ALPHA-BETA PRUNNING
2:      if nodo es hoja o profundidad = 0 then return false
3:      end if
4:      if Turno = Jugador.Turno then
5:          for Cada hijo del nodo do
6:              α := max(α, alfabeta(hijo, profundidad+1, α,β, Jugador))
7:              if β <= α then
8:                  Break;
9:              end if
10:         end for
11:         return α
12:     else
13:         for Cada hijo del nodo do
14:             β := min(β, alfabeta(hijo, profundidad+1, α,β, Jugador))
15:             if β <= α then
16:                 Break;
17:             end if
18:         end for
19:         return β
20:     end if
21: end procedure
```

Orden de jugadas

El algoritmo de alpha-beta prunning es sensible al orden en que evalúa las jugadas. Este algoritmo realiza una mayor cantidad de cortes y mas temprano en la ejecución cuando las jugadas son evaluadas en orden de evaluación decreciente.

En nuestro caso se aplicó un ordenamiento previo a las jugadas. Esto es, a cada nivel de minmax se realizó una evaluación de los tableros resultantes de cada movimiento. Cuanto mejor fuese la evaluación y más se aproximase al valor final del algoritmo, mejor sería el ordenamiento realizado y mas eficiente sería la poda.

En el peor de los casos, para un factor medio de branching de d, el algoritmo tendría el mismo desempeño que el minmax sin poda, con una ejecución de $O(b^d)$. Por el otro lado para un ordenamiento óptimo, la ejecución sería $O(b^d/2) = O(\sqrt{b^d})$. Cuando los nodos presentan un ordenamiento aleatorio, el tiempo de ejecución es aproximadamente $O(b^{3d/4})$.

Tablas de transposición

Las tablas de transposición son otra optimización implementada. En estas se almacenan los resultados de las búsquedas en profundidad. Se utiliza un caché LRU que se indexa mediante el tablero y se almacenan además la mejor jugada a emplear, el valor del tablero resultante de esta jugada y la profundidad a la que se ejecutó el minmax al resolver esta jugada.

Al momento de buscar un valor se busca mediante la clave y luego se compara con la profundidad restante en la búsqueda con la almacenada. Si la profundidad almacenada es menor, el resultado no es utilizado, en caso contrario, la función utiliza el valor almacenado. Cuando un nuevo tablero y sus jugadas son calculadas, se busca si el tablero ya se encuentra almacenado y de no ser así se almacena el tablero. Si este ya se encuentra en el caché, sólo es almacenado si la nueva profundidad es mayor o igual a la profundidad a la que fue calculado el elemento ya presente en el caché.

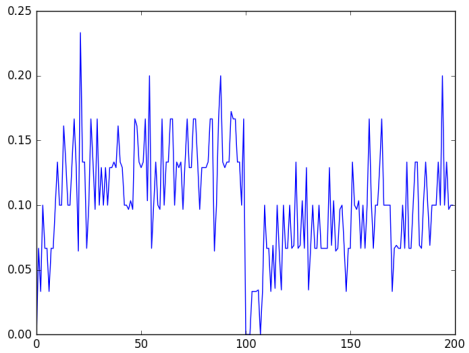

Figura 4: Hit ratio de las tablas de transposición

Es interesante notar la caída abrupta en la mitad de la ejecución. Esto corresponde al cambio del jugador que inicia la partida. Esto en cierta forma hace obsoletos, mientras no vuelva a iniciar el otro jugador, gran parte de los tablero aprendidos.

1.4. Bloques de la partida y smoothing

La bibliografía existente sobre el juego Othello sugiere que el juego posee tres estrategias diferentes definidos por tres bloques del juego. Estos son el *Inicio* de la partida, el *Medio* y el *Final* de la misma. Estos se se pueden identificar por la cantidad de fichas presentes y estas cantidades se distribuyen uniformemente.

Los primeras 20 jugadas pertenecen a la parte *inicial*, las siguientes 20 a la parte *media* y las siguientes, a lo sumo 20 más, pertenecen a la parte *final*.

A partir de estas características consideramos que sería interesante utilizar una red que se entrenase en desarrollar una estrategia para cada una de estas partes. De esta forma la estrategia a aprender por la red sería más homogénea que una sola gran red con el triple de neuronas.

Se evaluaron dos formas de realizar la transición entre las salidas de la red. Una de estas era aplicar una transición abrupta entre la salida de una u otra red según la fase de la partida en la que se encuentra el juego.

Para evitar una transición abrupta se empleó una función de suavizado que pondera las salidas de cada una de las redes.

$$V(t_i) = c_0(n)V_0(t_i) + c_1(n)V_1(t_i) + c_2(n)V_2(t_i)$$

$$c_i(n) = e^{-\frac{(n-\mu_i)^2}{\sigma^2}}$$

$$\sigma = 20, \mu_0 = 4, \mu_1 = 34, \mu_2 = 64$$

Como la suma de los coeficientes no da 1, además se aplicó una transformación para regularizar la salida.

$$V(t_i) = \frac{c_0(n)V_0(t_i) + c_1(n)V_1(t_i) + c_2(n)V_2(t_i)}{c_0 + c_1 + c_2}$$

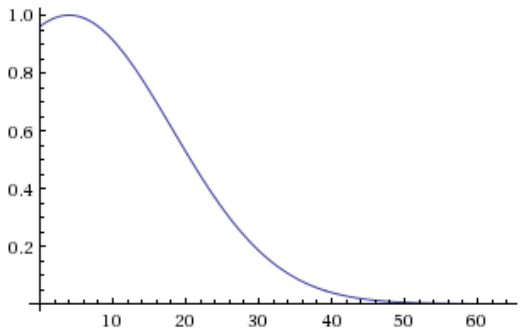

Figura 5: c_0 para $n \in [0, 64]$

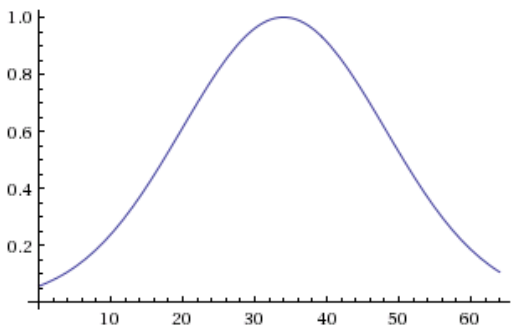

Figura 6: c_1 para $n \in [0, 64]$

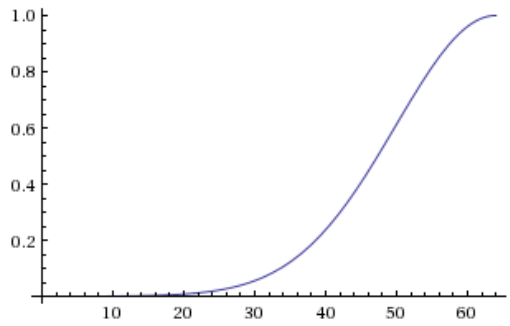

Figura 7: c_2 para $n \in [0, 64]$

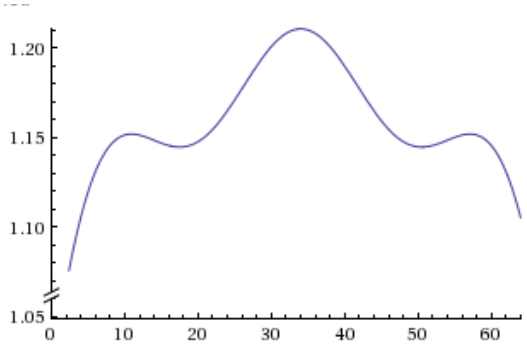

Figura 8: $c_0 + c_1 + c_2$ para $n \in [0, 64]$

1.5. $\epsilon - greedy$

Otra característica que se analizó, especialmente durante el entrenamiento es la capacidad de exploración de la red, particularmente la relación *exploración-explotación*. Para esto se agregó un parámetro $\epsilon \in [0, 1]$ que determina el comportamiento de la red en cada jugada.

Cuando al jugador le toca jugar, existe una probabilidad ϵ de que el jugador elija una jugada al azar en vez de la jugada mejor valorada.

Esto permite que el jugador explore el espacio de soluciones durante el entrenamiento, o incluso durante las partidas de evaluación.

Para que el jugador cambie al principio del entrenamiento por una exploración más agresiva hacia un juego mas *greedy* al final, se introdujo otro parámetro $\epsilon - factor \in [0, 1]$ que determina el decaimiento de ϵ durante las corridas.

Esto se empleó principalmente dado que el jugador presentado es un jugador determinista, y que un cierto grado de azar podrías ser provechoso.

2. Entrenamiento

2.1. Persistencia

En cuanto a la persistencia de los jugadores y su aprendizaje, lo que interesó persistir fue principalmente la red neuronal asociada. Por el otro lado, los datos de ϵ, $\epsilon - factor$, profundidad de Minmax, entre otros se cargaban dinámicamente en cada corrida.

De esta forma fue posible probar diferentes configuraciones además de las relacionadas directamente a la red sin necesidad de volver a entrenar el jugador.

Para la persistencia de la red se empleó el mecanismo de serialización de Python *pickle*. Este ofrece un mecanismo esencial de serialización y deserialización. Al ser un tipo de serialización binaria, no garantiza compatibilidad entre diferentes versiones de las librerías.

2.2. Jugadores

Random

Random juega de forma totalmente aleatoria. Fue el jugador predilecto empleado durante el entrenamiento, ya que permite explorar mejor el espacio de soluciones

Greedy

Greedy juega de forma agresiva, eligiendo los movimientos que mas fichas propias garantizan en el tablero. Es un jugador bastante previsible si se utiliza el algoritmo MinMax.

Positional

El jugador positional aplica una valoración del tablero según la ubicación de las fichas, asignándole un valor a cada posición y tomando el valor o su negativo según si la ficha es propia o del contrincante.

$$V(t) = \sum_{x=1..,8} \sum_{y=1..,8} v(t_{x,y}) T_{x,y}$$

$$T = \begin{matrix} 100 & -20 & 10 & 5 & 5 & 10 & -20 & 100 \\ -20 & -50 & -2 & -2 & -2 & -2 & -50 & -20 \\ 10 & -2 & -1 & -1 & -1 & -1 & -2 & 10 \\ 5 & -2 & -1 & -1 & -1 & -1 & -2 & 5 \\ 5 & -2 & -1 & -1 & -1 & -1 & -2 & 5 \\ 10 & -2 & -1 & -1 & -1 & -1 & -2 & 10 \\ -20 & -50 & -2 & -2 & -2 & -2 & -50 & -20 \\ 100 & -20 & 10 & 5 & 5 & 10 & -20 & 100 \end{matrix}$$

$$v(p) = \begin{cases} 0 & vaca(p) \\ 1 & propia(p) \\ -1 & \neg propia(p) \end{cases}$$

Cuando el tablero se encuentra ocupado en más del 80 %, el jugador cambia de estrategia y busca maximizar en todas sus movimientos la diferencia de fichas.

Mobility

Este jugador intenta maximizar la movilidad propia y disminuir la del oponente. Esto es, intenta maximizar la relación entre la cantidad de jugadas disponibles para él y la cantidad de jugadas disponibles por el contrincante.

De todas formas, al igual que para el jugador posicional, se le da prioridad a poseer las esquinas.

$$V(t) = 10(c_p - c_c) + \frac{m_p - m_c}{m_p + m_c}$$

c_p	Cantidad de esquinas capturadas propias
c_c	Cantidad de esquinas capturadas por el contrincante
m_p	Cantidad de jugadas posibles propias
m_c	Cantidad de jugadas posibles por el contrincante

Cabe destacar que la evaluación de movilidad (cantidad de jugadas) se realiza independientemente de quién sea el turno, de forma de realizarla en iguales condiciones.

Al igual que el jugador posicional, al alcanzar el 80 % del tablero ocupado, el objetivo pasa a ser maximizar la cantidad de fichas propias.

2.3. Torneo

Los torneos fueron la modalidad utilizada en el entrenamiento del jugador, se realizo con una implementación simple en la cual se ingresaba la cantidad de partidas que se jugarían por cada

enfrentamiento en cada ronda, la cantidad de rondas y los adversarios. Finalizada cada ronda se mostraban los resultados de los partidos mostrándonos así a grandes rasgos los porcentajes de éxito.

Creímos conveniente que se realizaran partidos tanto de blancas como de negras para tener un entrenamiento más competo, además rondas de no menos de 100 partidos y torneos de no menos de 100 rondas.

Además, los torneos eran parametrizables de forma de poder generar torneos de aprendizaje, donde se aplicaba backpropagation luego de la ejecución de una ronda para un par de jugadores y las redes resultantes eran almacenadas, o torneos de evaluación donde no se aplicaba aprendizaje.

Al comenzar cada torneo se cargan los jugadores, de existir, de uno o más archivos ".pkl". De no existir, se utilizaba la red pasada a la definición del jugador como red de partida y se la inicializaba.

Luego se procedía a ejecutar un número de enfrentamientos entre cada par de jugadores. Por ejemplo, de haber definido 100 partidas por enfrentamiento, y 3 jugadores aprendices A, B, C, la ronda consistiría en:

A vs B	A como blancas	100 partidas
A vs B	B como blancas	100 partidas
A vs C	A como blancas	100 partidas
A vs C	C como blancas	100 partidas
B vs C	B como blancas	100 partidas
B vs C	C como blancas	100 partidas

Se dividieron los jugadores en los grupos *Aprendices* y *Contrincantes*. Los aprendices jugarían contra los contrincantes y otros aprendices mientras que los contrincantes solo jugarían contra los aprendices.

Es decir que en un ronda de p partidas por enfrentamiento, a aprendices y c contrincantes, los contrincantes participarían en $2ap$ partidas, los aprendices en $2(a-1+c)p$. Se totalizarían $a((a-1)/2+c)p$ partidas en la ronda.

Luego de cada enfrentamiento jugando de cada lado del tablero, se almacenaría el estado de la red para los aprendices.

Por torneo, se ejecutarían tantas iteraciones de este procedimiento como rondas se hayan solicitado.

La ejecución del torneo estaría acompañada de información relevante en la salida estándar, así como gráficas al finalizar todas las rondas solicitadas.

2.4. Movimientos al azar

Dado el determinismo de los jugadores presentados (salvo el jugador Random), y obviando el uso de ϵ, se debió encontrar una forma de comparar los algoritmos.

Dado un mismo tablero de partida, los jugadores deterministas presentarían siempre el mismo juego. Por eso, en los torneos de evaluación de los jugadores deterministas un número inicial aleatorio de jugadas son realizadas aleatoriamente.

De esta forma se busca evaluar los algoritmos ante diferentes situaciones. De todas formas, el número de jugadas aleatorias es $n \in [0, 30]$ con lo que cabría una análisis más profundo de si esto favorece a los algoritmos que presentan una mejor estrategia para el juego medio y final.

3. Resultados

El entrenamiento para un jugador J se realizó enfrentándolo en 100 rondas de 100 partidas contra el jugador Random, totalizando 20000 partidas. Para aquellas redes que aún presentasen claros signos de continuar convergiendo rápidamente, se les permitió más iteraciones de este procedimiento.

Cuando fue posible, se emplearon redes ya entrenadas para comparar las variaciones en los jugadores. Por ejemplo, para la evaluación de los jugadores de tres redes con y sin función de smoothing se empleó la misma red.

Vamos a utilizar la notación $R[v_1, v_2]xN$ para referirnos a N redes con v_1 y v_2 neuronas en la primera y segunda capas ocultas. Para esta investigación N toma los valores $N = 1, 3$ de acuerdo a lo planteado a la sección de bloques de partida.

3.1. Aprendizaje

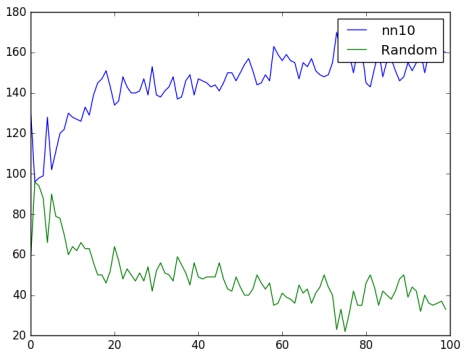

Figura 9: $R[10]x1$ durante 20000 partidas

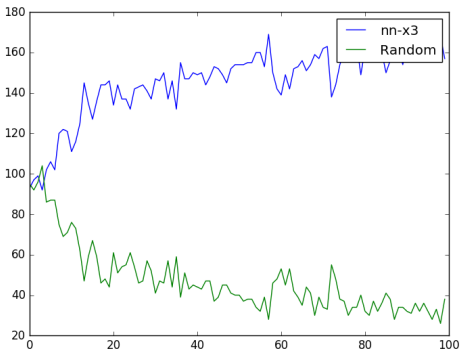

Figura 10: $R[10]x3$ durante 20000 partidas

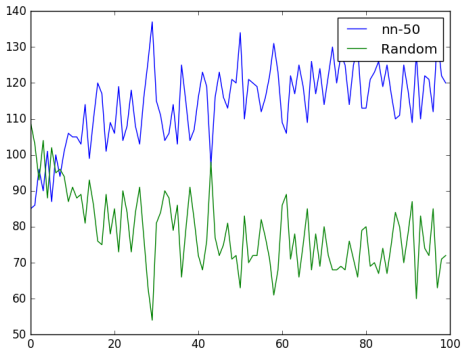

Figura 11: $R[50]x1$ durante 20000 partidas

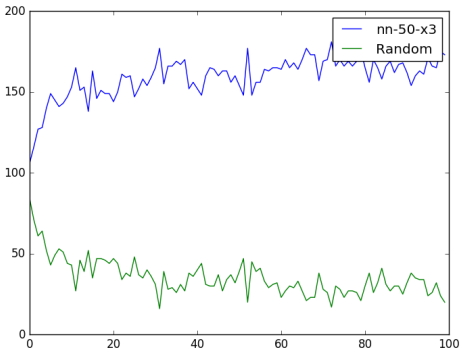

Figura 12: $R[50]x3$ durante 20000 partidas

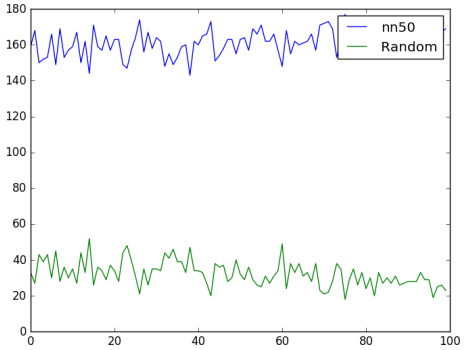

Figura 13: Siguientes 20000 partidas de $R[50]x1$

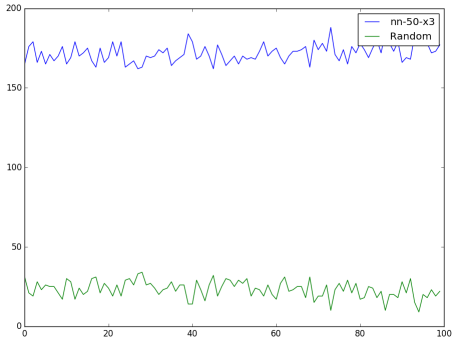

Figura 14: Siguientes 20000 partidas de $R[50]x3$

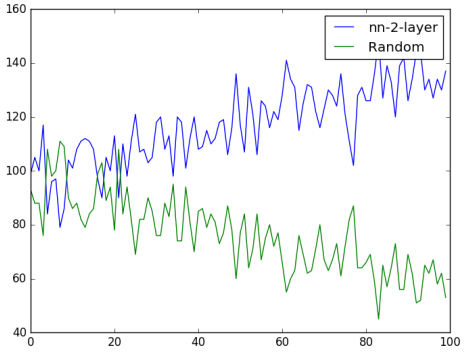

Figura 15: $R[10, 10]x1$ durante 20000 partidas

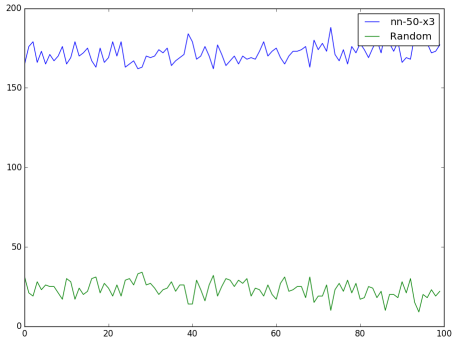

Figura 16: Siguientes 20000 partidas de $R[50]x3$

3.2. Evaluación

Para la evaluación se compararon los resultados no sólo contra el contrincante contra el que fue entrenado la red, Random, sino contra otros contrincantes con estrategias especificas de forma de evaluar la capacidad de juego del jugador desarrollado frente a otros escenarios y modalidades de juego especificas.

También se compararon entre sí diferentes configuraciones del jugador para decidirse para una configuración a presentar como competidor.

3.3. Entrega

A partir de los resultados obtenidos se armó un jugador a entregar. A este jugador se le incorporó la configuración minmax=3 y se realizaron dos torneos adicionales para comprobar la eficacia del algoritmo de minmax.

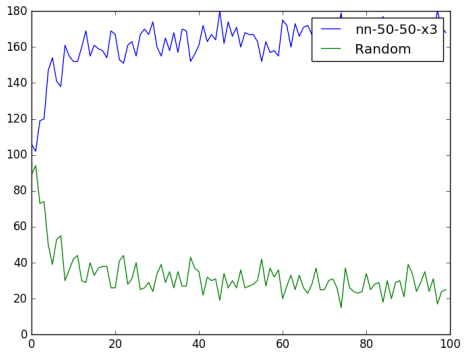

Figura 17: $R[50, 50]x3$ durante 20000 partidas

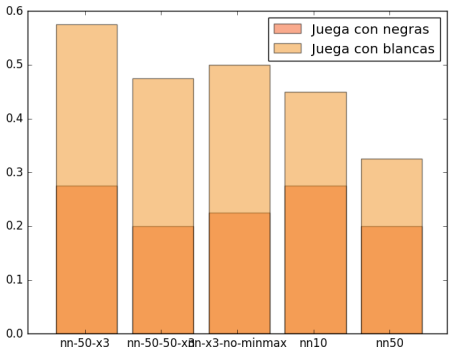

Figura 18: $R[50]x3$ enfrentado a otras configuraciones del jugador

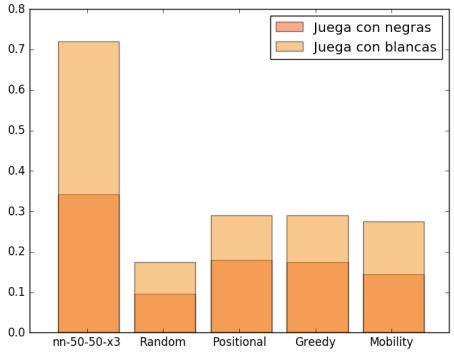

Figura 19: $R[50]x3$ enfrentado a otros jugadores deterministas con estrategias específicas

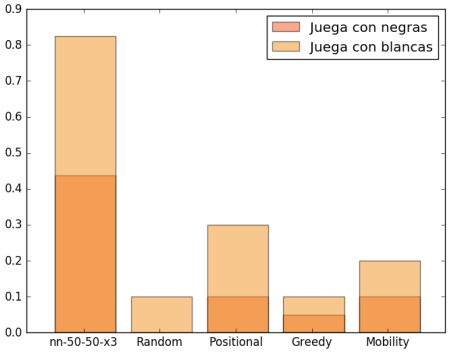

Figura 20: $R[50]x3$ con minmax

Referencias

[1] Anton V. Leouski and Paul E. Utgoff, *What a Neural Network Can Learn about Othello*, Department of Computer Science,University of Massachusetts

[2] Nees Jan van Eck, Michiel van Wezel,*Reinforcement Learning and its Application to Othello*, Econometric Institute, Faculty of Economics, Erasmus University Rotterdam

[3] Gerald Tesauro,*Temporal Difference Learning and TD-Gammon*, Communications of the ACM, March 1995 / Vol. 38, No. 3

[4] Volodymyr Mnih,Koray Kavukcuoglu,David Silver, Alex Graves,Ioannis Antonoglou,Daan WierstraMartin Riedmiller *Playing Atari with Deep Reinforcement Learning*, DeepMind Technologies

[5] M. Vuurboom *Learning Othello using Neuroevolution and Temporal Difference Learning*, Universiteit Utrecht