



Dpto. de Ciencias de la computación e
Inteligencia Artificial

CC
IA

Gonzalo Torres-Quevedo Acquaroni

Francisco Guijarro Grau

INDICE

1. Resumen del Artículo	3
2. Diseñar la Arquitectura de una red neuronal.....	5
3. Implementar una clase en Python	6
4. Implementar las clases y métodos necesarios	8
5. Repetir varias veces el proceso de aprendizaje.....	10
6. Documentación del trabajo	11

1. Resumen del Artículo

Anaconda Defeats Hoyle 6-0

El artículo comienza introduciéndonos en el juego de las damas, explicándonos como se juega. Primero se comentan las características del tablero después los movimientos principales de las fichas y por último como acaba el juego. También se menciona que ha habido estudios con redes neuronales para extraer información de cómo jugar a nivel experto.

En la siguiente sección (2) se nos explica cómo han intentado desde los años 60 crear un programa que pueda jugar a las damas e intentar derrotar a expertos en la materia. Se nombra la estrategia mini-max. Esta estrategia consistió en un autoaprendizaje mediante el cual un jugador competía contra otro. El perdedor era reemplazado por una variante determinista del ganador al alterar los pesos en las características que se usaron, o en algunos casos reemplazando características que tenían un peso muy bajo por otras características.

Esta estrategia tuvo éxito y junto con las redes neuronales es lo que más se tuvo en cuenta.

En el tercer punto ("Método evolutivo con redes neuronales para el juego de las damas desde cero") se explica el funcionamiento de un nuevo juego de damas implementado a partir de redes neuronales. En él, cada tablero está representado por un vector de longitud 32, donde cada componente corresponde a una posición en el tablero. Las componentes pueden tomar los siguientes valores: $\{-k, -1, 0, 1, k\}$ siendo $-k$ un rey del oponente, -1 una dama del oponente, 0 una casilla vacía, 1 una dame del jugador y k un rey del jugador. Un movimiento era determinado evaluando la presunta calidad de potenciales posiciones.

Esta red neuronal se implementó con una capa de entrada, tres capas ocultas y una capa de salida. La segunda y la tercera capa oculta tenían una estructura de conexión total mientras que las conexiones en la primera capa oculta estaban especialmente diseñadas para capturar posible información espacial del tablero.

Con tal de proporcionar información de proximidad, como por ejemplo si dos casillas eran vecinas, estaban muy cerca o muy separadas, se usó una arquitectura neuronal en la que la primera capa tenía 91 nodos y donde las capas ocultas recopilaban información de subdivisiones del tablero.

A continuación, se nos explica un modelo de algoritmo genético adaptado al juego de las damas en el que los tableros podían evolucionar según la salida de la red neuronal.

Por otro lado, también se explica cómo los juegos se desarrollaban usando una búsqueda en profundidad con el algoritmo minimax.

El proceso evolutivo fue iterado para 250 generaciones. Estas generaciones fueron clasificadas según su nivel (expert, master, grand master y class A).

Observando como jugaba el programa, se le denominó "Anaconda" ya que ganaba restringiendo la movilidad de sus oponentes.

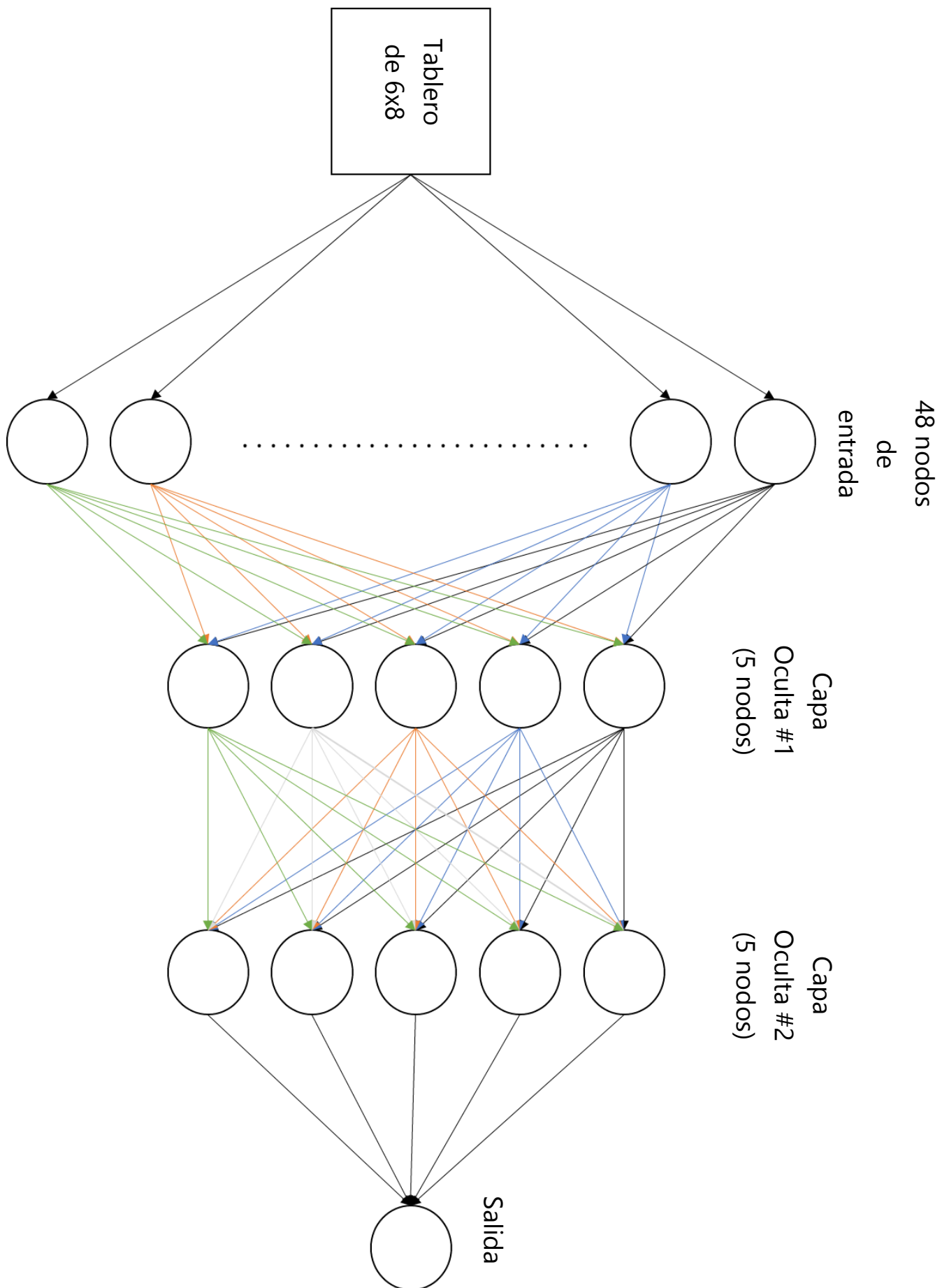
Anaconda añadió valor al mercado entre otras cosas por su característica de poder elegir el nivel de dificultad del programa por número de movimientos.

El experimento aquí expuesto demuestra la habilidad de un algoritmo evolucionado en ir aprendiendo generación tras generación como vencer a un oponente ya no solo humano sino también máquina. Y todo esto sin una información preprogramada en el juego.

Métodos utilizados:

- Método evolutivo con redes neuronales para el juego de las damas desde cero
- MiniMax

2. Diseñar la Arquitectura de una red neuronal



3. Implementar una clase en Python

```
# -*- coding: utf-8 -*-
#-----RED NEURONAL-----#
import numpy as np
from copy import deepcopy

class RedNeuronal:
    def __init__(self, neuronas_entrada, neuronas_oculta1, neuronas_oculta2,
                 neuronas_salida):

        self._n_entrada = neuronas_entrada
        self._n_oculta1 = neuronas_oculta1
        self._n_oculta2 = neuronas_oculta2
        self._n_salida = neuronas_salida

        #Se definen los pesos de manera aleatoria
        #Con rand generamos una matriz con valores entre 0 y 1
        #Primer parametro filas, segundo columnas

        self._w1 = np.random.rand(self._n_entrada, self._n_oculta1)
        self._w2 = np.random.rand(self._n_oculta1, self._n_oculta2)
        self._w3 = np.random.rand(self._n_oculta2, self._n_salida)

        #X es la matriz de entrada
    def activacion(self,x):

        self._z1 = np.dot(x, self._w1)
        self._a1 = self.sigmoid(self._z1)
        self._z2 = np.dot(self._a1, self._w2)
        self._a2 = self.sigmoid(self._z2)
        self._z3 = np.dot(self._a2, self._w3)

        salida = self.sigmoid(self._z3)

        return salida

    def sigmoid(self, z):

        return 1/(1+np.exp(-z))

    def pesosEnLista(self):

        w1 = self._w1.tolist()
        w2 = self._w2.tolist()
        w3 = self._w3.tolist()
        res = []

        for i in range(48):
            for e in range(5):
                res.append(w1[i][e])
        for x in range(5):
            for z in range(5):
                res.append(w2[x][z])
        for w in range(5):
            res.append(w3[w][0])
        return res
```

```

def actualizaPesos(self, lista):
    l1 = []
    l2 = []
    l3 = []
    listaAux = deepcopy(lista)

    for i in range(48):
        aux = []
        for e in range(5):
            x = listaAux[0]
            aux.append(x)
            listaAux.pop(0)
        l1.append(aux)

    self._w1 = l1

    for elem in range(5):
        aux = []
        for y in range(5):
            x = listaAux[0]
            aux.append(x)
            listaAux.pop(0)
        l2.append(aux)

    self._w2 = l2

    for elem in range(5):
        aux = []
        x = listaAux[0]
        aux.append(x)
        listaAux.pop(0)
        l3.append(aux)

    self._w3 = l3

```

Con esta clase podemos crearnos redes neuronales con unos pesos aleatorios, pero también podemos actualizar dichos pesos con el método `actualizaPesos()`.

También podemos obtener una salida de la red neuronal mediante su función de activación (`activación()`) y haciendo uso de la función `sigmoide` (`sigmoide()`).

4. Implementar las clases y métodos necesarios

```
class HeuristicaNeuronal(Heuristic):

    instance = None

    def __init__(self, redNeuronal):
        self._redNeuro = redNeuronal

    def __new__(cls, *args, **kwargs):
        if cls.instance is None:
            cls.instance = object.__new__(cls, *args, **kwargs)
        return cls.instance

    def heuristic(self, state, player):

        tablero = state.board
        v = [tablero[0]+tablero[1]+tablero[2]+tablero[3] +tablero[4]+tablero[5]]
        if (state.winner==None):

            return self._redNeuro.activacion(v)

        if (state.winner==player):

            return 1000000

        return -1000000
```

Con esta función heurística conseguimos que dos redes neuronales puedan jugar partidas de Conecta 4.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jan 26 20:27:34 2018

@author: Paco-Lee & Gonzalo

"""
import AG
import random
import Red_Neuronal
import conecta4

mejorC = [#lista de numeros aleatorios entre -1 y 1 de tamaño 270]
#esta variable era demasiado larga para meterla en el documento
```



```

def fitnessNeuronal(cromosoma):

    redNeuronal = Red_Neuronal.RedNeuronal(48,5,5,1)
    redNeuronal.actualizaPesos(cromosoma)

    redNeuronal2 = Red_Neuronal.RedNeuronal(48,5,5,1)
    redNeuronal2.actualizaPesos(AG.mejorC)

    ganador = conecta4.compareHeuristics (conecta4.HeuristicaNeuronal(redNeuronal),
                                           conecta4.HeuristicaNeuronal(redNeuronal2),3,True)
    if ganador == 0:
        return 0
    elif ganador == 1:
        AG.mejorC = cromosoma
        return 1000000
    else:
        return -1000000

def AlgoritmoGenetico(tamPoblacion):

    lista = []

    for i in range(tamPoblacion):
        lista.append(p())
    for e in range(tamPoblacion):
        resultado = fitnessNeuronal(lista[e])
        print(AG.mejorC)
        if resultado == 0:
            mutaCromosoma(mejorC)

    return AG.mejorC

def setGen(i,cromosoma,gen):
    cromosoma[i] = gen

def mutaCromosoma(cromosoma):
    for i in range(0,len(cromosoma),2):
        setGen(i,cromosoma,random.uniform(-1,1))

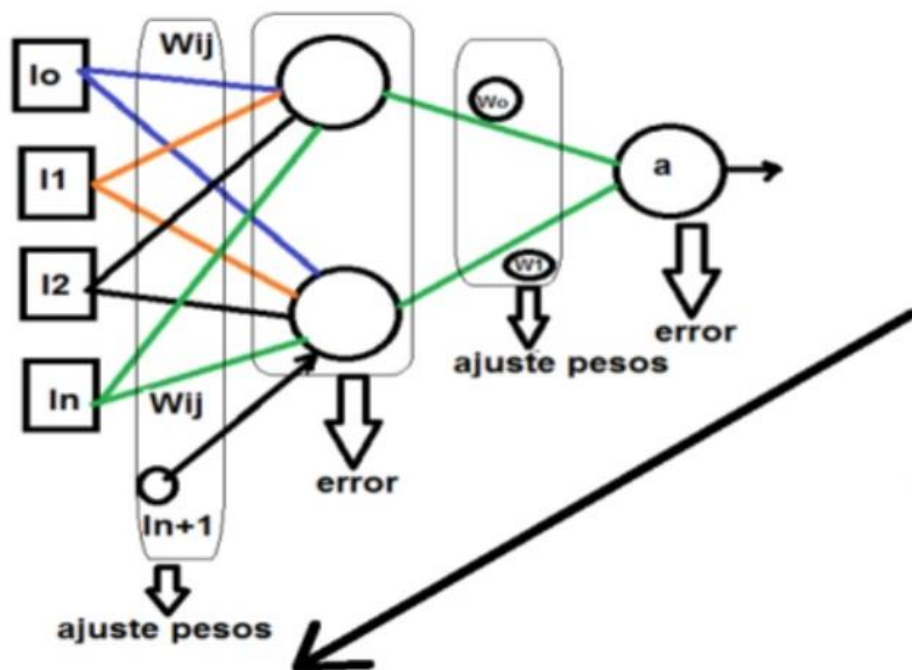
```

Con estos métodos conseguimos por un lado implementar una función de fitness que se vaya quedando con el mejor cromosoma de pesos para una red neuronal. Esto lo hace con el método `compareHeuristics()`. Por otro lado, tenemos un método para crearnos un algoritmo genético que mediante un tamaño de población nos cree varios individuos y los vaya enfrentando y mutando.

5. Repetir varias veces el proceso de aprendizaje

Aquí nos hemos encontrado con un problema y es que a veces, cuando ejecutamos el algoritmo genético y se juegan las partidas, estas son la mayoría iguales y el cromosoma ganador es el mismo. Creemos que puede ser debido a que no se este actualizando el cromosoma ganador.

En la parte de resultados del punto 6 de este trabajo se pueden ver dos ilustraciones de este proceso de aprendizaje.



6. Documentación del trabajo

Algoritmos genéticos y redes neuronales para jugar al Conecta-4

Inteligencia Artificial (Tecnologías de la Información)

Curso 2017/2018

Departamento de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Abstract – Hemos hecho una investigación sobre cómo conseguir que una Red Neuronal sea lo más inteligente posible a la hora de jugar al Conecta-4, para ello primero nos hemos tenido que crear una Red Neuronal, cuando conseguimos que la Red funcionara, nos creamos una Heurística a la que le pasamos nuestra Red. Con esto pudimos enfrentar dos Redes Neuronales entre sí. Posteriormente nos creamos una función fitness para mediante un Algoritmo Genético enfrentar una población de Redes Neuronales y quedarnos con la mejor. Por último, nos creamos un Algoritmo Genético que pasándole un tamaño de población nos permitió hacer esta selección.

INTRODUCCION

Al Conecta-4 se juega tradicionalmente en un tablero 6x7 (aunque nosotros en este trabajo tratamos un tablero 6x8) en el que participan dos jugadores. Suele ser con fichas rojas y amarillas para diferenciar a cada oponente y el objetivo es hacer un 4 en línea, es decir, que 4 fichas del mismo color estén alineadas horizontalmente, verticalmente o en diagonal.

Las reglas del juego son muy sencillas, únicamente se trata de respetar los turnos de los jugadores y que solo podemos colocar una ficha por turno.

Nuestro esfuerzo actual explora el potencial de usar Redes Neuronales para extraer información sobre cómo jugar al Conecta-4 de la mejor manera posible, mediante un aprendizaje no supervisado.

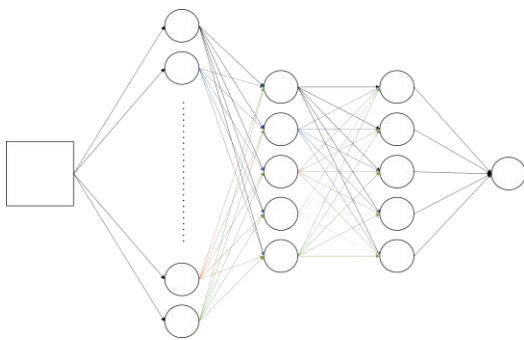
Usando un procedimiento coevolucionario, hemos intentado conseguir que una población de Redes Neuronales vaya evolucionando unas listas de pesos y nos da la mejor que sería una partida de nivel experto.

Ahora vamos hablar sobre la Metodología, los Resultados, la Conclusión y finalmente las Referencias que hemos tomado para este trabajo.

METODOLOGIA

Para este trabajo hemos utilizado la herramienta Python que nos ha permitido implementar las funciones, clases y métodos necesarios para resolver el problema planteado.

Primero tuvimos que plantear un diseño de nuestra Red Neuronal, que tras muchos diseños decidimos dejarla así.



Tenemos un tablero 6x8 en el que recibimos 48 nodos de entrada que corresponden con cada hueco libre que hay en él.

Luego hemos añadido 2 capas ocultas al diseño con 5 nodos cada una de ellas para realizar un procesamiento más detallado de los datos de la capa de entrada y proporcionarnos una función de activación más precisa.

Por último, la salida de la red neuronal puede ser diferentes cosas, pero como en nuestro caso hemos usado la función sigmoideal como función de activación nuestros valores estarán comprendidos entre 0 y 1.

Siendo los valores desde 0 hasta 0´4999 la victoria del ganador 1, el valor 0´5 sería un

empate y los valores hasta 1 la victoria del jugador 2.

Nuestra metodología se ha basado en pruebas y errores con lo que íbamos programando lo que nos hacía falta en cada momento y probando poco a poco si funcionaba.

Una parte importante de la metodología también ha sido reunirnos con nuestros demás compañeros y profesor y debatir sobre el trabajo ya que al principio era difícil de ver todo lo que se quería conseguir y hasta donde había que llegar.

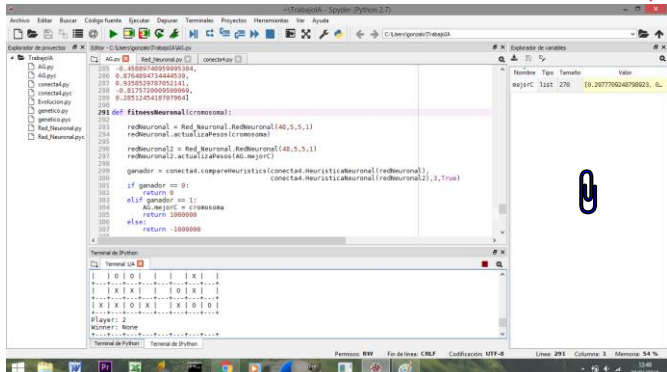
RESULTADOS

Tras comprobar que nuestra Red Neuronal funcionaba correctamente mediante la consola del *Spyder* (Programa usado para la implementación del trabajo), tocaba crear nuestra propia Heurística tomando como referencia una de las que ya estaba creada en el código conecta-4 [3] añadiéndole las líneas de código necesarias para "pasarle" nuestra Red Neuronal.

Primero enfrentamos nuestra heurística con las dos que ya había creadas y posteriormente pudimos enfrentar dos de las nuestras. Después implementamos un algoritmo genético que creando una población de individuos fuera enfrentando heurísticas y así nos quedamos con el mejor cromosoma.

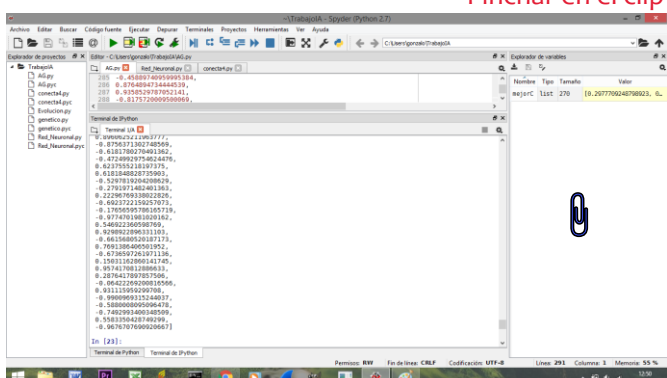
Esta captura representa un momento del entrenamiento que una población de redes neuronales esta realizando para ser la mejor en el Conecta 4.

Pinchar en el clip



Esta otra imagen nos muestra el mejor cromosoma tras la ejecución de las partidas mediante nuestro algoritmo genético.

Pinchar en el clip



puede obtener un conocimiento no supervisado.

Siendo esto así el proceso de aprendizaje no es preciso del todo.

REFERENCIAS

1. Anaconda Defeats Hoyle 6-0: A Case Study Competing an Evolved CheckersProgram against Commercially Available Software
2. https://en.wikipedia.org/wiki/Connect_Four
3. <https://github.com/Ignacio-Perez/connect-four>
4. <https://en.wikipedia.org/wiki/Minimax>
5. <https://www.cs.us.es/cursos/aia-2016/temas/tema-03.pdf>
6. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3474301/>
7. <http://pabloem.github.io/Programando-una-red-neuronal/>
8. <https://www.youtube.com/watch?v=fP762NAkGu8>
9. <https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/>

CONCLUSIONES

Con el desarrollo de este trabajo hemos mejorado nuestras habilidades como ingenieros y programadores.

A través del juego Conecta 4 y con el diseño de varios métodos y clases hemos visto como mediante redes neuronales se