



Desarrollo de aplicaciones multiplataforma

Módulo Proyecto

[Título del proyecto]

Autor/es: Samuel Hervás González

Tutor/es: Roberto Fernández

Madrid, Mayo 2026

Resumen

Agradecimientos

Índice

Listado de abreviaturas, siglas y acrónimos

Inteligencia artificial: Rama de la informática que desarrolla sistemas capaces de imitar funciones cognitivas humanas como el aprendizaje, razonamiento, la percepción y la resolución de problemas

Machine learning: Subdisciplina de la IA que utiliza datos para enseñar a las computadoras y máquinas a aprender por sí solas

Deep learning: Subcampo del Machine Learning que emplea redes neuronales para aprender

Red neuronal: Un conjunto de nodos enlazados con pesos usados para analizar los datos

Nodo / Neurona: Unidad básica de una red neuronal que recibe entradas, realiza un cálculo y produce una salida. Simula de forma simplificada el funcionamiento de una neurona biológica.

Peso: Valor numérico que indica la importancia de una conexión entre neuronas. Determina cuánto influye una entrada en el resultado.

Bias: Valor adicional que se suma al cálculo de la neurona para ajustar la salida y dar mayor flexibilidad al modelo.

Función sigmoide [SIGM]: Esta es una función que convierte un número de $-\infty$ a ∞ a un número del rango de 0 a 1

Derivada función sigmoide: Es la derivada de la función sigmoide, es decir, la tasa de cambio instantánea de la función sigmoide

Alimentación hacia adelante: Proceso mediante el cual la información avanza desde la capa de entrada hasta la capa de salida de la red neuronal, pasando por las capas intermedias, sin retroceder.

Retropropagación: Algoritmo que cambia, con los resultados de unos datos, todos los pesos de la red neuronal, haciendo que las respuestas de esta sean más acertadas respecto a esa prueba

Mini-batch: Optimización del algoritmo de aprendizaje. Divides todos los datos de aprendizaje en subconjuntos de un número de muestras

Aprendizaje supervisado: Tipo de aprendizaje de las redes neuronales, en el que tienes los datos de aprendizaje. Es usado para problemas de clasificación

Aprendizaje no supervisado: Tipo de aprendizaje de las redes neuronales, en el que no tienes los datos de aprendizaje. Es usado para entender la estructura de datos

Aprendizaje por refuerzo: Tipo de aprendizaje de las redes neuronales. Este consta de un agente (inteligencia artificial) que interactúa con entorno, y no sabe lo que tiene que hacer. Cuando hace algo correctamente, aprende de estos. Es usado para optimizar el comportamiento de este agente. Un ejemplo puede ser una IA que aprenda a jugar a juegos de estrategia como ajedrez o Go

NPC: Personaje no jugable

1 Introducción

[Además de la introducción en sí, se puede hablar, de manera resumida, los apartados que va a tener el documento.]

La Inteligencia Artificial se ha convertido en una herramienta presente en prácticamente todos los ámbitos de la vida cotidiana. Puede usarse de forma consciente, como mediante asistentes virtuales como Gemini, Claude o Alexa, pero también de forma inconsciente, como ocurre en los algoritmos de recomendación de plataformas digitales, en los motores de búsqueda o en el comportamiento de personajes de videojuegos.

2 Planteamiento del problema

[Aquí se describe el problema, si es una mejora de un producto (software, hardware u otros) y/o hay una necesidad de mejora de algo.]

Esta amplia intervención de la inteligencia Artificial en nuestras vidas genera una necesidad de comprender el funcionamiento de una Inteligencia Artificial funcional.

Por otro lado, la Inteligencia Artificial es un concepto tan amplio que no se comprende en su totalidad, solo a escala muy pequeña y de manera generalizada. En base a lo expuesto surgen las siguientes dudas:

¿Cómo funcionan las redes neuronales? ¿Cómo consiguen resolver a partir de un mismo algoritmo diferentes problemas? ¿Cómo son capaces de encontrar patrones eficientes?

3 Justificación. Beneficiarios

[¿Por qué realizamos este proyecto o esta investigación? ¿Quién se beneficia?]

Crear una red neuronal ayuda a comprender el funcionamiento interno de las redes neuronales, tanto la propagación hacia delante como la retropropagación, en lugar de usar librerías externas y considerar las redes neuronales como una caja negra.

Dicho de otra manera, se busca analizar y comprender en profundidad las redes neuronales y cómo procesan información, ajustan sus parámetros y aprende a resolver distintos problemas

Este proyecto es público, permitiendo así que cualquier persona interesada pueda consultar y aprender en base a este código el funcionamiento interno de las redes neuronales

4 Objetivos

[Responde a qué esperamos conocer o conseguir según el planteamiento del problema. Deben ser concretos y realistas.]

Los objetivos se formulan en infinitivo. Ejemplos pueden ser: desarrollar, diseñar, construir, medir, evaluar, decidir o identificar, entre otros.]

- Desarrollar un programa desde cero en Java sin uso de librerías externas que pueda diseñar redes con diferentes funcionalidades, dependiendo de los datos de entrenamiento que se le otorgue
- Diseñar con este programa una red neuronal que identifique cualquier dígito dibujado a mano en una matriz de 28 x 28 píxeles en una escala de grises
- Simplificar el código para su fácil comprensión, de cara a que pueda servir de ayuda a cualquier persona que quiera aprender sobre el funcionamiento de una red neuronal
- Conseguir la máxima eficiencia del programa, teniendo en cuenta tanto el rendimiento como el número de recursos usados por el programa
- Lograr la mayor flexibilidad posible, permitiendo al usuario añadir y eliminar pesos en la red neuronal
- Obtener un ratio de aciertos superior al 90% en pruebas de números no utilizados para la fase de entrenamiento

5 Requisitos técnicos y legales

[Desglosados en requisitos técnicos y legales. ¿Qué requisitos, técnicos y/o legales, existen para poder llevar a cabo el proyecto? ¿Se aplica la prevención de riesgos laborales?]

El proyecto debe cumplir las normativas relacionadas con el uso de datos, derechos de autor y protección de la privacidad.

En este proyecto los números utilizados como datos de entrenamiento han sido previamente anonimizados y cumplen por lo tanto con el reglamento general de protección de datos.

6 Ayudas y subvenciones económicas

[Como mecanismos de ayuda en la financiación del proyecto.]

No ha sido necesario ningún tipo de ayuda ni subvención económica

7 Viabilidad

[Técnica, económica y legal. ¿Podremos alcanzar los objetivos y/o materializar el proyecto? ¿Puede ser rentable económicamente?]

Este proyecto es viable en el aspecto técnico, ya que existen muchos lenguajes de programación, librerías, recursos y documentación suficientes para desarrollar este proyecto

Respecto a la viabilidad económica, el proyecto no tiene ningún coste económico, ya que el lenguaje, la documentación y las bases de datos usadas son completamente gratuitas.

El proyecto es legalmente viable ya que se respeta la normativa relacionada con el uso de datos, derechos de autor y protección de la privacidad.

8 Recursos

[Tanto humanos como materiales.]

- Portátil con Java 23 instalado
- Base de datos MNIST con 60.000 datos (50.000 de ellos usados para el entrenamiento, 10.000 para las pruebas) de números dibujados a mano. Cada número tiene que estar en una matriz de 28x28 en una escala de grises. La base de datos está desarrollada por NIST, la Universidad Nacional de Tecnología y Estandar de Estados Unidos
- Tiempo de investigación, programación del proyecto, y aprendizaje de distintas redes neuronales

9 Planificación

[Dentro de la planificación de un proyecto es necesario especificar las actividades (tareas) de manera ordenada. Para ello puede resultar útil el uso de alguna herramienta, como puede ser un diagrama de GANTT.]

1. Investigación y análisis teórico:
 - a. Investigar qué son las redes neuronales
 - b. Estructura de una red neuronal y funcionamiento de estas
 - c. La propagación hacia delante.
 - d. La retropropagación.
 - e. Analizar los diferentes tipos de aprendizaje (supervisado, no supervisado y aprendizaje por refuerzo) y decidir cuál de estos tipos de aprendizaje usar
2. Crear un proyecto de Java, que contenga
 - a. Clase abstracta llamada nodo (Node), con 3 clases que hereden de esta (Input, Connection y Output)
 - b. Clase de red neuronal (NeuralNetwork)
 - c. Clase NeuralMath, para los cálculos extra
3. Crear una interfaz gráfica con un panel de 28x28 píxeles en la que el usuario pueda escribir un número del 0 al 9 y pasar a la red neuronal este para que intente adivinar qué número cree que es.

10 Presupuesto económico

[Para ello se tendrán en cuenta los apartados anteriores.]

0

11 Documentación técnica¹

[En el caso de proyectos software pueden incluirse, entre otros:

- Requisitos (funcionales, no funcionales y de bases de datos)
- Arquitectura
- Diseño
- Implementación
- Pruebas.]

11.1 Requisitos

11.1.1 Requisitos funcionales

[Requisitos que describen las funcionalidades que debe tener el sistema.]

Capacidad de crear redes neuronales que aprendan por sí mismas con los datos que se le aporten para entrenar.

Las redes neuronales creadas en este proyecto identificarán números del 0 al 9 dibujados a mano en una matriz 28x28 con una escala de grises.

La red neuronal recibirá un array de tamaño 784 y calculará el resultado, en este caso, el número que está dibujado

11.1.2 Requisitos no funcionales

[Ejemplos pueden ser, entre otros: usabilidad, eficiencia, rendimiento, tecnologías, seguridad.]

Alto rendimiento y eficiencia: Muy bajo tiempo de respuesta para procesar la matriz y arrojar la predicción, siendo este inferior a 0,5ms.

Elevada precisión o tasa de acierto, superior a 95%. El sistema es capaz de reconocer correctamente los dígitos trazados a mano.

¹ Este apartado puede denominarse “Especificación técnica” en el caso de proyectos software.

La escalabilidad de este programa es alta, ya que, para hacer la red neuronal más grande, solo es necesario añadir más capas y nodos, teniendo un límite de 32.767 capas y 2.147.483.647 nodos por capa. Además, es aplicable a otros problemas de las redes neuronales, aunque sería recomendable crear una red neuronal distinta para cada problema.

Alta usabilidad. Con este programa es posible crear una red neuronal con diferentes funcionalidades de una manera muy simple.

11.1.3 Requisitos de bases de datos

[Pueden contener, según el caso, esquemas descriptivos, conceptuales, lógicos y físicos.]

La base de datos utilizada en este proyecto es MNIST. Contiene números del 0 al 9 trazados a mano. Cada dato contiene los 784 píxeles de la matriz y el número que representa. En este caso, hago uso de la base de datos MNIST, compuesta por 60.000 muestras.

11.2 Arquitectura y marco teórico

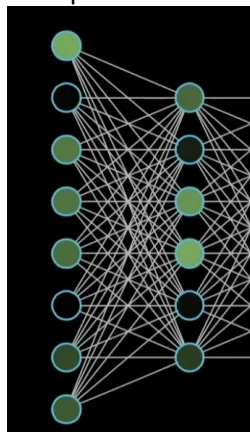
Una red neuronal está compuesta de nodos, también llamadas neuronas, organizadas por capas, y de conexiones. Hay 3 tipos de nodos:

- Nodos de entrada que reciben los datos.
- Nodos que sirven como conexiones entre diferentes capas y están configurados por unos pesos.
- Nodos encargados de generar el resultado final.

Una red neuronal se conecta por capas, y las neuronas de las capas se conectan con pesos. Los nodos sólo se pueden conectar con las capas de detrás y de delante.

La inteligencia artificial realiza una alimentación para delante, y luego devuelve un resultado en los outputs. La alimentación para delante funciona de la siguiente manera:

- La red neuronal recibe los datos de Input
- Cada nodo de la capa de input se envía a los nodos de la siguiente capa, multiplicando el valor del nodo por el peso correspondiente



- A continuación, se le suma el valor del bias de cada nodo de la capa de delante y se le aplica a cada uno de estos valores la función sigmoide. Esta función lo que hace es convertir un número cualquiera en un número en el rango entre 0 y 1. He escogido como función sigmoide la siguiente [SIGM]:

$$\frac{1}{1 + e^{-n}}$$

- Estos pasos se repiten en cada capa de la red neuronal, hasta llegar a la capa con los nodos de Output, en los que también se suma el valor del bias y se le aplica a cada uno de estos valores la función sigmoide
- Finalmente, la función devuelve un array con todos los datos. El último array contiene la última capa, es decir, los resultados. La inteligencia artificial no puede devolver un número del 0 al 9, ya que después se debe interpretar el output. En nuestro caso, el valor que la inteligencia artificial piensa que el que está dibujado será el valor más activado entre todos los outputs.

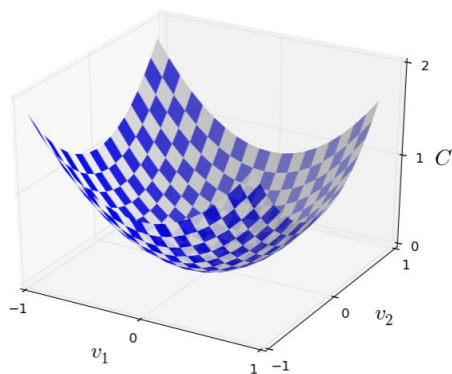
El proceso de aprendizaje se divide en 3 partes:

- 1) Alimentación para delante
- 2) Retropropagación
- 3) Cambio de pesos y biases

Para que una red neuronal aprenda, hay 3 métodos principales usados:

- **Aprendizaje supervisado:** En este tipo de aprendizaje, tienes los datos de aprendizaje. Es usado principalmente para problemas de clasificación
- **Aprendizaje no supervisado:** En este tipo de aprendizaje no tienes los datos de aprendizaje. Es usado para entender la estructura de datos
- **Aprendizaje por refuerzo:** Este consta de un agente (inteligencia artificial) que interactúa con un entorno, sin saber lo que debe de hacer. Al realizar una acción correctamente, se le recompensa con una función y aprende. Es usado para optimizar el comportamiento de este agente. Un ejemplo puede ser una IA que aprenda a jugar a juegos de mesa de estrategia como ajedrez o Go

Para explicar la retropropagación se utiliza el descenso de gradiente. Lo que presenta esta idea es minimizar el problema como una función. Para mayor facilidad, se simplifica la función con únicamente 2 variables.



La idea del descenso de gradiente es encontrar qué punto tiene menor coste, en nuestro caso de redes neuronales, menor margen de error. Esto se puede lograr seleccionando un lugar aleatorio de la gradiente, y viendo hacia qué dirección se desciende el coste de la función. Una manera simple de visualizar esto es imaginarse una pelota en la función y, viendo dónde cae, imaginarnos hacia qué dirección rodaría la pelota. Este método, en funciones más complejas puede no ayudarnos a encontrar el coste mínimo, pero puede ayudar a minimizar el margen de error.

La retropropagación se puede dividir en 2 partes

- a. Cálculo del margen de error en los nodos
- b. Cálculo del margen de error en cada uno de los pesos y biases

y funciona de la siguiente manera:

- Se obtiene el margen de error de los outputs, es decir, la diferencia entre el output (en el último valor de values) y el dato que queríamos que tenga realmente (el array expectedData). A continuación, se añade al margen de error la derivada de la función sigmoide de cada nodo, que necesitaremos para un paso más adelante.
- Seguidamente, con los datos del margen de los nodos output, se va calculando el margen de error de los nodos anteriores. La manera de calcularlo no es como en la capa de los outputs, es sumando cada peso por el valor del nodo al que llevaba ese peso, para obtener el valor de cuánto afectó ese nodo al cálculo del peso. Después, como en el margen de error de los outputs, se le multiplica la derivada de la función sigmoide. Este cálculo es necesario para todas las capas, excepto para la última, la de los Inputs, en la que no es necesario.
- Una vez obtenidos los márgenes de error, se calcula el cambio de los pesos para que se acerque al dato esperado
- Finalmente, se devuelven estos valores para luego cambiar los pesos y biases.

Estos procesos se repiten muchas veces hasta que la red aprende patrones útiles y minimiza el error.

Para aumentar la eficiencia, se puede dividir los casos de prueba en fragmentos de tamaño n , y luego verificar los cambios de cada uno, guardando esos datos y luego sumando todos los cambios de ese fragmento a la vez. Esto aumenta la velocidad de aprendizaje, ya que reduce el número de cálculos necesarios.

Un método usado para mejorar los resultados es disminuir la tasa de aprendizaje respecto a la cantidad de iteraciones realizadas y respecto al margen de error respecto a la primera iteración. El objetivo es aumentar la precisión para llegar a un mayor margen de error. Otro método usado es mover el panel de los casos unos píxeles hacia una dirección aleatoria, permitiendo que identifique mejor los patrones.

Por último, se cambian los valores de los pesos y biases.

11.3 Diseño

11.3.1 Diseño de la red neuronal

El programa se estructura en varias clases principales:

a) Clase Nodo

Se desarrolla una clase base denominada `Nodo`, encargada de representar el comportamiento general de una neurona artificial.

De esta clase heredan otras clases especializadas:

- `InputNode`: nodos de entrada que reciben los datos.
- `ConnectionNode`: conexiones entre neuronas y almacenamiento de pesos.
- `OutputNode`: nodos encargados de generar el resultado final.

Esta estructura permite organizar de forma modular el funcionamiento interno de la red neuronal.

La red neuronal, al inicializarla, conecta todos los nodos a los nodos de la siguiente capa. La red neuronal recibe un conjunto de datos (input) y devuelve unos valores (output)

Tanto los `InputNode` como los `ConnectionNode` tienen 2 `ArrayList`. Uno muestra el ID del nodo de la siguiente capa al que está conectado y el otro muestra el valor de la conexión.

b) Clase `NeuralNetwork`

La clase `NeuralNetwork` es la encargada de administrar toda la estructura de la red neuronal.

Funciones principales:

- gestionar las capas de neuronas,
- realizar la propagación hacia delante,

- ejecutar la retropropagación y actualizar pesos,

c) Clase NeuralMath

Se implementa una clase auxiliar denominada NeuralMath, destinada a centralizar los cálculos matemáticos necesarios para el funcionamiento de la red neuronal.

Esta clase incluye operaciones como:

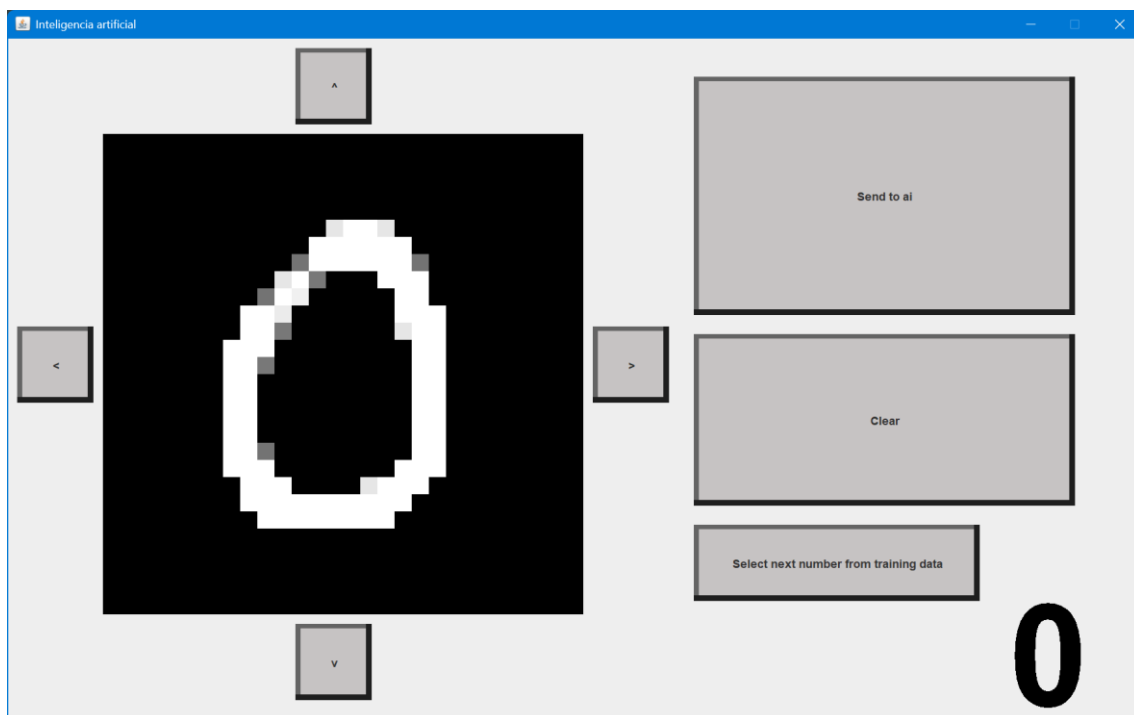
- funciones de activación
- derivadas
- otras funciones auxiliares

11.3.2 Diseño de la interfaz gráfica

Finalmente, se desarrolla una interfaz gráfica que permite interactuar con la red neuronal de forma visual.

La interfaz incluye:

- un panel de dibujo donde el usuario puede escribir números manualmente,
- un sistema para convertir el dibujo en datos interpretables por la red neuronal,
- una sección donde se muestra el número que la inteligencia artificial considera más probable.



El objetivo de esta interfaz es facilitar la visualización práctica del funcionamiento de la red neuronal y permitir comprobar cómo el sistema interpreta patrones dibujados por el usuario.

11.4 Implementación

11.4.1 Estructura general

La red neuronal funciona de la siguiente manera:

Existe una clase llamada `NeuralNetwork`, con 3 variables:

- `Node[][] nodes`: es un array bidimensional, en el que se añaden los nodos. Cada array es una capa, y cada capa contiene un número de nodos.
- `short shapeTotal`: número de capas de la red neuronal. No se utiliza el valor `nodes.length` por temas de eficiencia.
- `long serialVersionUID`: sirve para indicar el identificador de la clase al leer y escribir la clase en un archivo `.obj` con `ObjectOutputStream` y `ObjectInputStream` respectivamente.

La clase `Node` es la clase que contiene todos los pesos y biases de la red neuronal. La clase `Node` es una clase abstracta, y tiene 3 clases que heredan de esta:

- `InputNode`: Recibe los datos de entrada. En este caso, los píxeles del número dibujado. El valor que debe recibir es un valor entre 0 y 1
- `ConnectionNode`: Conecta los nodos de la red neuronal
- `OutputNode`: Devuelve los valores calculados en la red neuronal. En este caso, devuelve una interpretación de qué número está dibujado

En la estructura de la red neuronal, la primera capa contiene `InputNode`, la última contiene `OutputNode` y el resto contiene `ConnectionNode`.

Las conexiones se guardan en la lista de `Double weightsFrontLayer`, y la referencia del nodo de la siguiente capa se guarda en `idNodeFrontLayer`, una lista de `Integer`. Estas listas las contienen los `InputNode` y `ConnectionNode`, ya que los `OutputNode` no contienen ninguna capa por delante a la que conectarse.

```
private ArrayList<Integer> idNodeFrontLayer;  
private ArrayList<Double> weightsFrontLayer;
```

En Java 23 de 64 bits, almacenar un `Integer` de 4 bytes para identificar la neurona conectada consume menos memoria que guardar una referencia a un objeto, que normalmente ocupa 8 bytes.

Este es un ejemplo de implementación en la función del `InputNode` de transferir los datos en la alimentación hacia delante:

```
for (int i = 0; i < weightsFrontLayer.size(); i++) {  
    int id = idNodeFrontLayer.get(i);  
    ret_values[id] = value * weightsFrontLayer.get(i);  
}
```

Este proyecto tiene un reenfoque distinto al de otros códigos. En lugar de pasar los datos en las funciones feedforward y backpropagation con una multiplicación de matriz, el código realiza todos estos cálculos manualmente. Los motivos principales son:

1. Mayor facilidad para comprender el funcionamiento de cómo se pasan los datos en cada momento
2. Teóricamente, al eliminar pesos, la red neuronal necesita menos recursos y menos tiempo de cálculo.
3. De esta manera, con redes neuronales de mayor escala, se pueden hacer lo que en este proyecto denominamos como 'árboles neuronales'. La idea de este es que puede que en algunos casos sea más eficaz una agrupación de datos concretos. Por ejemplo, en un videojuego, puede que para calcular la posición de un personaje necesites primero vectorizar de una manera que la inteligencia artificial entienda la posición (1ªcapa), y luego con esa posición vectorizada haga los cálculos que necesite, y que por otra parte también haga cálculos con otros datos que reciba

11.4.2 Inicialización

La inicialización de la red neuronal empieza con la creación de una instancia de la clase `NeuralNetwork`, siendo los parámetros:

- `int[] shape`: Tamaño de la red neuronal, indicando en cada valor el tamaño de las distintas capas
- `double max`: Valor máximo con el que se puede inicializar tanto los pesos como los biases. El rango es de `-max` a `max`

```
ai = new NeuralNetwork(shape, 1);
```

Este constructor primero crea los nodos y luego los conecta.

```
public NeuralNetwork(int[] shape, double max) throws Exception {  
    setNodes(createNodes(shape, max));  
    connectNodes(max);  
}
```

Para crear los nodos, llama a la función createNodes. Esta función devuelve todos los nodos de la red neuronal por capas.

Esta función primero crea las capas, y luego crea los nodos de sus respectivas capas.

La primera capa contiene InputNode, la última contiene OutputNode y el resto contiene ConnectionNode

```
private Node[][] createNodes(int[] shape, double max) throws Exception {
    if (shape.length <= 1) {
        throw new Exception("Tiene que tener más de una capa");
    }

    Node[][] nodes = new Node[shape.length][];

    // Crear la forma de la red neuronal
    nodes[0] = new InputNode[shape[0]];
    for (int i = 1; i < nodes.length - 1; i++) {
        nodes[i] = new ConnectionNode[shape[i]];
    }
    nodes[shape.length - 1] = new OutputNode[shape[shape.length - 1]];

    // Rellenar la primera capa del array de nodos con InputNodes
    for (int i = 0; i < shape[0]; i++) {
        nodes[0][i] = new InputNode();
    }

    // Rellenar el resto de capas excepto la última del array de nodos con
    ConnectionNodes
    for (int i = 1; i < shape.length - 1; i++) {
        for (int j = 0; j < shape[i]; j++) {
            nodes[i][j] = new ConnectionNode(max);
        }
    }

    // Rellenar la última capa del array de nodos con OutputNodes
    for (int i = 0; i < shape[shape.length - 1]; i++) {
        nodes[nodes.length - 1][i] = new OutputNode(max);
    }
    return nodes;
}
```

En los constructores de tanto ConnectionNode como de OutputNode la variable es el valor máximo del bias

```
public ConnectionNode(double max) {  
    // Inicializar los arrayLists de las conexiones de la siguiente capa  
    setIdNodeFrontLayer(new ArrayList<>());  
    setWeightsFrontLayer(new ArrayList<>());  
    // Inicializar el bias en un valor aleatorio del -max al max  
    setBias(Math.random() * (max * 2) - max);  
}
```

En el setter de los nodos se define también automáticamente la variable shapeTotal, indicando el tamaño total de la red neuronal

```
private void setNodes(Node[][] nodes) {  
    this.nodes = nodes;  
    shapeTotal = (short) nodes.length;  
}
```

La función connectNodes conecta todos los nodos de una capa con todos los pesos de la siguiente capa mediante pesos.

```
private void connectNodes(double max) {  
    for (int i = 0; i < nodes.length; i++) {  
        if (nodes[i][0] instanceof InputNode) {  
            for (int j = 0; j < nodes[i].length; j++) {  
                for (int k = 0; k < nodes[i + 1].length; k++) {  
                    ((InputNode) nodes[i][j]).addNodeFront(k, max);  
                }  
            }  
        } else if (nodes[i][0] instanceof ConnectionNode) {  
            for (int j = 0; j < nodes[i].length; j++) {  
                for (int k = 0; k < nodes[i + 1].length; k++) {  
                    ((ConnectionNode) nodes[i][j]).addNodeFront(k, max);  
                }  
            }  
        }  
    }  
}
```

La función addNodeFront crea un peso, y sus parámetros son

- int idNode: Es el index del nodo al que el peso se conecta en la siguiente capa
- double max: Valor máximo con el que se puede inicializar su nodo

```
public void addNodeFront(int idNode, double max) {  
    // Inicializar el valor de una conexión, con un rango del -max al max  
    getIdNodeFrontLayer().add(idNode);  
    getWeightsFrontLayer().add(Math.random() * (max * 2) - max);  
}
```

11.4.3 Alimentación para delante

Después de inicializar la red neuronal, los datos de muestra guardados en distintos archivos son leídos por el programa. Se encuentran separados en 2 archivos distintos para facilitar su uso. Cada archivo contiene registros de 785 valores numéricos cada uno. El primer valor es un número del 0 al 9, que representa qué número es realmente el dato de prueba. Los 784 restantes representan cada uno de los píxeles de los números trazados a mano en la matriz de 28x28. En total, hay 60.000 registros.

Uno de los archivos contiene los datos de aprendizaje, y el otro los datos de testeo. Los datos de aprendizaje son los datos usados para que la red neuronal aprenda, y los datos de testeo los usados para verificar el margen de error.

Los datos de aprendizaje son 50.000 registros, mientras que los datos de testeo son 10.000. Como la red neuronal no aprende en ningún momento de estos, de esta manera se verifica que no está simplemente memorizando los casos de prueba.

La alimentación para delante empieza llamando a la función train, siendo sus parámetros:

- List<double[]> data: El dato del Input
- List<double[]> expectedData: El valor que se espera en el output
- int minibatchSize (opcional): El tamaño del minibatch
- double learningRate: La tasa de cambio en los pesos, para el aprendizaje

```
public void train(List<double[]> trainingDatas, List<double[]> expectedDatas,  
double learningRate)
```

```
public void train(List<double[]> trainingDatas, List<double[]> expectedDatas,  
int miniBatchSize, double learningRate)
```

Esta función sobrecargada, ya que tiene 2 posibilidades. Una de ellas contiene un parámetro miniBatchSize, que sirve para dividir la lista de datos de aprendizaje en fragmentos de tamaño n

```
for (int j = 0; j < trainingDatas.size(); j += miniBatchSize) {  
    double[][] miniBatchData;
```

```

double[][] miniBatchExpected;
if (j + miniBatchSize > trainingDatas.size()) {
    miniBatchData = new double[trainingDatas.size() - j][];
    miniBatchExpected = new double[trainingDatas.size() - j][];
}
else {
    miniBatchData = new double[miniBatchSize][];
    miniBatchExpected = new double[miniBatchSize][];
}
for (int k = 0; k < miniBatchData.length; k++) {
    miniBatchData[k] = trainingDatas.get(index_records.get(j + k));
    miniBatchExpected[k] = expectedDatas.get(index_records.get(j + k));
}

runMiniBatch(miniBatchData, miniBatchExpected, learningRate, false);
}

```

La otra función de train hace los cambios manualmente, y llama a la función run por cada dato de aprendizaje

```

public void train(List<double[]> trainingDatas, List<double[]> expectedDatas,
double learningRate) {
    for (int j = 0; j < trainingDatas.size(); j++) {
        run(trainingDatas.get(j), expectedDatas.get(j), learningRate, false);
    }
}

```

```

public void run(double[] data, double[] expectedData, double learningRate,
boolean showMarginError) throws ExceptionInInitializerError {
    // Ejecutar la red neuronal, y obtener los valores de todos los nodos
    durante la alimentación hacia delante
    double[][] values = feedforward(data);

    if (showMarginError) {
        // Calcular el margen de error de la alimentación para delante
        respecto al dato esperado
        double marginError = 0;
        for (int i = 0; i < nodes[shapeTotal - 1].length; i++) {
            marginError += Math.pow(expectedData[i] - values[values.length -
1][i], 2) / 2;
        }

        // Imprimir los outputs
        // for (int i = 0; i < nodes[shapeTotal - 1].length; i++) {
        //     System.out.println((values[values.length - 1][i]));
        // }

        System.out.println("Error de margen: " + marginError);
    }

    // Obtener los cambios de los pesos y biases y hacer los cambios de los
    datos obtenidos
    changeWeightsAndBiases(backpropagation(values, expectedData),

```

```
learningRate);  
}
```

La función runMiniBatch luego ejecuta todos los datos y hace todos los cambios a la vez. Esto es para optimizar el aprendizaje

```
public void runMiniBatch(double[][] data, double[][] expectedData, double  
learningRate, boolean showMarginError) throws ExceptionInInitializerError {  
    double[][][] changes = null;  
    for (int i = 0; i < data.length; i++) {  
        double[][] values = feedforward(data[i]);  
  
        if (showMarginError) {  
            // Calcular el margen de error de la alimentación para delante  
            // respecto al dato esperado  
            double marginError = 0;  
            for (int j = 0; j < nodes[shapeTotal - 1].length; j++) {  
                marginError += Math.pow(expectedData[i][j] -  
values[values.length - 1][j], 2) / 2;  
            }  
  
            // Imprimir los outputs  
            for (int j = 0; j < nodes[shapeTotal - 1].length; j++) {  
                System.out.println(values[values.length - 1][j]);  
            }  
  
            System.out.println("Error de margen: " + marginError);  
        }  
  
        // Obtener los cambios de los pesos y biases  
        if (changes != null) {  
            changes = NeuralMath.addArrays(changes, backpropagation(values,  
expectedData[i]));  
        }  
        else {  
            changes = backpropagation(values, expectedData[i]);  
        }  
    }  
    // Hacer los cambios de los datos obtenidos  
    changeWeightsAndBiases(changes, learningRate);  
}
```

La función feedforward es la que hace la alimentación hacia delante. El único valor que recibe como parámetro es un array con los valores de Input

```
private double[][] feedforward(double[] data) throws  
ExceptionInInitializerError
```

La variable en la que se guarda el valor de cada nodo durante la alimentación hacia delante es un array bidimensional llamado `double[][]`, llamada `values`. Esta variable el valor devuelto es esta variable misma.

Primero, se copia el array de los datos en la primera capa

```
System.arraycopy(data, 0, values[0], 0, values[0].length);
```

Luego, en cada capa, se van pasando los valores de cada capa a través de los pesos con la función `transferData` de cada nodo.

```
for (int j = 0; j < nodes[0].length; j++) {
    values[1] = NeuralMath.addArrays(
        values[1], ((InputNode)nodes[0][j]).transferAllData(
            values[0][j], nodes[1].length));
}

// Añadir el bias de cada nodo en la segunda capa, y editar el valor con la
función sigmoide
// El valor de la variable capa es 1 porque estamos editando la segunda capa
ahora, al ya haber pasado todos los datos
if (nodes[1][0] instanceof ConnectionNode) {
    setUpConnections(values[1], 1);
}
else {
    setUpOutputs(values[1], 1);
}

for (short i = 1; i < shapeTotal - 1; i++) {
    // Añadir los valores de la siguiente capa
    for (int j = 0; j < nodes[i].length; j++) {
        values[i + 1] = NeuralMath.addArrays(
            values[i + 1], ((ConnectionNode)nodes[i][j]).transferAllData(
                values[i][j], nodes[i + 1].length));
    }

    // Añadir el bias de cada nodo, y editar el valor con la función sigmoide
    if (nodes[i + 1][0] instanceof ConnectionNode) {
        setUpConnections(values[i + 1], i + 1);
    }
    // Añadir el bias de cada nodo en la última capa, y editar el valor con
la función sigmoide
    else {
        setUpOutputs(values[i + 1], i + 1);
    }
}
}
```

El código está duplicado, pero es para que en la primera capa se haga con los `InputNode` y en el resto con las `ConnectionNode`.

La función de `transferAllData` devuelve un array que es el valor que este nodo pasa a cada valor de la siguiente capa.

```

public double[] transferAllData(double value, int sizeNextLayer) {
    double[] ret_values = new double[sizeNextLayer];
    for (int i = 0; i < weightsFrontLayer.size(); i++) {
        int id = idNodeFrontLayer.get(i);
        ret_values[id] = value * weightsFrontLayer.get(i);
    }

    return ret_values;
}

```

La función `setUpConnections` y `setUpOutputs` en la clase `NeuralNetwork` suma el bias a todos los nodos de esta capa, al igual que sacar la función sigmoide de sí misma:

```

for (int j = 0; j < nodes[capa].length; j++) {
    values[j] += ((ConnectionNode) nodes[capa][j]).getBias();
    values[j] = NeuralMath.setSigmoid(values[j]);
}

```

El valor final de cada capa es

$$\sigma \left(\sum_j (w_j x_k) + b \right)$$

donde, $w_{(j)}$ es el peso del nodo j y $x_{(k)}$ es el valor de cada nodo k de la capa anterior.

La función sigmoide (σ) lo que hace es convertir un número cualquiera en un número en el rango entre 0 y 1. He escogido como función sigmoide la siguiente [SIGM]:

$$\frac{1}{1 + e^{-n}}$$

La función sigmoide se encuentra en la clase `NeuralMath`:

```

public static double setSigmoid(double n) {
    return (1 / (1 + Math.exp(- n)));
}

```

Finalmente, el valor de toda la red neuronal es devuelta a la función `run` y es guardada en una variable

11.4.4 Retropropagación

Después, el valor devuelto por la función `feedforward` es usada en la función `backpropagation` como parámetro para después cambiar los valores de los pesos y biases.

```

changeWeightsAndBiases(backpropagation(values, expectedData), learningRate);

```

Primero, se calcula el valor del margen de error de cada nodo en el array `double[][] difNodeData`, empezando por la última capa.

```
difNodeData[shapeTotal - 2] = NeuralMath.subtractArrays(expectedData,
values[values.length - 1]);

for (int j = 0; j < nodes[shapeTotal - 1].length; j++) {
    difNodeData[shapeTotal - 2][j] *=
NeuralMath.setDerivativeSigmoid(values[values.length - 1][j]);
}
```

El tamaño de `difNodeData` es del tamaño del array `nodes` en la red neuronal menos la primera capa, ya que no es necesario para el cálculo.

El cálculo de la última capa es:

$$(\sigma(x^l) - y) \odot \sigma'(x^l)$$

donde l es el número de la última capa

Para calcular el margen de error del resto de nodos, se llama a la función `getExpectedDataNode` de cada nodo, pasando como parámetro el margen de error de la capa de delante.

```
public double getExpectedDataNode(double nodeValue, double[] errorsNextLayer)
{
    double ret_values = 0;
    for (int i = 0; i < weightsFrontLayer.size(); i++) {
        int id = idNodeFrontLayer.get(i);
        ret_values += weightsFrontLayer.get(i) * errorsNextLayer[id];
    }
    ret_values *= NeuralMath.setDerivativeSigmoid(nodeValue);
    return ret_values;
}
```

La operación final es

$$\left(\sum_j w_j \delta_{l+1}^j \right) \sigma'_x$$

donde x es el nodo del que estamos calculando el margen de error, j es el índice del peso w al nodo que está conectado en la capa de delante y $\delta_{(l+1)}$, siendo el margen de error de la capa siguiente a la que estamos calculando

Finalmente, se calcula el margen de error de cada peso y bias y se guarda en la variable `double[][][] ret_values`, la cual se terminará devolviendo. La estructura devuelta por la función es en un array tridimensional. Cada array es una capa de la red neuronal, excepto la última. Dentro de cada capa, contiene un array para cada nodo, y dentro de cada uno de estos contiene un valor para cada peso. En el último array se guarda una estructura similar, pero para los biases.

```

ret_values[0] = new double[nodes[0].length][];
for (int i = 0; i < nodes[0].length; i++) {
    ret_values[0][i] = ((InputNode)
nodes[0][i]).getExpectedDataWeights(difNodeData[0], values[0][i],
nodes[1].length);
}

// Usar Los datos de Los valores de Los nodos en difNodeData de La red neuronal para obtener el cambio de cada peso en el resto de capas, menos la última
for (short i = 1; i < shapeTotal - 1; i++) {
    ret_values[i] = new double[nodes[i].length][];
    for (int j = 0; j < nodes[i].length; j++) {
        ret_values[i][j] = ((ConnectionNode)
nodes[i][j]).getExpectedDataWeights(difNodeData[i], values[i][j], nodes[i + 1].length);
    }
}

// Usar Los datos de Los valores de Los nodos en difNodeData de La red neuronal para obtener el cambio de cada bias de La red neuronal
// Es guardado en la última posición del array devuelto
ret_values[ret_values.length - 1] = difNodeData;

```

El margen de error de cada peso es calculado en la función `getExpectedDataWeights`. Los parámetros son:

- El margen de error de la capa siguiente
- El valor de ese nodo
- Tamaño de la siguiente capa

Devuelve todos los cambios de los pesos de ese nodo

```

protected double[] getExpectedDataWeights(double[] difDatas, double value,
int sizeNextLayer) {
    double[] ret_values = new double[sizeNextLayer];
    for (int i = 0; i < getWeightsFrontLayer().size(); i++) {
        int index = getIdNodeFrontLayer().get(i);
        ret_values[index] = difDatas[index] * value;
    }
    return ret_values;
}

```

La operación de este es:

$$\sigma(x_k^{l-1})\delta_j^l$$

Donde l es la capa en la que estamos, j es el nodo del que está conectado, y k es el nodo al que está conectado

Los márgenes de error de los bias son simplemente los márgenes de error de su respectivo nodo

Finalmente, se llama a la función `changeWeightsAndBiases`, siendo sus parámetros

- `double[][][] changes`: array tridimensional con los cambios de los pesos y biases

- learningRate: Tasa de aprendizaje, un multiplicador para mostrar cuánto se cambia el valor de cada peso y bias

```

public void changeWeightsAndBiases(double[][][] changes, double
learningRate) {
    // Usar los datos esperados de changes para cambiar los pesos de la
    primera capa de la red neuronal
    for (int j = 0; j < nodes[0].length; j++) {
        // Obtener el index del peso. Después, actualiza ese peso
        for (int k = 0; k < ((InputNode)
nodes[0][j]).getWeightsFrontLayer().size(); k++) {
            int index = ((InputNode)
nodes[0][j]).getIdNodeFrontLayer().get(k);
            ((InputNode) nodes[0][j]).addToWeightsFrontLayer(k,
changes[0][j][index] * learningRate);
        }
    }
    for (short i = 1; i < shapeTotal - 1; i++) {
        for (int j = 0; j < nodes[i].length; j++) {
            // Obtener el index del peso. Después, actualiza ese peso
            for (int k = 0; k < ((ConnectionNode)
nodes[i][j]).getWeightsFrontLayer().size(); k++) {
                int index = ((ConnectionNode)
nodes[i][j]).getIdNodeFrontLayer().get(k);
                ((ConnectionNode) nodes[i][j]).addToWeightsFrontLayer(k,
changes[i][j][index] * learningRate);
            }
        }
    }
    // Usar los datos esperados de changes para cambiar todos los biases
    de la red neuronal
    for (short i = 1; i < shapeTotal - 1; i++) {
        for (int j = 0; j < nodes[i].length; j++) {
            ((ConnectionNode) nodes[i][j]).addBias(changes[changes.length
- 1][i - 1][j] * learningRate);
        }
    }
    // Usar los datos esperados de changes para cambiar los biases de la
    última capa de la red neuronal
    for (int j = 0; j < nodes[shapeTotal - 1].length; j++) {
        ((OutputNode) nodes[shapeTotal -
1][j]).addBias(changes[changes.length - 1][shapeTotal - 2][j] *
learningRate);
    }
}

```

11.5 Uso

- Lectura de las bases de datos (tanto del entrenamiento como el de las pruebas).
- Crear la red neuronal, indicando el tamaño de las conexiones.
- [Opcional]: En base a los objetivos deseados (eficiencia, velocidad) realizar las modificaciones oportunas en la red neuronal, (añadir o eliminar pesos, y añadir nodos)

- Entrenar a la red neuronal usando la función 'train', pasando como parámetros
 - o List<double[]> data: Dato del Input. Es necesario que los valores introducidos sean del 0 al 1.
 - o List<double[]> expectedData: Valor que se espera en el output. Es necesario que los valores introducidos sean del 0 al 1.
 - o int minibatchSize (opcional): Tamaño del minibatch.
 - o double learningRate: Tasa de cambio en los pesos, para el aprendizaje.
- Con la función 'run' la red neuronal obtiene un resultado respecto a los valores pasados (píxeles). Este mismo proceso se puede realizar de manera más fácil con el programa **DrawNumber**.

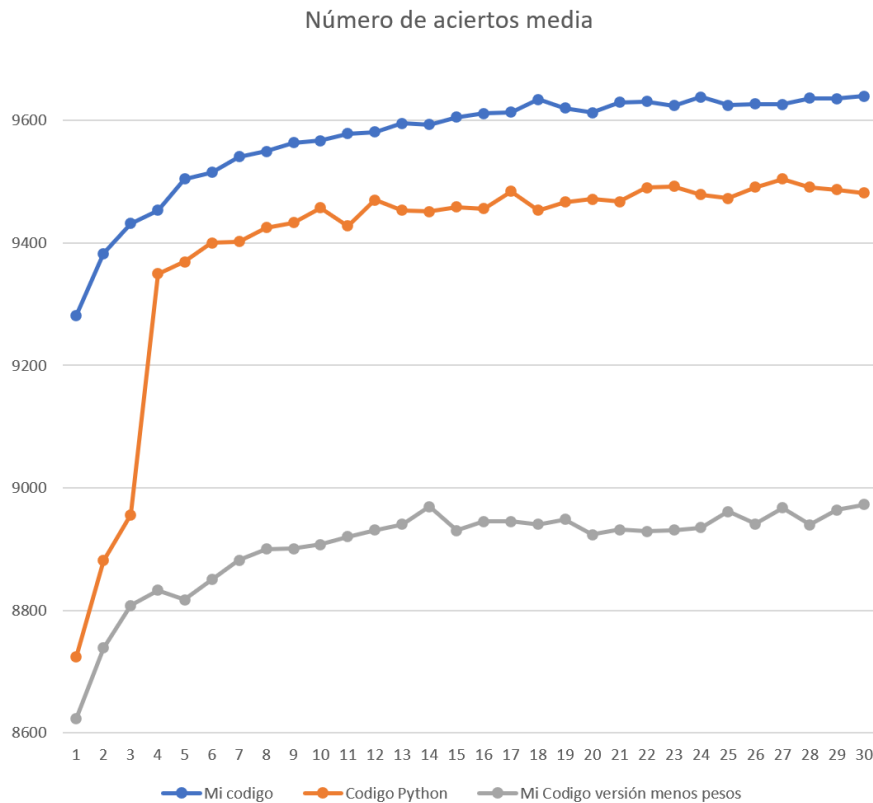
11.6 Pruebas

Se han efectuado 3 pruebas de cada uno de los códigos, usando una red neuronal con tamaño [784, 30, 10], con tasa de aprendizaje 0,3, minibatch de tamaño 10, 30 iteraciones de todos los datos de aprendizaje y con todas las optimizaciones nombradas anteriormente

Luego, para comparar tanto la eficiencia como la velocidad del código, se realiza nuevamente pruebas con otro programa en Python, creado por otro usuario (el del tutorial) con los mismos datos.

Además, se crea una tercera versión de red neuronal eliminando el máximo número de pesos en la primera capa, teniendo ésta únicamente 784 pesos, en lugar de 23520 pesos como en las 2 versiones anteriores.

Aciertos de las pruebas (test)



Tiempo código en Python

103,72	108,31	106,01
--------	--------	--------

Tiempo mi código

227,93	217,24	240,88
--------	--------	--------

Tiempo de la versión de mi código con menos pesos

117,85	129,02	116,12
--------	--------	--------

Como se puede observar en los resultados, el código de Java tiene una media de aciertos superior al del tutorial, lo cual es un dato muy positivo.

Sin embargo, el código de Java es 2 veces más lento que el de Python, incluso superando en velocidad a la versión del mínimo de pesos.

El motivo de esto es que la librería Numpy es una librería creada en C++, que usa operaciones nativas, es decir, hace operaciones directamente en el procesador, controlando la memoria de forma muy óptima. El código de Java sería mucho más rápido si el código en Python no usase la librería Numpy, o se utilizaran otras librerías de Java, como puede ser ND4J, que también usa C++.

Conclusiones

[Deben guardar relación con los objetivos del proyecto.]

12 Líneas futuras

[Si procede. Derivados de las conclusiones.]

He avanzado bastante, y he descubierto muchas cosas, pero todavía me quedan muchas preguntas. ¿Cómo encuentran esos patrones (pregunta inicial, y una de las mayores de los expertos)? ¿Si no tienes los datos reales, cómo aprenden igualmente (aprendizaje no supervisado, aprendizaje por refuerzo...)? ¿Se puede optimizar de alguna manera mi código, y cómo?

Las próximas cosas que haría para este proyecto son:

- Optimizar mi código, usando librerías como ND4J para optimización de bucles
- Mejorar la función de aprendizaje
- Hacer el programa más eficaz, principalmente usando la potencia de la gráfica
- Recrear el proyecto en C++, para aumentar la optimización
- Crear mi propia IA, que se usaría para mejorar la jugabilidad en los juegos automáticamente

Referencias

- 3Blue1Brown explicación: https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
- Explicación a mayor profundidad: <http://neuralnetworksanddeeplearning.com/chap1.html>
- Retropropagación: <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>
- Red neuronal (otra en Python, me he ayudado en este código): <https://github.com/mattm/simple-neural-network/blob/master/neural-network.py>
- Aprendizaje no supervisado: <https://stackoverflow.com/questions/15514618/machine-learning-unsupervised-backpropagation>

- Diferencia entre los 3 tipos de aprendizaje:
<https://www.educative.io/answers/supervised-vs-unsupervised-vs-reinforcement-learning>
- [SIGM] Función sigmoide:
https://es.wikipedia.org/wiki/Funci%C3%B3n_sigmoide
-

Anexos

- Hacer la pizarra para dibujar los números:
<https://stackoverflow.com/questions/40746687/drawing-a-game-board-in-java>
- Video que me inició en el mundo de la IA, la primera vez que oí de IA a profundidad:
https://www.youtube.com/watch?v=Dw3BZ6O_8LY&t=3s
-

[Si proceden.

La documentación que no forme parte del contenido principal se puede adjuntar como documentación adicional. Si un anexo pertenece al autor del proyecto se le suele denominar “apéndice”.

Ejemplos de anexos pueden ser: código fuente escrito en un lenguaje de programación determinado, capturas de pantalla u otros

Deberán aparecer, cada anexo, en páginas independientes.]

TODO:

Poner los identificadores en consolas 10

Pedirle a la IA sugerencias para todo el documento

- Estructura
- Expresión
- Contenido

Mayor presencia código

Formatear fórmulas

Organizar arquitectura, diseño e implementación (cambiar marco teórico arquitectura)