Instituto San Mateo

Bachillerato de Excelencia

PROYECTO DE INVESTIGACIÓN

Diego Berrocal Gutiérrez

INTELIGENCIA ARTIFICIAL
EDITOR DE REDES NEURONALES

Trabajo dirigido por

María Gaspar

curso 2018-2019

Inteligencia Artificial
Editor de Redes Neuronales

Proyecto de investigación del BE – curso 2018-2019

Diego Berrocal Gutiérrez

Resumen

Actualmente la inteligencia artificial se está utilizando en importantes empresas como Amazon o Google y próximamente su uso se extenderá a nuestro ámbito personal. Científicos de todo el mundo tratan de entender el funcionamiento de nuestros prodigiosos cerebros. La inteligencia artificial no es fácil de comprender, y enseñar a personas a utilizarla no es una tarea simple. Por ello, en este proyecto me he propuesto desarrollar un simulador y a la vez editor de inteligencia artificial con una interfaz gráfica que facilite desarrollar, entender e innovar en este sector tecnológico.

Abstract

Nowadays, artificial intelligence is taking an important role in many companies and in our personal lives. People around the world are trying to understand how our brains work. Artificial intelligence is not easy to learn. For this reason, I have created an editor of artificial intelligence with a graphical interface in order to help people to learn this new technology.

Palabras clave: Inteligencia artificial, futuro, algoritmos, red neuronal, aprendizaje autónomo, programación, AI, machine learning, aprendizaje profundo, educación

Keywords: Artificial Intelligence, Deep Learning, learning, education, machine learning, editor, programming, self-learning, future, AI, feed-forward, back-propagation

Índice

1. Introducción

En este proyecto se pretende diseñar un editor de inteligencia artificial con el que poder crear, editar, ajustar y visualizar distintas redes neuronales. Se crearán los distintos algoritmos, interfaces gráficas y mecanismos internos utilizando Java[1] como lenguaje de programación principal.

El programa se podrá descargar en la dirección web siguiente: `https://www.retopall.com`. Además, se podrá ver un trailer de la aplicación en el enlace: `https://www.youtube.com/watch?v=W1fH_xmx6Xg`

Se explicarán los conceptos principales utilizados de inteligencia artificial en el programa y el funcionamiento interno de este. Se recomendarán unos conocimientos básicos matemáticos y de programación.

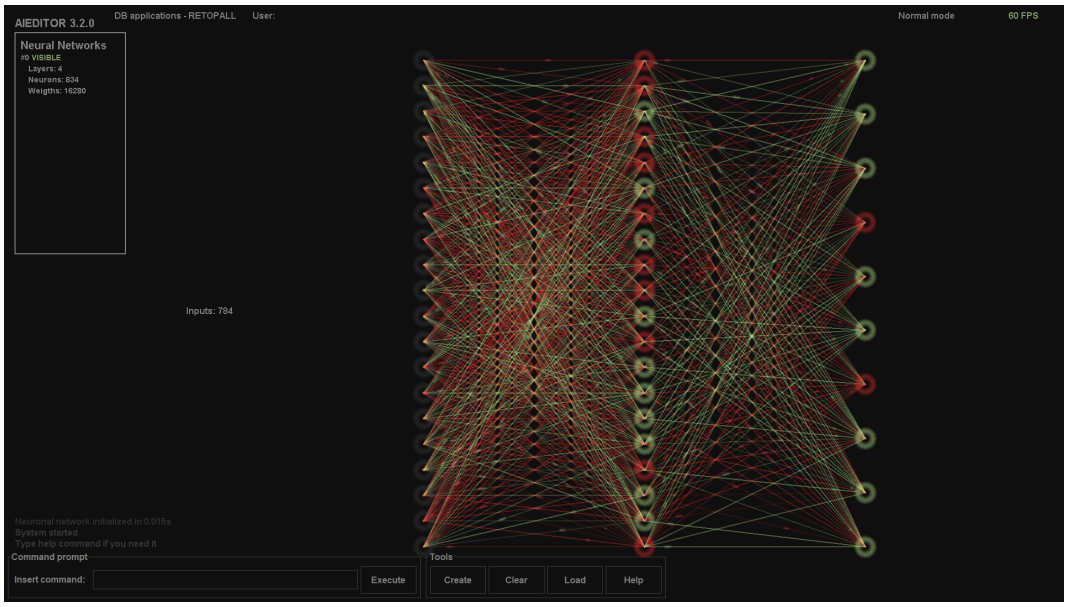


Figura 1: Red neuronal 784x20x20x10

2. Metodología

Las herramientas necesarias para la elaboración del programa son las siguientes:

2.1. Herramientas informáticas

He trabajado con un ordenador con las siguientes especificaciones para la elaboración del programa:

[1]Lenguaje de programación ampliamente usado por empresas y que actualmente pertenece a la compañía Oracle

- Sistema operativo Windows 10
- Procesador Intel Core i7 8700 - Ejecución de los algoritmos
- Tarjeta gráfica Nvidia GeForce 1070 - Entorno gráfico del procesamiento de la red
- Memoria RAM 16GB DDR4 - Almacenamiento de las variables dinámicas del programa
- Disco SSD - Agilizar el almacenamiento de las redes neuronales en el ordenador y la lectura de estas.

El programa no utiliza ninguna librería externa ya que todos los algoritmos están escritos manualmente.

2.2. Herramientas matemáticas

Las redes neuronales utilizan algoritmos de gran complejidad, por lo que usaré herramientas de análisis matemático (derivadas, derivadas parciales, descenso de gradiente, regresión) y del álgebra lineal (matrices, vectores, análisis de funciones). Las herramientas mencionadas anteriormente estarán implementadas en el programa con sus respectivos métodos y funcionalidades.

2.3. Herramientas de programación

Se utilizará un IDE (*Integrated Development Environment*) para programar los algoritmos y la interfaz gráfica de la aplicación. Usaré *Eclipse Photon*.[2] Como se ha dicho anteriormente, el lenguaje utilizado será Java para los algoritmos y es ampliamente usado en grandes empresas como Google, Spotify, Netflix o Amazon.

Java, es el lenguaje más usado por programadores, está siendo utilizado para inteligencia artificial y puede integrar *Tensorflow*[3], una librería desarrollada por Google y más usada para crear algoritmos y redes inteligentes.

3. El editor de redes neuronales

La aplicación objeto de este proyecto puede ser descargada para Windows, MacOSX o Linux. El programa no utiliza librerías externas, es decir, todo está programado manualmente sin ninguna facilidad añadida. La aplicación tiene un total de 16280 lineas de código y 81 clases, lo que lo convierte en un proyecto de amplias dimensiones.

[2] Aplicación desarrollada por Eclipse Foundation. Descargable en: `https://www.eclipse.org/`

[3] Librería de código abierto para aprendizaje automático y utilizada para entrenar redes neuronales, detectar patrones y otras funciones

La estructura de la aplicación es la siguiente:

1. Conexión al servidor web por medio de peticiones HTTP
2. Funcionamiento de una red neuronal
3. Algoritmos de inteligencia artificial
4. Interfaz gráfica para ajustar y manipular la red neuronal
5. Proceso de renderizado gráfico[4] y multitarea[5]
6. Almacenamiento y lectura de redes neuronales

3.1. Conexión al Servidor Web

La aplicación está sincronizada con la página web mencionada anteriormente (`www.retopall.com`)

Para iniciar sesión es necesario disponer de una licencia de usuario. Para verificarla se requiere ejecutar una petición al servidor web PHP[6] para que este haga otra petición a la base de datos MySQL[7].

Las contraseñas se almacenan de forma segura en la base de datos gracias al método de encriptación utilizado MD5.[8] Se digieren los bytes de la contraseña que queremos encriptar y ese conjunto de bytes se convierte en una cadena de números y letras, como se indica a continuación:

```
try {
digester = MessageDigest.getInstance("MD5");
} catch (NoSuchAlgorithmException e) {
e.printStackTrace();
}
if (str == null || str.length() == 0) {
throw new IllegalArgumentException("String to encript cannot be null or
    zero length");
}

digester.update(str.getBytes());
byte[] hash = digester.digest();
StringBuffer hexString = new StringBuffer();
```

[4]Renderizar (en inglés, render) es una función utilizada ampliamente por los programadores para actualizar los gráficos en pantalla

[5]Los programas que desean llevar a cabo varias ejecuciones al mismo tiempo, deben tener una serie de "hilos"que se ejecuten de forma secuencial

[6]Lenguaje de programación web ejecutado en el servidor generalmente para ejecutar peticiones a una base de datos de manera segura

[7]Sistema de gestión de bases de datos

[8]Algoritmo de reducción criptográfico de 128 bits

```
for (int i = 0; i < hash.length; i++) {
if ((0xff & hash[i]) < 0x10) {
hexString.append("0" + Integer.toHexString((0xFF & hash[i])));
} else {
hexString.append(Integer.toHexString(0xFF & hash[i]));
}
}
return hexString.toString();
```

Para enviar la petición al servidor web se crea una conexión URL y se lleva a cabo la petición con el método GET[9].

El código PHP elaborará una respuesta de tipo booleano[10] dependiendo de si los datos son correctos. Si la respuesta es verdadera se procederá a abrir el editor de redes neuronales. La petición al servidor se muestra en el código siguiente:

```
url = new URL(request);
HttpURLConnection connection = (HttpURLConnection) url.openConnection()
    ;
connection.setDoOutput(true);
connection.setInstanceFollowRedirects(false);
connection.setRequestMethod("GET");
connection.setRequestProperty("Content-Type", "text/plain");
connection.setRequestProperty("charset", "utf-8");
connection.connect();
BufferedReader in = new BufferedReader(new InputStreamReader(connection
    .getInputStream(), "UTF-8"));
String get="",line="";
int c=0;
int bool=0;
while((line=in.readLine())!=null){
get+=line;
}
System.out.println(get.trim());
bool=Integer.parseInt(get.trim());
if(bool==1) {
return "1";
}else {
return "User and password dont match";
}
```

La interfaz gráfica para iniciar sesión en la aplicación se muestra en la figura 2

3.2. Red neuronal

Una red neuronal es un modelo computacional utilizado para el aprendizaje por medio, únicamente, de relaciones numéricas.

[9]Petición a un servidor web en el que los parámetros que se dan a este son visibles
[10]Valor falso o verdadero (0 o 1)

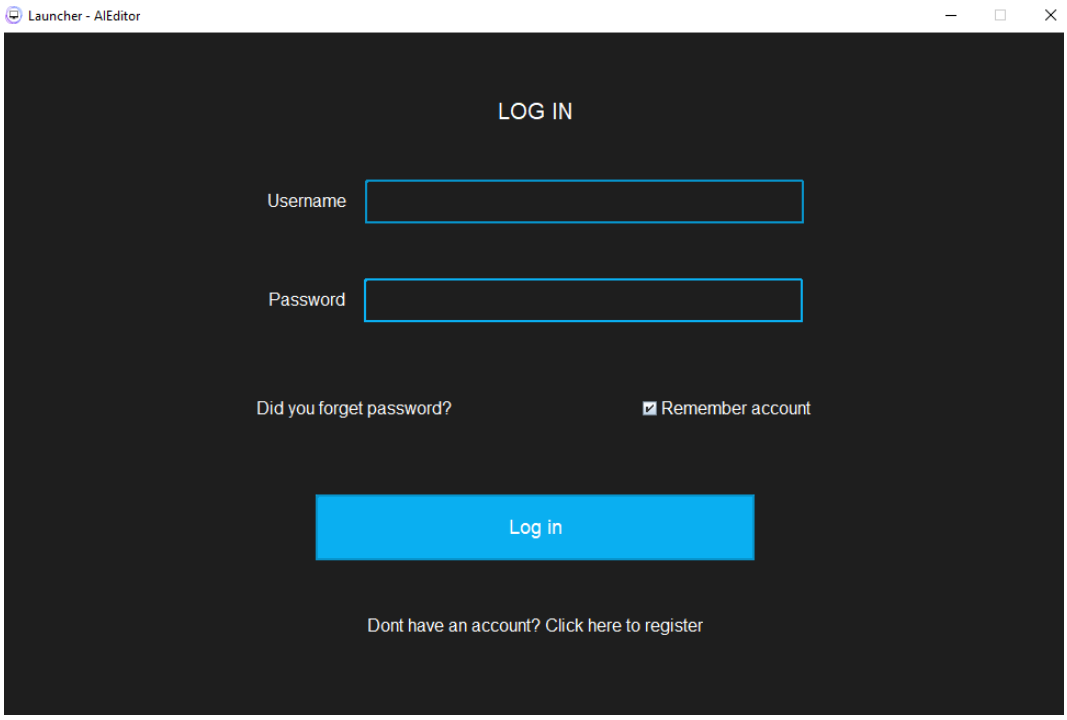


Figura 2: Ventana de inicio de sesión

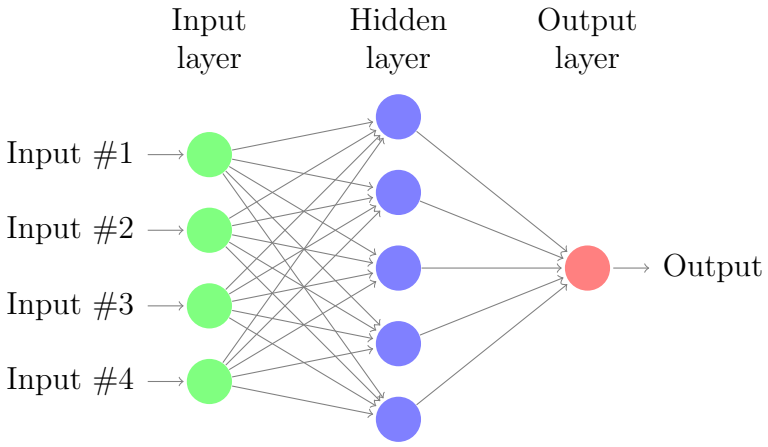
Estas redes están formadas por capas, neuronas y pesos principalmente. Los pesos son las conexiones entre neuronas de una capa anterior con neuronas de una posterior. Los pesos y neuronas almacenan un valor decimal.
En cuanto a las capas, existe una capa de entrada, otra de salida y una serie de capas ocultas.

En la capa de entrada se añaden los valores de los que se quiere partir. Por ejemplo, los píxeles de una imagen a analizar en la red serían los valores de entrada. Dependiendo del color del píxel, a las neuronas de entrada se las asignarían valores de entrada en el intervalo [0,1]. Las neuronas de la capa de salida son los valores de las soluciones y predicciones de la red neuronal.

Finalmente, tendríamos las capas ocultas que mejoran la inteligencia de la red neuronal. Para el entrenamiento de la red, se editarán los valores de los pesos para minimizar el error de las neuronas de salida. Los valores de las neuronas dependerán de los pesos, excepto los de entrada que dependerán de los estímulos dados.

La estructura de la red neuronal se puede apreciar en el siguiente gráfico. Con una red neuronal que posee 3 capas con una capa de entrada de cuatro neuronas, una capa oculta de 5 neuronas y una capa de salida de una única neurona. Las neuronas

de capas próximas están relacionadas con una serie de pesos.



Para entrenar la red neuronal necesitaremos un conjunto de datos con los que aprender. En el ejemplo propuesto de los pixeles de entrada, necesitaremos un conjunto de imágenes en las que cogeremos los pixeles para representarlos como valor de entrada.

A continuación tendremos que calcular el valor de las siguientes neuronas en base a los valores de entrada. Esto se conseguirá con el algoritmo *feed-forward* o propagación hacia delante. Este método nos proporcionará unos valores de salida acotados entre 0 y 1, que corresponderán a la certeza que la red neuronal tendrá en cada neurona de salida.

Debemos tener en cuenta que los valores de entrada siempre serán los estímulos o información que queremos que la red neuronal analice para predecir y producir una serie de resultados en la capa de salida.

Cuando la red neuronal no esté entrenada, los valores de salida serán prácticamente aleatorios. Sin embargo, al llevar a cabo el método *back-propagation* o propagación hacia atrás, la red neuronal calculará el error que ha tenido en los valores de salida en comparación con los deseados y ejecutará el descenso de gradiente[11] para disminuir el error y por tanto aprender.

3.3. Algoritmos de Inteligencia Artificial

La red neuronal tendrá dos funciones principales, que serán el backpropagation y la función feed-forward.

El primero será utilizado para minimizar el error de la red neuronal y el segundo para actualizar la red neuronal. Estas dos funciones estarán sincronizadas por la

[11]Es uno de los algoritmos de optimización más populares en aprendizaje automático, particularmente por su uso extensivo en el campo de las redes neuronales. Gradient descent es un método general de minimización para cualquier función f. A la versión original se le considera lenta pero versátil, sobre todo para casos funciones multi-dimensionales.

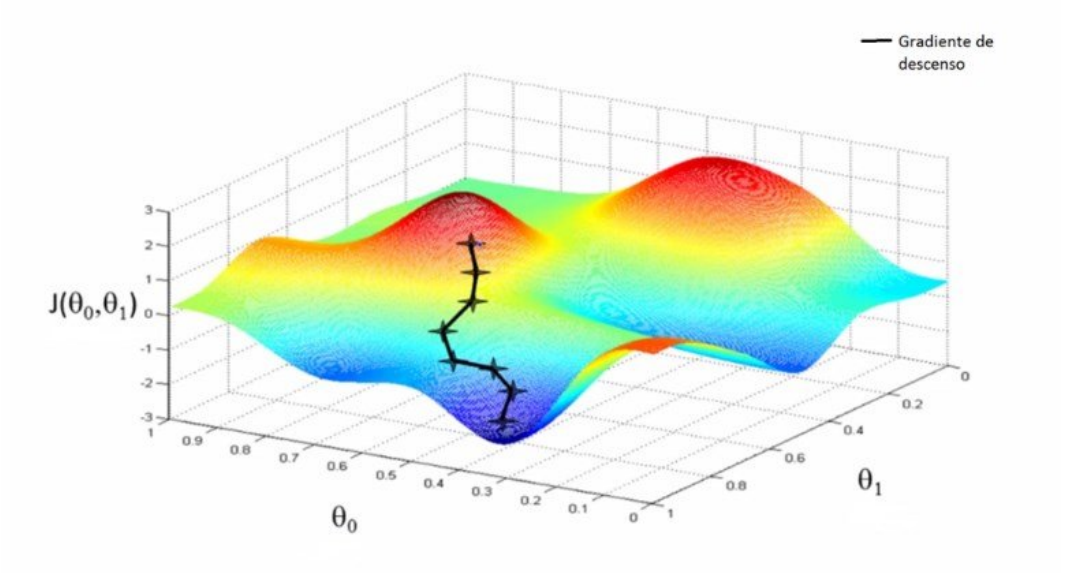


Figura 3: Minimización del error con el algoritmo de descenso de gradiente [10]

función de aprendizaje que iterará sobre todos los conjuntos de datos. Actualizará la red neuronal, calculará el error y finalmente ejecutará la función back-propagation.

3.3.1. Initialize

Al crear la red neuronal se inicializarán todos los pesos y neuronas. Los valores de los pesos se crearán pseudoaleatoriamente con una distribución normal generalmente en el intervalo [-1,1].

$$rand.nextGaussian() * 2 - 1;$$

3.3.2. Feed Forward

La función feed-forward es aquella dedicada a actualizar la red neuronal. Con los valores de entrada llegará a los valores de salida. Dependerá de los valores asignados a todos los pesos. La propagación es hacia delante ya que se ejecuta desde las primeras capas hacia las últimas. A continuación se muestran los términos utilizar para ejecutar el feed-forward. La cantidad de capas que tiene la red neuronal será la capa de entrada, de salida y las ocultas.

a_n^l, a_m^l: son los valores de una neurona en la que l es la capa y m,n son las posiciones en esa capa.
c: son las conexiones con la capa anterior a esta neurona.
$w_{n,k}^l$: peso que conecta la neurona n de la capa l con la neurona m de la siguiente.
$\sigma(x)$: función de activación que se explicará a después
b_n^l: parámetro adicional también a ajustar, que aporta mayor inteligencia a la red neuronal. En términos de programación se denomina *bias*.

Cada neurona se actualizará de la siguiente manera:

$$a_m^{l+1} = \sigma(\sum_{n=0}^{c} w_{m,n}^l a_n^l + b_n^l)$$

La actualización de la red neuronal completa se conseguirá de la siguiente manera. Se recorrerán todas las capas desde la de entrada hasta la de salida donde l será $0, 1, 2, ...$

$$\begin{bmatrix} a_0^{l+1} \\ a_1^{l+1} \\ ... \\ a_m^{l+1} \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{0,0}^l & w_{0,1}^l & w_{0,2}^l & ... & w_{0,n}^l \\ w_{1,0}^l & w_{1,1}^l & w_{1,2}^l & ... & w_{1,n}^l \\ ... & ... & ... & \ddots & ... \\ w_{m,0}^l & w_{m,1}^l & w_{m,2}^l & ... & w_{m,n}^l \end{bmatrix} \begin{bmatrix} a_0^l \\ a_1^l \\ ... \\ a_n^l \end{bmatrix} + \begin{bmatrix} b_0^l \\ b_1^l \\ ... \\ b_n^l \end{bmatrix} \right)$$

La actualización de cada neurona se lleva a cabo en el siguiente fragmento de código:

```
1
for (int i = 0; i < nn.getLayers().size() - 1; i++) {
3 ArrayList<Float[]> newValues = multiplicateByVector(nn.getLayers().get(
    i).getNeuronas(), weigthsArray.get(i),nn.getLayers().get(i));
ArrayList<GraphicalNeuron> neurons = nn.getLayers().get(i + 1).
    getNeuronas();
5 for (int j = 0; j < neurons.size(); j++) {
if(Constants.DEBUG) {
7 addNewInfoToReport("New value in " + j + " Value: " + newValues.get(j)
    [1],nn);
System.out.println("New value in " + j + " Value: " + newValues.get(j)
    [1]);
9 }
neurons.get(j).setOutValue(newValues.get(j)[1]+neurons.get(j).getBias()
    );
11 neurons.get(j).setNetValue(newValues.get(j)[0]+neurons.get(j).getBias()
    );
}
13 }
private static ArrayList<Float[]> multiplicateByVector(ArrayList<
    GraphicalNeuron> neuronsLayer, Matrix2D weigths,
15 Layer l) {
ArrayList<Float[]> vector = new ArrayList<>();
17 float tempSum = 0;

19

21 for (int i = 0; i < weigths.getRows(); i++) {

23 for (int j = 0; j < weigths.getColumns(); j++) {
tempSum = tempSum + neuronsLayer.get(j).getOutValue() * weigths.
    getValue(i, j);
25 }
float value = (float)l.getActivationInstance().activate(tempSum);
27 Float[] values= {tempSum, value};
```

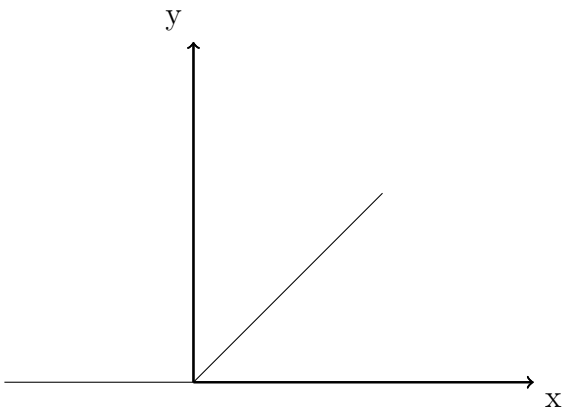
```
    vector.add(values);
29  tempSum = 0;

31  }
    return vector;
33
    }
```

Las neuronas se pueden delimitar mediante las funciones de activación. Estas funciones, en este caso denotadas con el simbolo griego sigma (σ), pueden variar dependiendo de la finalidad y el uso que se le quiera dar a la red neuronal. Existen muchas funciones de activación, pero las principales que usaré en mi programa serán las funciones Sigmoid y RELU.

- Función RELU
  Esta función es muy útil ya que, si la activación es negativa para una neurona, nunca se llegará a activar esa neurona, es decir, no tendrá un valor positivo. Esta función se muestra a continuación:

$$\sigma(x) = \begin{cases} 0 & si \quad x < 0 \\ \\ x & si \quad x > 0 \end{cases}$$

  Representación gráfica de RELU:

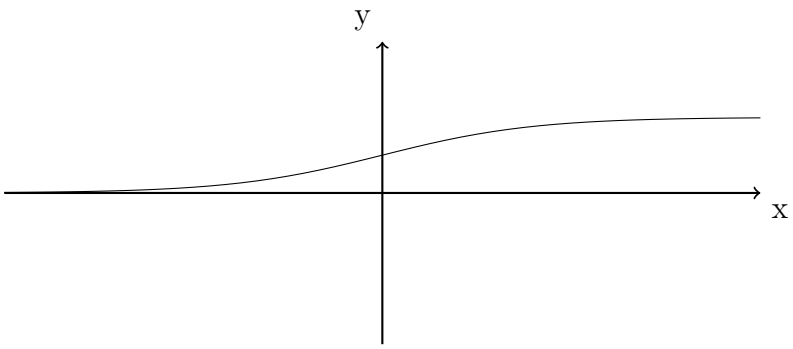


- Función sigmoide
  La función sigmoide es muy sensible a los valores muy cercanos a $x = 0$. Conforme nos alejamos, los cambios llegan a ser indiferentes pues tiene asíntotas horizontales en $y = 0$ e $y = 1$.

$$\lim_{x \to \infty} \sigma(x) = 1 \quad \lim_{x \to -\infty} \sigma(x) = 0$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Representación gráfica de la función sigmoide:



Tambien existen otras activaciones como la función hipérbolica (tanh) $\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

3.3.3. Back Propagation

El principal objetivo de la propagación hacia atrás, es la modificación de todos y cada uno de los pesos de la red con el fin de que la red neuronal aprenda.
Se llevará a cabo un algoritmo en el que, al conocer los valores de salida y los deseados, se calculará el error.

Al recorrer el conjunto de datos de los que aprender, cada dato tendrá su valor deseado de la capa de salida. Después, con el método de descenso de gradiente, se decrecerá el error de la red neuronal. Se recorrerán todos y cada uno de los pesos, se calculará su variación dependiendo del error total y se modificarán estos valores con el fin de disminuir la función de coste $E(x)$.

A continuación se mostrarán los pasos a seguir para la disminución de este coste:

A) Cálculo del error de la capa de salida

Para llevar a cabo el cálculo del error de la red neuronal se determinará el error de la capa salida, es decir, se recorrerán las neuronas de la última capa y se calculará el error sumando todos los cuadrados de las diferencias entre el valor deseado y el real obtenido en la neurona de salida.

x_n: el valor deseado que debería tener la neurona en la posición n de la capa de salida
a_n: el valor real obtenido en la neurona en la posición n de la capa de salida k: número de conjuntos de datos para entrenar.

$$E_{total} = \frac{1}{k} \sum_{a=0} (x_n - a_n)^2$$

B) Minimizando el error de la capa de salida

Para minimizar el error deberemos calcular cuanto decrece este con respecto a cada peso de la red neuronal. Esto lo llevaremos a cabo calculando las derivadas parciales del error con respecto a cada peso.
La notación *out* representa el valor de la neurona con la función de activación aplicada. La notación *net* representa el valor de la neurona sin aplicar la función de activación.

$$\frac{\partial E_{total}}{\partial w_{m,n}} = \frac{\partial E_{total}}{\partial out(a_n^l)} \frac{\partial out(a_n^l)}{\partial net(a_n^l)} \frac{\partial net(a_n^l)}{\partial w_{m,n}}$$

Resolviendo las derivadas se obtiene:

$$\frac{\partial E_{total}}{\partial w_{m,n}} = -(x_n^l - a_n^l)\sigma'(a_n^l)a_m^{l-1}$$

Esta fórmula podrá ser utilizada únicamente en los pesos que conectan la capa de salida con la última oculta, ya que los demás pesos dependen de la variación de los anteriores actualizados.
La derivada de la función de activación σ' en el caso de la función sigmoide será:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

En la fórmula del cálculo de la derivada parcial del error se observa que depende de la neurona anterior. Podremos definir una nueva variable δ que será la sensibilidad de una única neurona repecto al error total.
Utilizaremos δ para calcular la sensibilidad del error con respecto a un peso que solo dependerá de una neurona a_n^l

$$\delta_n^l = -(x_n^l - a_n^l)\sigma'(a_n^l)$$

Además, si queremos modificar los *bias* de la red, se deberá minimizar el error con respecto a cada $\frac{\partial E_{total}}{\partial b_m^l}$

C) Minimizando el error de las capas ocultas

Conociendo la variación necesaria para minimizar el error de los pesos de la capa de salida, se podrán hallar las variaciones necesarias para el resto de pesos. Sin embargo estos, al depender de los anteriores pesos, tendrán un cálculo diferente. Esta es la razón por la que la propagación hacia atrás es desde la capa de salida a la de entrada y se ejecuta en dirección opuesta al algoritmo feed-forward.

k: recorre todos de los pesos de la capa anterior a la del peso a ajustar.
l: la capa de neuronas que está conectada al peso a ajustar y los pesos de la capa siguiente (ya actualizada).

Procederemos entonces al cálculo del resto de pesos que no están conectados a la capa de salida.

$$\frac{\partial E_{total}}{\partial w_{m,n}} = \left(\sum_{k=0} (\delta_k^l w_{m,k}^{l+1}) \right) \sigma' a_n^{l-1}$$

D) Modificando los pesos y bias de la red

Al tener la variación óptima de un peso, para minimizar el error actualizaremos el valor del peso dependiendo de su variación ($\frac{\partial E_{total}}{\partial w_{m,n}}$).

Llamamos η al coeficiente de aprendizaje. Se utiliza para no bajar el error drásticamente ya que hay otro tipo de datos y la red necesita aportar cierta tolerancia a todos ellos. El error obtenido solo depende de un único dato.
Existen varias formas de actualizar los pesos:

- Descenso de gradiente estocástico: actualiza cada peso por cada nuevo dato

$$w_{m,n}^l = w_{m,n}^l - \eta \frac{\partial E_{total}}{\partial w_{m,n}}$$

- Descenso de gradiente por grupos (Batch descent): En cada vuelta completa de todo el conjunto de datos se ajustan a los pesos. t: tamaño conjunto de datos

$$w_{m,n}^l = w_{m,n}^l - \eta \frac{1}{t} \sum_{t=0}^{t} \frac{\partial E_{total}}{\partial w_{m,n}}$$

- Descenso de gradiente por mini-grupos (Mini-Batch descent):
  Es el algoritmo intermedio entre los anteriores, en el que se actualizan los pesos tras un determinado número de iteraciones sobre el conjunto de datos. Es el algoritmo óptimo entre los descritos, ya que no consume mucha memoria al procesador pero a la vez es rápido y por tanto eficiente.
  s tamaño del minigrupo o *mini-batch* (20-100).

$$w_{m,n}^l = w_{m,n}^l - \eta \frac{1}{s} \sum_{s=0}^{s} \frac{\partial E_{total}}{\partial w_{m,n}}$$

- Modificación de los bias
  β: coeficiente de aprendizaje del bias.

Igual que con los pesos también se modificarán los bias de la siguiente manera:

$$b_m^l = b_m^l - \beta \frac{\partial E_{total}}{\partial b_m^l}$$

Estos algoritmos se ejecutarán sobre cada peso y así todos ellos estarán ajustados y el error total de la red disminuirá. Se puede observar este decrecimiento en la imagen 4

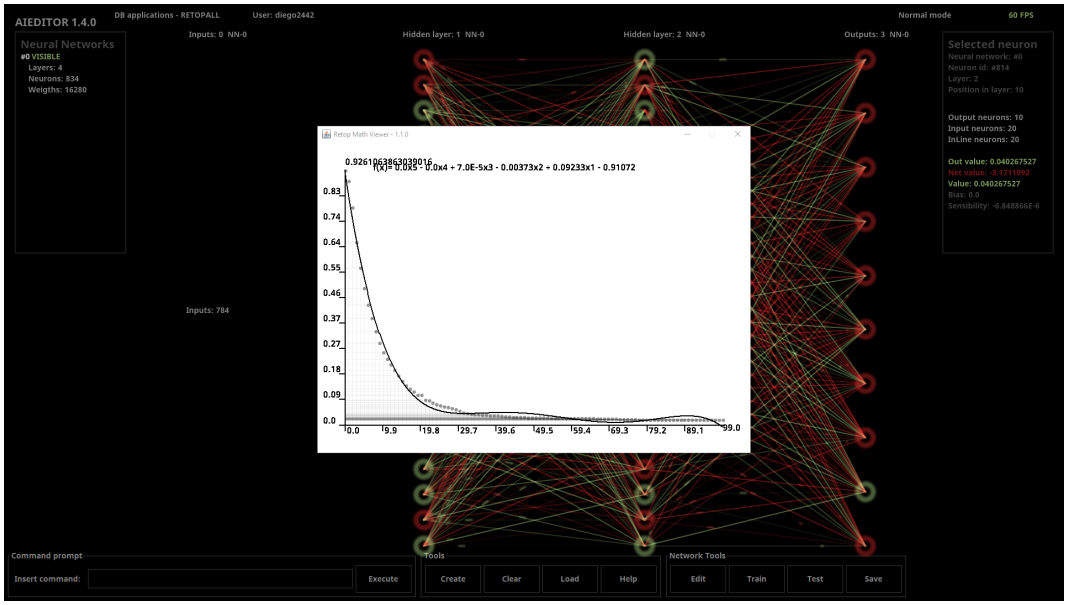


Figura 4: Gráfica del decrecimiento de error de la red neuronal

3.3.4. Entrenamiento

Para llevar a cabo un entrenamiento completo se necesitarán las siguientes variables:

- Conjunto de datos: será un array con las imágenes y, por tanto, los pixeles de todas las imágenes.
- Iteraciones: i el número de conjunto de datos con los que aprender.
- Épocas: *epochs* cantidad de veces que se iterará el conjunto de datos para el aprendizaje.

Se recorrerán todas las épocas. En cada una de estas, se pasará por todo el conjunto de datos si el descenso de gradiente es estocástico. El entrenamiento de la red se muestra a continuación:

```
//Stochastic Gradient Descent
for(int i=0;i<epochs;i++){
    for(ImageTrain trainSet:images){
        float[] outputs=ExecuteFeedForward(trainSet.inputs);
        float error=CalculateError(outputs);
        ArrayList<Matrix2D>weightsUpdates=BackPropagation(error);
        updateWeights(weightsUpdates);
    }
}
```

Se analizarán los colores de los píxeles de las imágenes y se asignarán en la primera capa como valores de entrada. A continuación, ejecutará la función *feed-forward* y con los valores de salida obtenidos se calculará el error. Después se llevará a cabo el *back-propagation* para disminuir el error o función de coste $C(x)$ y actualizar los pesos de la red neuronal.

3.3.5. Algoritmos de optimización

El descenso de gradiente buscará disminuir el error, pero no conseguirá llegar al mínimo absoluto en el que se ha maximizado el aprendizaje de la red neuronal con los conjuntos de datos. Además, no es la forma más rápida de llegar a un mínimo ya que el coeficiente de aprendizaje es constante por lo que siempre habrá un margen de error en el decrecimiento del coste. Por ello, se utilizarán los algoritmos de optimización [Figura 5].

Existen muchos algoritmos de optimización:

- **Momentum**

Se puede conseguir reducir las grandes oscilaciones que presenta el descenso de gradiente estocástico en la búsqueda de un mínimo local. Acelera el descenso en la dirección correcta y reduce la aceleración en el resto de direcciones [Figura 6].
$v(t)$: velocidad en un tiempo concreto t
γ: término del momento. Dependiendo de si se quiere avanzar más pero con menos oscilaciones o viceversa. Suele ser 0.9.

$$v(t) = \gamma v(t-1) - \eta \frac{\partial E_{total}}{\partial w_{m,n}}$$

Cuando calculamos $v(t)$ procederemos, como se ha explicado anteriormente, a actualizar los pesos pero con $v(t)$. Podremos utilizar el descenso por mini-grupos si procedemos a ello.

$$w^l_{m,n} = w^l_{m,n} - v(t)$$

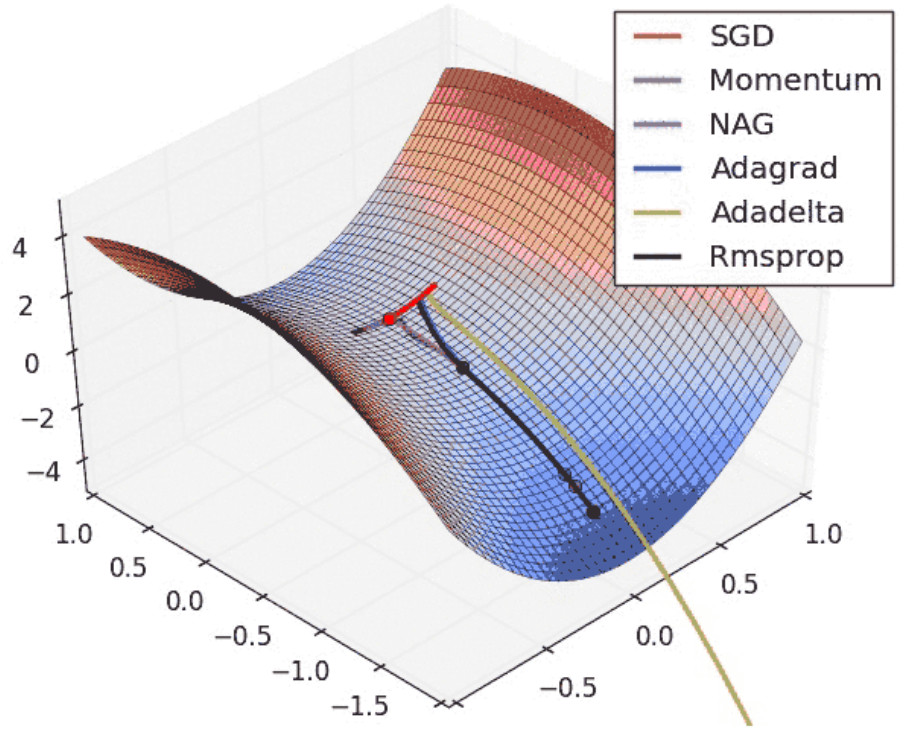
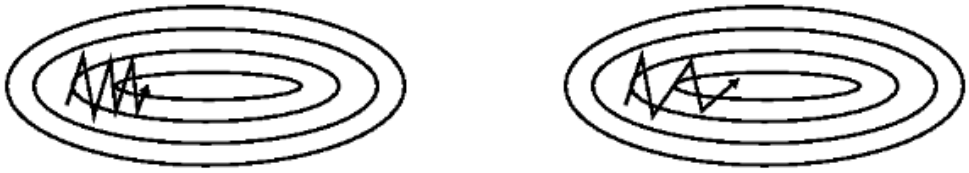


Figura 5: Algoritmos de optimización [11]



(a) SGD without momentum

(b) SGD with momentum

Figura 6: Momentum en contraposición a SGD [12]

- **RMSProp**

También trata de decrecer las oscilaciones de las direcciones que no apuntan al mínimo. β: el valor del momento. Suele ser 0.9

$$v_{dw} = \beta v_{dw} + (1 - \beta) dw^2$$

Al actualizar los pesos se debe hacer lo siguiente para variar el coeficiente de aprendizaje:

$$w^l_{m,n} = w^l_{m,n} - \eta \frac{dw}{\sqrt{v_{db}\epsilon}}$$

Para prevenir que obtenga una velocidad muy alta el descenso y se pase del mínimo, se utiliza ϵ, con valores positivos relativamente cerca del 0.

- **Adagram**

Algoritmo de optimización de descenso de gradiente, en el que se adapta el coeficiente de aprendizaje con unos parámetros aplicando pequeñas variaciones a este.

g_t: es la variación de un peso determinado para disminuir el error. El término t representa cuándo se ha actualizado el peso. Por lo tanto, con esta notación el descenso de gradiente estocástico sería:

$$g_t = \frac{\partial E_{total}}{\partial w_{m,n}} \Longrightarrow w_{t+1} = w_t - \eta g_t$$

El algoritmo de optimización Adagram tratará de modificar el coeficiente de aprendizaje dependiendo de los anteriores gradientes g_t donde t representa el orden de ejecución de esos gradientes.

G_t: es un matrix diagonal donde la posición i, i es la suma de los cuadrados de los gradientes del peso que queremos cambiar hasta el último cambio anterior t.
ϵ: término que previene la división por 0. Es un número muy bajo como 10^{-8}.
Se tendrá que calcular el producto vectorial entre las matrices G_t y g_t como se muestra a continuación:

$$w^l_{m,n} = w^l_{m,n} - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

El beneficio principal del algoritmo Adagram es la eliminación de la necesidad de cambiar manualmente el coeficiente de aprendizaje.

Adadelta, ADAM, Nadam y Nesterov Accelerated Gradient son otros ejemplos de algoritmos de optimización de descenso de gradiente. Sin embargo existen muchos más y variantes de los mencionados.

3.4. Interfaz Gráfica

La aplicación cuenta con una sofisticada interfaz para simplificar los controles y funciones del programa que se describen a continuación. Además, utiliza la librería gráfica integrada de Java denominada *Swing*. Los campos de texto, botones, editores y menús han sido creados con un diseño simple y útil y siempre funcionando de manera fluida.

El manual técnico del funcionamiento de la aplicación y los primeros pasos se puede conseguir en el siguiente enlace: `https://docs.google.com/document/d/1RCckXaYssdckRxK-5rSRjsPZ1_A6wLahVYPceHjJPkk/edit?usp=sharing`.

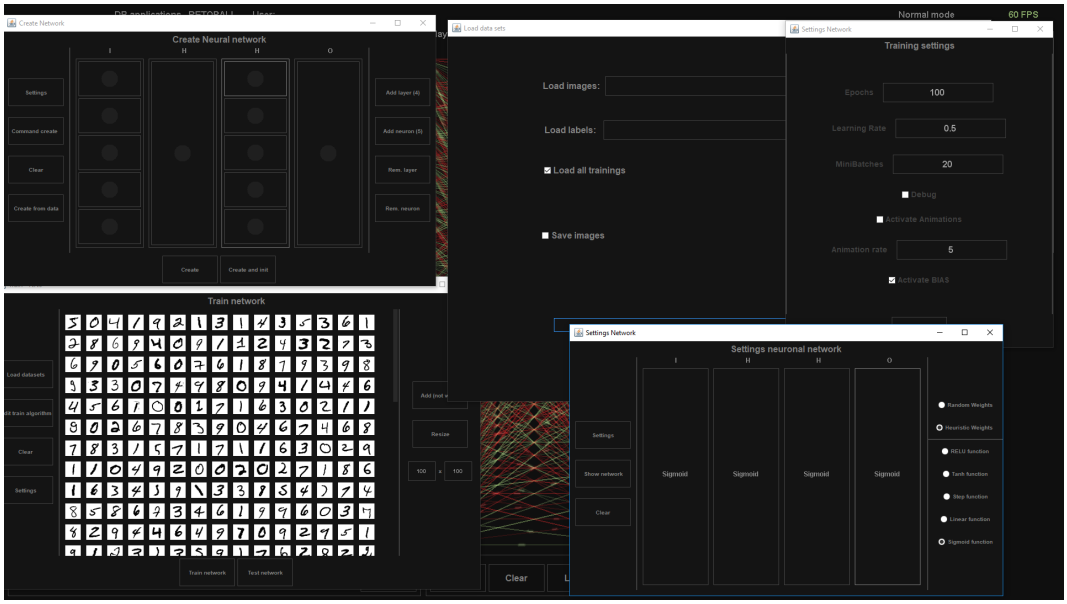


Figura 7: Ventanas del editor de redes neuronales

3.4.1. Ventana de creación

En la parte superior izquierda se puede observar un editor de redes neuronales en la que el usuario puede configurar de forma visual una red neuronal. Ofrece la posibilidad de creación por comandos o a partir de los conjuntos de datos

3.4.2. Ventana de importación

En la zona superior central se puede apreciar una ventana en la que se pueden importar de forma rápida conjuntos de datos para entrenar a la red neuronal. Las redes neuronales pueden ser modificadas editando el ritmo de aprendizaje, las iteraciones y las activaciones.

3.4.3. Ventana de aprendizaje

En el panel inferior izquierdo se pueden visualizar los distintos conjuntos de datos, en este caso para entrenar a una red neuronal con dígitos. Este es el panel de entrenamiento. Se puede seleccionar un único ejemplar para entrenar o todos los ejemplares. Ofrece la posibilidad de cambiar el tamaño de las imágenes y de configurarlas.

3.4.4. Simulaciones

También existe una ventana para probar distintas simulaciones y ver los resultados, la precisión y el rendimiento de la red neuronal. La numeración de cada fórmula está compuesta por dos números: el de la sección, dentro de la cual está la fórmula,

y el número que ocupa la fórmula dentro de esta sección. (Se pueden numerar de otras formas, por supuesto).

3.5. Proceso de renderización y actualización

Además de toda la interfaz y todos los algoritmos que utiliza, también es posible visualizar y entender el comportamiento de la red neuronal. Para ello, integra un motor gráfico programado desde cero para renderizar la red neuronal, efectos, textos...

```
1  //RENDERING NEURONS
   for (Layer l : Main.neuralNetwork.get(x).getLayers()) {
3  if (Main.neuralNetwork.get(x).isShowInputs()) {
   for (GraphicalNeuron gn : l.getNeuronas()) {
5  gn.render(g);
   }
7  } else {
   if (l.getPos() != 0) {
9  for (GraphicalNeuron gn : l.getNeuronas()) {
   gn.render(g);
11 }
   }
13 }

15 //RENDERING WEIGHTS
   ArrayList<Weight> weights = Main.neuralNetwork.get(x).getWeights();
17 if (Main.neuralNetwork.get(x).isShowInputs()) {
   for (int i = 0; i < weights.size(); i++) {
19 Weight w = weights.get(i);
   w.render(g);
21
   }
23 //EFFECTS
   int transparency = (int) (30 * Math.cos(s) + 60);
25 float rest = t %(float) (Math.PI);
   float rest3 = Math.abs((t*2) %(float) (Math.PI));
27 if (rest2 <=0.02f&&t>0.1f) {
   t = 0f;
29 }
   length = (int) (Math.sin(rest2) * initialLength);
31
   float pos = (float) (Math.sin(resT) * (module - length / 2));
33
   float xEffect1 = x1 + ((module - pos) * (x2 - x1)) / module;
35 float yEffect1 = y1 + ((module - pos) * (y2 - y1)) / module;
   float xEffect2 = x1 + (((module - pos - length)) * (x2 - x1)) / module;
37 float yEffect2 = y1 + (((module - pos - length)) * (y2 - y1)) / module;
   paintEffect(g, (int) xEffect1, (int) yEffect1, (int) xEffect2, (int)
       yEffect2);
39 }
```

Todos los algoritmos están ejecutados en distintos hilos de ejecución para prevenir

la disminución del rendimiento de la aplicación y a la vez gestionar varias redes, entrenamientos y otras tareas que requireren gran rendimiento computacional.

Este estará relacionado con la cantidad de objetos a renderizar. Cuanto más grande sea la red neuronal, más trabajo le costará al ordenador mantener la fluidez del programa. Por ello, si los *FPS*[12] decrecen demasiado, se reducirá la tasa de refresco del programa. La aplicación deja de garantizar una fluidez continua cuando se superan los 10.000.000 objetos a actualizar y los 100.000 a renderizar.

3.6. Almacenamiento de redes neuronales

Al entrenar una red neuronal, se dispone de la herramienta de guardar esa red neuronal en un ordenador para que posteriormente pueda ser utilizada por otras personas y/o por el usuario.

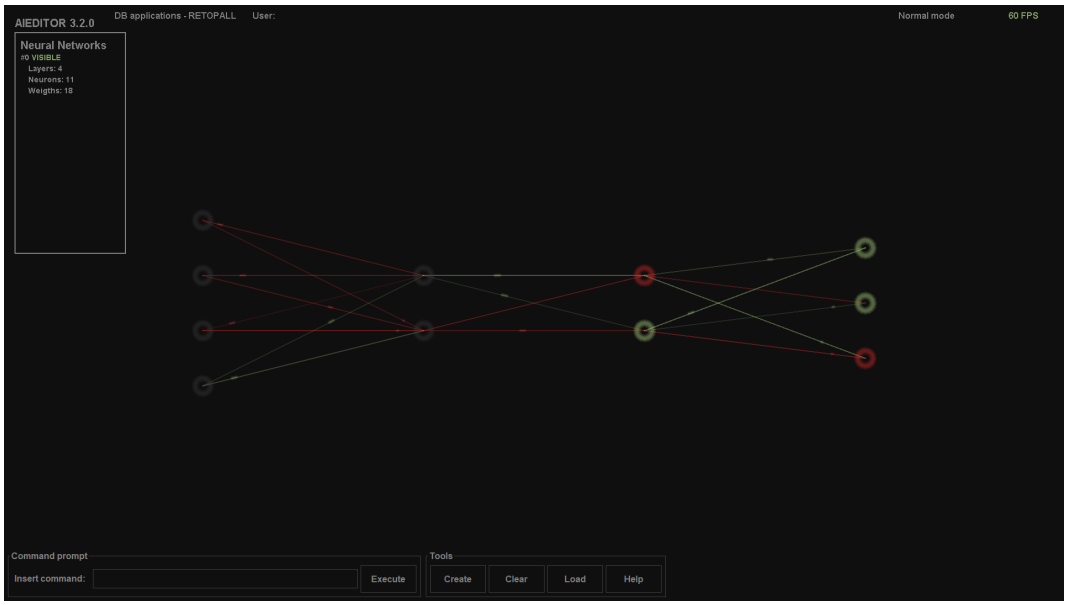


Figura 8: Red neuronal básica

Se guardará todo el aprendizaje, ajustes, y todas las neuronas, pesos, capas, etcétera. La red neuronal será serializable para poder guardarse en forma de objeto. Además, para garantizar la seguridad de los datos, todo estará encriptado de forma dinámica durante la ejecución del programa

[12] *Frames per second* o gráficos que puede renderizar un ordenador en un segundo

4. Prácticas

Además de programar los algoritmos y diseñar la aplicación he estado sacando provecho de ella creando distintas redes inteligentes y evaluándo sus rendimientos. He desarrollado una red neuronal preparada para reconocer dígitos y letras escritos manualmente.

Las imágenes usadas para entrenar a la red son de 28x28 pixeles pasadas a blanco y negro. Por lo tanto, la cantidad de neuronas en la capa de entrada será de 784 neuronas. Las neuronas de salida serán los dígitos del 0 al 9. He estado variando las capas ocultas y finalmente he conseguido obtener el mejor rendimiento con una red de 784x20x20x10 y que funcionaba con una precisión de reconocimiento del 99 %. La red tiene 16280 pesos a modificar en el entrenamiento.

Para analizar el rendimiento de la red, he implementado en el programa una librería que he programado en Java y que incluye técnicas de análisis matemático, representación de funciones, matrices y otras herramientas. Se puede descargar la aplicación y ver el código fuente en `https://github.com/dDevTech/MathLibrary`. La función coste de la red neuronal se puede visualizar en la figura 4. La función polinómica es la función del error con la herramienta de regresión aplicada.

Con lo que he llegado a aprender de inteligencia artificial con este proyecto he creado un simulador de coches autónomos utilizando redes neuronales, pero esta vez sin back-propagation y con aprendizaje de esfuerzo y en concreto con algoritmos genéticos. Se puede ver una demo del simulador en: `https://www.youtube.com/watch?v=aC7Euu4s66A`
El programa está escrito en C# con el motor gráfico Unity 3D [Figura 9].

5. Conclusión

Gracias a esta aplicación he podido conocer a fondo el funcionamiento de las redes neuronales, además de poder haberla utilizado para otros fines y aplicaciones. Por ejemplo a algoritmos genéticos. He observado, además, que la forma más rápida de aprendizaje para números es a través de minibatches, con un coeficiente de aprendizaje bajo (0.1-0.2) y con la función de activación sigmoid.

La aplicación está disponible en mi página web por lo que cualquier persona interesada se la puede descargar. Actualmente estoy desarrollando, gracias a los conocimientos aprendidos con este proyecto, otras aplicaciones de inteligencia artificial utilizando algoritmos genéticos y otros modelos de redes neuronales como *Adagram* para hacer aplicaciones más inteligentes. Sin embargo continúo desarrollando esta aplicación con nuevas características para que cualquiera pueda aprovechar *AIEditor* .

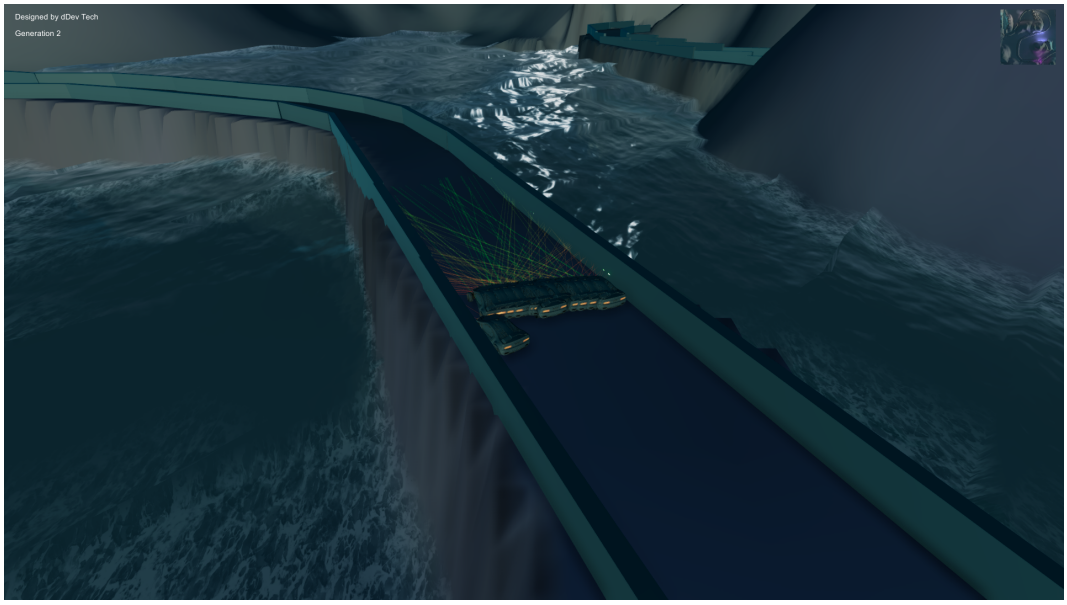


Figura 9: Simulador 3D de coches autónomos

Agradecimientos

Este proyecto no habría podido llevarse a cabo sin el apoyo de algunas personas que me han ayudado en mis dudas y se han esforzado en revisar el proyecto.
En primer lugar a María Gaspar y a Irene Gutiérrez por revisar el proyecto varias veces y darme su opinión de él.
A Juan Carlos Berrocal por escucharme cuando le explicaba la aplicación sin tener conocimientos de programación.
No cabe olvidar a Javier Martín por ayudarme en las dudas que me han ido surgiendo durante la elaboración del proyecto, junto con María Gaspar.

Referencias

[1] Matt Mazur *Step by step propagation example* URL: `https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/`

[2] 3Blue1Brown *Explaining Neural Networks* URL: `https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi`

[3] deeplearning.ai *Mini-batch gradient descent* URL: `https://www.coursera.org/lecture/deep-neural-network/mini-batch-gradient-descent-qcogH`

[4] Michael Nielsen *Using Neural Nets to recognize hand written digits* URL: `http://neuralnetworksanddeeplearning.com/chap1.html`

[5] Data Science Group *Loss function and optimization algorithms* URL: `https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c`

[6] Toward Data Science *Optimization algorithms used in Neural Networks*

[7] Sebastian Ruder *An overview of gradient descent optimization algorithms* URL: `http://ruder.io/optimizing-gradient-descent/`

[8] Diego Berrocal *Editor de Inteligencia Artificial* Figuras 1,2,4,7,8 de elaboración propia.

[9] Diego Berrocal *Simulador de coches autónomos en Unity 3D* Figura 9 de elaboración propia.

[10] *Descenso de gradiente* Figura 3 Ir a la imagen

[11] *Algoritmos de optimización* Figura 5 Ir a la imagen

[12] *Momentum* Figura 6 Ir a la imagen

Anexos

A continuación añadiré enlaces de interés relacionados con el proyecto, su descarga y plataformas para visualizar el código de las aplicaciones y librerías utilizadas. Además, incluiré anexos importantes de programación del proyecto.

Anexo I: Enlaces de interés

DDev Tech: Canal de youtube donde se pueden ver las demos de las aplicaciones y los proyecto que desarrollo.
`https://www.youtube.com/channel/UCwsrCNLN4xQE9OVSP5Sz2dA`

DDev Tech: código de proyectos de programación como la librerís de matemáticas o de conexión al servidor.
`https://github.com/dDevTech`

Retopall: Sitio web donde publico mis proyectos y las descargas de estos.
`https://retopall.com/`

Anexo II: Imágenes del editor de redes

Las siguiente imágenes son capturas de la aplicación AIEditor:

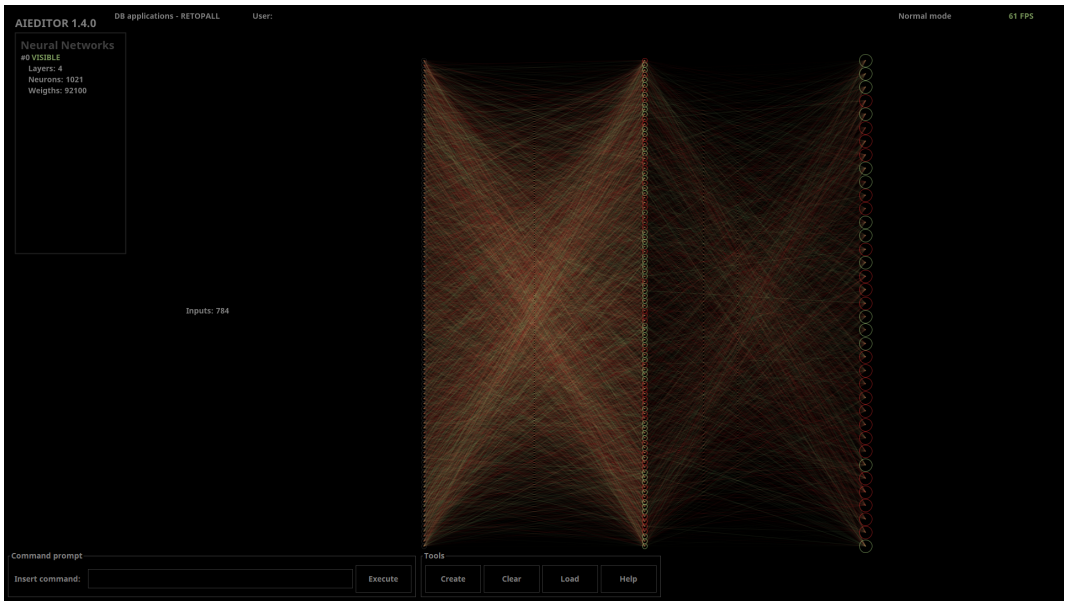


Figura 10: Red neuronal 784x100x100x27

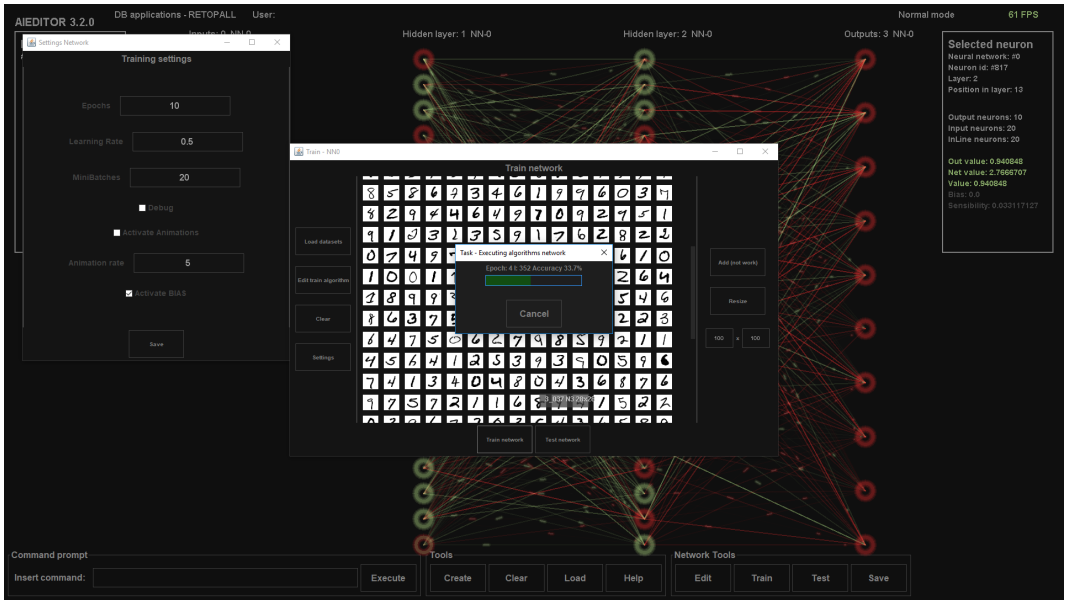


Figura 11: Entrenamiento de la red neuronal

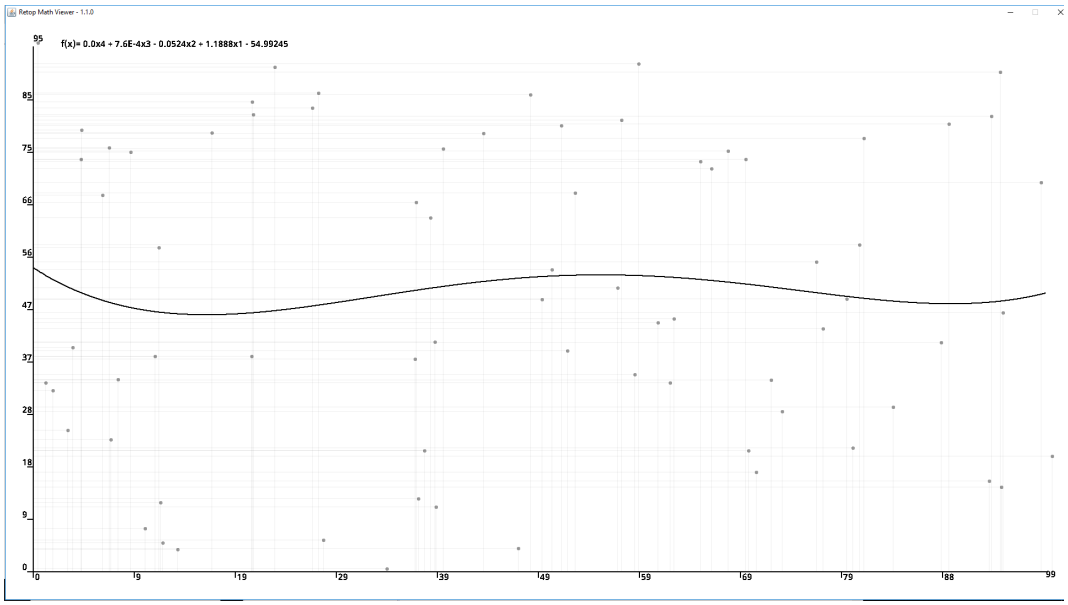


Figura 12: Regresión de un conjunto de datos

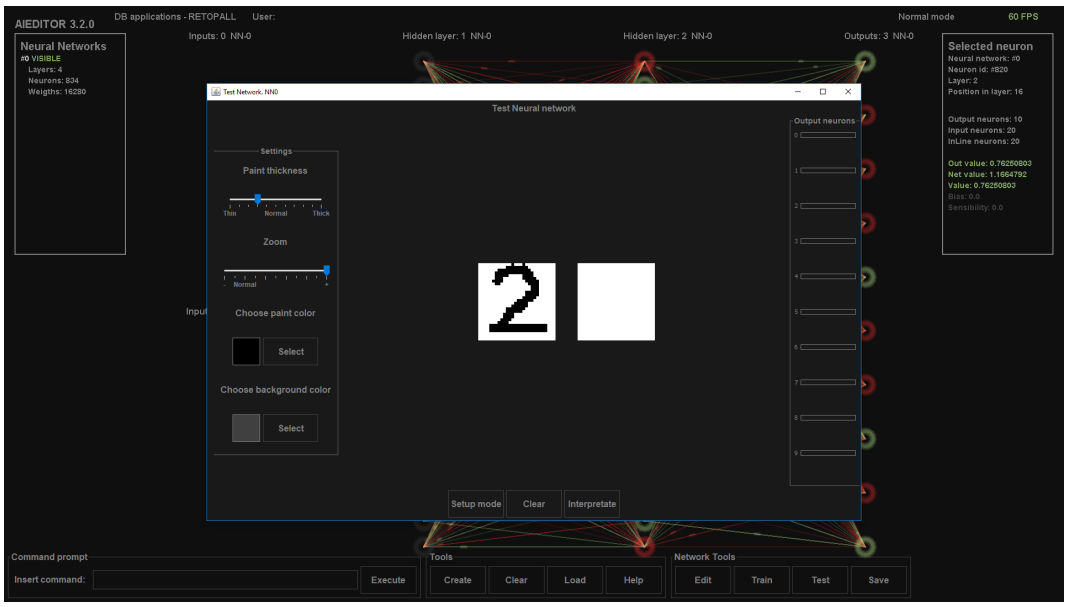


Figura 13: Dibujando números para probar el funcionamiento de la red neuronal