



Universitat d'Alacant
Universidad de Alicante

Sistemas inteligentes
Grado Ingeniería Robótica

Práctica 2

Redes neuronales
Detección de género

Carmen Ballester Bernabeu

[Enlace al github de la práctica](#)

Índice

Índice	2
Introducción al problema	3
Preparación del dataset	3
Librerías utilizadas	3
Código del algoritmo	4
Espacios de color y canales	5
Implementación de la red neuronal	6
Cargar dataset	6
Definir la estructura de la red neuronal	7
Fundamento teórico	7
Código	10
Definir la estrategia de entrenamiento y la selección del modelo	11
Fundamento teórico	11
Código	15
Analizar los resultados de la red entrenada	15
Fundamento teórico	15
Código	17
Resultados del entrenamiento y validación	18
Quasi-Newton Method	18
Normalized Squared Error	19
Minkowski Error	19
Mean Squared Error	20
Weighted Squared Error	21
Tabla de resultados	22
Gradient Descent	22
Normalized Squared Error	23
Minkowski Error	23
Mean Squared Error	24
Weighted Squared Error	25
Tabla de resultados	25
Detección de caras	26
Código	26
Resultados	27
Conclusiones	30
Bibliografía	30

Introducción al problema

El objetivo del desarrollo de esta práctica es crear una red neuronal que sea capaz de determinar el sexo de las personas que aparecen en una imagen. Este problema está ampliamente resuelto en la actualidad, pero es necesario sopesar la relación entre precisión de los resultados y tiempo de cómputo, ya que cuanto mejores resultados deseemos más capacidad computacional se requiere.

Para programar esta red se utiliza la librería **Open Neural Network Library - OpenNN**, que es una librería opensource escrita en C++ que implementa redes neuronales, y que además puede integrarse con otras herramientas como *QT Creator*, que es el IDE que se utiliza para el desarrollo de la práctica.

El dataset que se va a utilizar está constituido por 303 imágenes, de las cuales 101 pertenecen a hombres, 101 a mujeres y 101 son imágenes sin persona. Como el dataset no es muy grande, es necesario dividirlo en tres conjuntos de datos, de forma que el sistema no aprenda todas las imágenes. Estos conjuntos son:

- **De entrenamiento:** es el que se utiliza para entrenar la red y actualizar los pesos, por lo que tiene que ser el de mayor tamaño, un 70% del dataset.
- **De prueba:** es el que se utiliza al final de cada época para analizar cómo de bien trabaja la red ante caras nuevas, de forma que nos aseguramos de que es capaz de generalizar correctamente. Constituye un 15% del dataset.
- **De validación:** es el que se utiliza cuando la red está totalmente entrenada, por lo que nos proporciona el error final de validación de esta. Lo forma un 15% del dataset.

Una vez que la red se ha entrenado y probado con fotos del dataset, hay que comprobar su eficacia en un escenario más real, por lo que tiene que ser capaz de determinar el género en una imagen donde hayan varias personas.

Preparación del dataset

La red neuronal se entrena con una serie de imágenes de caras de hombres, de mujeres y sin persona. Para crear nuestro dataset en un fichero .csv que luego pueda ser cargado como dataset, hay que leer las imágenes, extraer las características y almacenarlas en el fichero.

Librerías utilizadas

Para tratar con imágenes, necesitamos usar librerías específicas que permiten leer imágenes, redimensionarlas, extraer características, etc. Para este propósito existen varias librerías.

El primer intento fue utilizar el conjunto de librerías **OpenCV**, ya que permiten la manipulación de imágenes y se utiliza mucho en todo el ámbito de visión por computador.

Sin embargo, al incluir las librerías en el proyecto de Qt aparecían una serie de errores de compilación y de ejecución, además de ciertos *warnings*.

Por este motivo, se han usado las librerías propias de Qt para el tratamiento de imágenes, **QImage**. Estas librerías están optimizadas para el proyecto, y además tienen una amplia documentación y una serie de ejemplos que son muy útiles a la hora de escribir nuestro código. Se utilizan otras librerías como **QFile**, **QTextStream** y **QDir** para el tratamiento del archivo .csv en el que vamos a escribir los datos. Para realizar la conversión entre espacios de colores y guardar diferentes canales de información de las imágenes, se utiliza la clase **QColor**. Todas estas clases tienen unos métodos que permiten trabajar con el código de forma muy sencilla y modulada.

Código del algoritmo

El pseudocódigo del algoritmo es el siguiente:

```
generarDataset:
    obtener la ruta de la carpeta dataset
    crear el fichero csv
    crear un vector con los nombres de las carpetas de las clases

    Para cada carpeta de clases
        obtener los nombres de todas las imágenes de la clase

        Para cada imagen de la clase
            redimensionarla

            Para cada pixel
                obtener el color rgb
                obtener la información de un canal de un espacio de color
                escribir la información en la misma línea del archivo csv
            fin Para

            escribir la clase de la imagen en el archivo csv y "\n"
        fin Para
    fin Para

    cerrar el fichero csv
fin generarDataset
```

Para obtener el directorio de la carpeta dataset se utiliza el método *getExistingDirectory*, de la clase **QFileDialog** proporcionada por Qt. Este método despliega una ventana en tiempo de ejecución, de modo que podemos navegar por nuestras carpetas y seleccionar la adecuada. Hay que tener en cuenta que en esta implementación tanto el nombre de las carpetas de clase como el del fichero csv generado son siempre los mismos, por lo que no se pide al usuario que los introduzca, sino que se definen las rutas de cada uno de los archivos en tiempo de compilación. Como la ruta de la carpeta dataset es una entrada y el nombre del resto de carpetas y del archivo csv son siempre los mismos, ya tenemos definida la ruta de las carpetas que contienen las imágenes.

Para cada carpeta que contiene las imágenes de una clase, tenemos que obtener el nombre de todas las imágenes. Tenemos la ruta del directorio de la carpeta de clase, obtenida según lo explicado en el párrafo anterior, definida en un objeto de la clase **QDir**. Los nombres de los archivos que contiene esa carpeta se obtienen utilizando el método *entryList* de esa clase, y se almacenan en un objeto de tipo **QStringList**. Hay que tener en cuenta que este método muestra todos los archivos de la carpeta, por lo que se únicamente queremos que muestre los archivos jpg hay que utilizar el método *setNameFilters* de **QDir** para establecer esta condición.

Una vez obtenidas las rutas de todas las imágenes que componen el dataset, simplemente tenemos que cargarlas una a una y extraer su información. El primer paso es redimensionar las imágenes, ya que a pesar de que no tienen una gran resolución (250x250), el proceso para generar el dataset es demasiado costoso computacionalmente. Por este motivo, se reduce el tamaño de las imágenes a 50x50 píxeles. La información que vamos a almacenar de las imágenes simplemente va a ser el valor de intensidad de cada píxel. Por lo tanto, según el tamaño definido anteriormente vamos a tener 2500 datos de entrada (**variables input**) a nuestra red.

Almacenamos esta información en el formato dat, es decir, cada línea (fila) del archivo csv corresponde a una imagen, y cada columna representa el valor de intensidad de un píxel. Como todas las imágenes se recorren en el mismo sentido, cada columna va a recoger el valor de intensidad del píxel (i,j) de cada imagen del dataset. Añadimos la clase de la imagen utilizando un sistema de *true/false*. Como queremos clasificar las imágenes en *hombre*, *mujer* y *sin persona*, la red va a tener tres neuronas en la última capa, que es la capa probabilística. Por lo tanto, en el dataset necesitamos tener tres variables de salida (**variables target**). Cada nueva columna del dataset representa respectivamente las clases *hombre*, *mujer* y *sin persona*. Se pone a 1 la columna que representa la clase de esa imagen y a 0 las otras dos. Por lo tanto, cada está representada por los tres últimos números de la fila tal que: 1 0 0 es *hombre*, 0 1 0 es *mujer* y 0 0 1 es *sin persona*.

Para escribir en el archivo csv se utiliza la clase **QTextStream**, que permite de forma muy sencilla escribir información en un fichero cualquiera.

Espacios de color y canales

Según esta forma de almacenar la información de cada imagen, tenemos muchas opciones posibles para almacenar los valores de intensidad de los píxeles. Podemos jugar con los diferentes espacios de color y con sus respectivos canales. Para trabajar con los colores de la imagen, se utiliza la clase **QColor**, que tiene una gran variedad de métodos para cambiar entre espacios de color y para obtener información de un solo canal.

Implementación de la red neuronal

Cargar dataset

El primer paso para poder entrenar una red neuronal es crear un objeto de la clase *Dataset* que codifique y que permita utilizar la información que está recopilada en el fichero csv del dataset en sí. Para ello OpenNN tiene una clase propia que se llama **Dataset**, que permite gestionar la carga del dataset. Además, podemos definir información para cada una de las variables que componen el dataset, cómo puede ser asignarles un nombre y una unidad de medida. Las variables son cada una de las columnas del dataset, y las instancias son cada una de las filas.

Como se ha mencionado anteriormente, el dataset está formado por 2500 variables de entrada, que corresponden a cada uno de los píxeles de una imagen; y por 3 variables de salida, correspondientes a la información de clase. A pesar de que la información asociada a cada variable se puede añadir de manera opcional, es recomendable hacerlo porque la información del dataset queda mucho más clara en el archivo que se genera al guardarlo. Además, es importante definir para cada una de las variables si se trata de una variable de entrada o de salida. Para modificar estos atributos se utiliza un puntero a una variable de tipo **Variable**. A partir de los métodos de esta clase es como podemos modificar los atributos de las variables.

Una vez hemos cargado el dataset a partir de un archivo csv y hemos definido toda la información necesaria, hay que dividirlo en tres conjuntos: entrenamiento, validación y test. El conjunto de **entrenamiento** se utiliza para modificar los pesos de la red, de forma que sean capaces de clasificar correctamente las imágenes. El conjunto de **validación** se utiliza tras cada etapa de entrenamiento para que la red no pierda la capacidad de generalización. El conjunto de **test** se utiliza una vez se ha acabado el entrenamiento de la red para obtener los resultados finales.

El conjunto de entrenamiento es siempre el mayor, y normalmente se divide el dataset con un 60% para entrenamiento, 20% para validación y 20% para test. En este caso, dividimos el conjunto de entrenamiento tal que el 70% es el conjunto de entrenamiento, el 15% es el conjunto de validación y el 15% es el conjunto de test. Esta división se lleva a cabo utilizando un método que la realiza de forma aleatoria.



Figura 1. Esquema del porcentaje de cada uno de los conjuntos de entrenamiento, validación y test

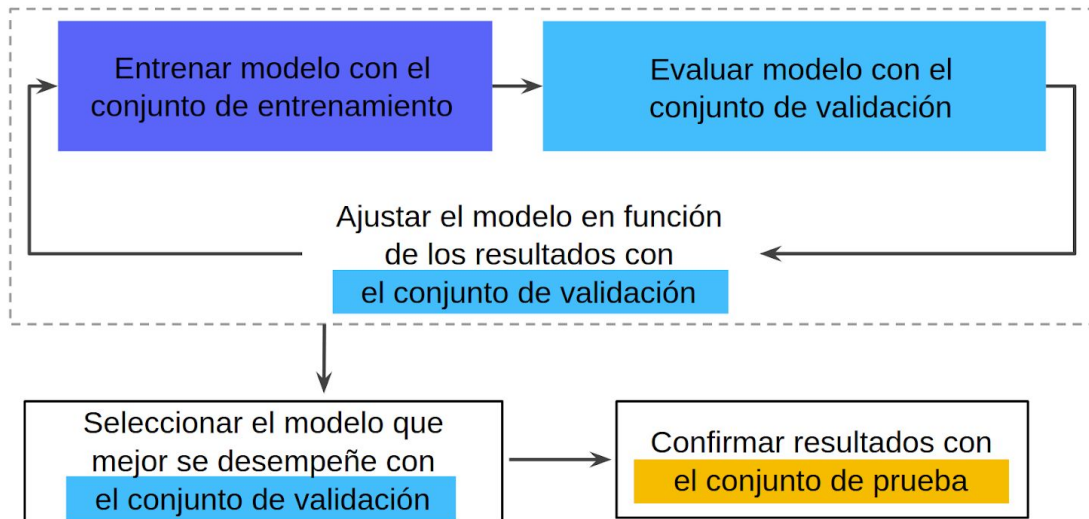


Figura 2. Esquema de la función de cada uno de los conjuntos de entrenamiento, validación y test

El pseudocódigo para cargar el dataset correctamente según todo lo especificado anteriormente es:

```
cargarDataset:
    especificar la ruta del archivo que se va a cargar
    cargar la información del dataset

    Para cada instancia del dataset
        definir el nombre de esa variable y las unidades
        definir si es entrada o salida
    fin Para

    obtener la información de las variables y mostrarlas por pantalla
    dividir los datos en los conjuntos de entrenamiento, validación y test
fin cargarDataset
```

Definir la estructura de la red neuronal

Fundamento teórico

Para entender el funcionamiento de cualquier red neuronal es necesario conocer cómo funciona el **perceptrón simple**, que es la unidad más básica de una red neuronal. Un perceptrón simple no es más que la suma ponderada de una serie de entradas. El resultado de esta suma ponderada es la entrada a una **función de activación** para obtener la salida mejorada del perceptrón. Además, en la suma se puede añadir otro término que no multiplica por los pesos de las entradas y que recibe el nombre de **bias**. Por lo tanto, es posible obtener la salida del perceptrón simple a través de una serie de operaciones matemáticas.

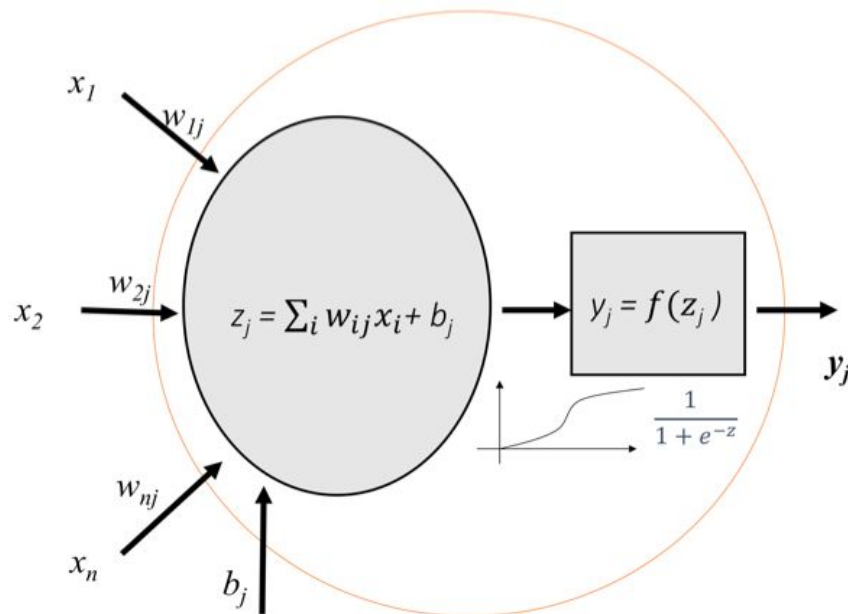


Figura 3. Esquema de un perceptrón simple con una función de activación.

El problema de utilizar un solo perceptrón simple es que únicamente sirve para clasificar conjuntos linealmente separables. *Rosenblatt* demostró que si los patrones usados para entrenar el perceptrón son sacados de dos clases linealmente separables, entonces el algoritmo del perceptrón converge y toma como superficie de decisión un hiperplano entre estas dos clases. La prueba de convergencia del algoritmo es conocida como el **teorema de convergencia del perceptrón**.

El perceptrón simple tiene sólo una neurona, por lo que está limitado a realizar clasificación de patrones con sólo dos clases. Expandiendo la capa de salida del perceptrón para incluir más de una neurona, podemos realizar dicha clasificación de más de dos clases siempre que sean linealmente separables.

Para solventar las limitaciones del perceptrón simple se crea un nuevo tipo de red neuronal, denominada **perceptrón multicapa**, que consiste en incluir capas ocultas de neuronas entre la capa de entrada y la capa probabilística de salida.

El perceptrón multicapa con una única capa de neuronas ocultas es capaz de discriminar regiones lineales convexas para la clasificación, y el perceptrón multicapa con dos capas de neuronas ocultas es capaz de discriminar regiones de forma arbitraria. Por lo tanto, un perceptrón multicapa es capaz de aproximar cualquier función continua en un intervalo hasta el nivel deseado, y fue demostrado por *Funahashi* en 1989.

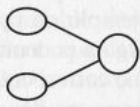
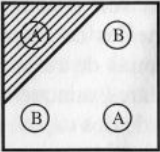
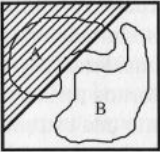
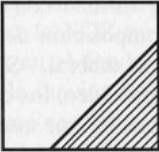
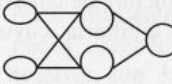
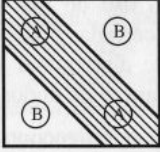
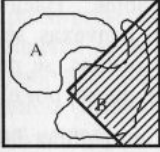
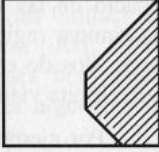
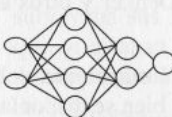
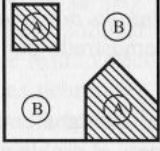
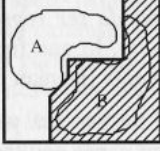
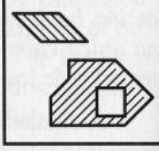
Arquitectura	Región de decisión	Ejemplo 1: XOR	Ejemplo 2: clasificación	Regiones más generales
Sin capa oculta 	Hiperplano (dos regiones)			
Una capa oculta 	Regiones polinomiales convexas			
Dos capas ocultas 	Regiones arbitrarias			

Figura 4. Tabla comparativa entre diferentes arquitecturas de redes neuronales

Por lo tanto, podemos definir un perceptrón multicapa o **Multilayer Perceptron (MLP)** como un conjunto de capas de neuronas ocultas, que matemáticamente se pueden tratar como múltiples capas de nodos de un grafo dirigido donde cada capa oculta está totalmente conectada a la siguiente.

Para el problema de la detección de género vamos a usar un modelo MLP, ya que vamos a tener que separar entre regiones complejas que no se pueden dividir utilizando un hiperplano. Dentro de los modelos MLP, hay que comprobar si con una única capa oculta podemos conseguir aproximaciones polinomiales decentes o si por el contrario es necesario una segunda capa oculta.

Una diferencia entre este problema y los esquemas anteriores es que en este caso no tenemos una clasificación pura, sino que hay que distinguir entre tres clases. Por lo tanto, la última capa se trata de una **capa probabilística** en la que vamos a obtener la probabilidad de pertenencia a cada clase. Como hay tres clases, es necesario que la capa de salida tenga tres neuronas.

Parte del desarrollo de la práctica consiste en probar diferentes arquitecturas para comprobar cuál es la mejor arquitectura. Por lo tanto, en el esquema general siguiente hay que tener en cuenta que tanto el número de neuronas en cada capa oculta como el número de capas ocultas en sí son variables. Además, la arquitectura de la red suele ser muy dependiente del tipo de estrategia de entrenamiento, por lo que unas arquitecturas que pueden funcionar bien para cierto tipo de estrategia de entrenamiento no funcionan tan bien para otra.

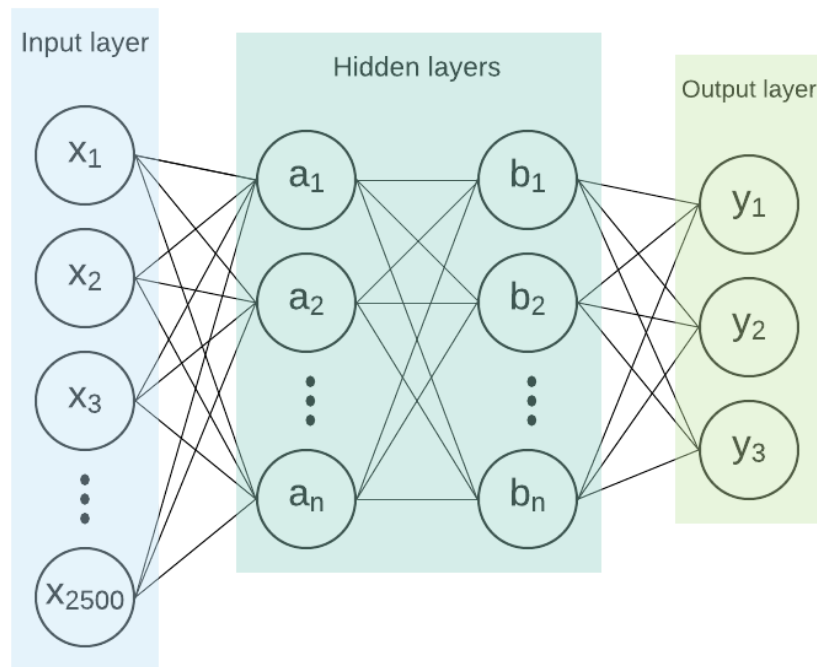


Figura 5. Esquema general de las MLP utilizadas

Normalmente es necesario además añadir una **capa de escalado** de las entradas, para que estén en un rango adecuado. Normalmente esta capa se representa directamente como la capa de entrada, y existen diversos métodos de escalado de las entradas como de mínimo y máximo, media y desviación, etc. También es necesario una capa de desescalado para que la salida esté en las unidades originales del problema.

Código

Para implementar la red neuronal vamos a utilizar, como se ha dicho antes, la librería *OpenNN*. Esta librería tiene una clase que permite crear una red neuronal y definir sus parámetros y arquitectura, la clase **NeuralNetwork**. Se crea la red a partir de un vector de enteros, donde el tamaño representa el número de capas, cada entero representa el número de neuronas de esa capa, y además el primer elemento corresponde a la capa de entrada y el último elemento a la capa de salida.

Además, hay que asignar el dataset generado anteriormente a esta red neuronal. Para ello, la clase contiene una serie de métodos que permiten, además de asignar el dataset, obtener información de la arquitectura y el estado de la red. Los métodos de la clase se encuentran disponibles en la [documentación de OpenNN](#).

Para manejar la asignación de las entradas y las salidas y la definición de las capa de escalado y la probabilística, se usan también otras clases como: **Inputs**, **Outputs**, **ScalingLayer** y **ProbabilisticLayer**.

El pseudocódigo de la función para crear la arquitectura de la red neuronal es el siguiente:

```
redNeuronal:  
  definir el número de neuronas de las capas de entrada y salida  
  crear la red con arquitectura MLP  
  
  asignar la información del dataset a las entradas y las salidas  
  
  definir la capa de escalado  
  definir la capa probabilística  
fin redNeuronal
```

Definir la estrategia de entrenamiento y la selección del modelo

Fundamento teórico

El proceso por el que la red aprende, es decir, actualiza los pesos de las neuronas para que la clasificación sea correcta, recibe el nombre de **estrategia de entrenamiento**. Por lo tanto, la estrategia de entrenamiento es un algoritmo que se le aplica a la red neuronal para conseguir el menor error posible, encontrando los pesos de la red que mejor encajan con la clasificación de los datos proporcionados en el dataset. La estrategia de entrenamiento está formada por el *algoritmo de entrenamiento* y por el *índice de pérdida*.

El **algoritmo de entrenamiento** (*training strategy*) define la forma en la que se va a intentar reducir el error en cada iteración. Normalmente, la red se considera entrenada cuando se obtiene un mínimo en el índice de pérdida, es decir, cuando el gradiente del índice de pérdida es cero. El problema es que el índice de pérdida normalmente es una función no lineal, lo que dificulta el uso de algoritmos de optimización convencionales. El problema se resuelve utilizando algoritmos que funcionan por épocas, de manera que el objetivo es reducir el error de manera progresiva en cada época hasta llegar a un mínimo del índice de pérdida. Estos algoritmos empiezan con una inicialización aleatoria de los pesos de la red y los modifican en cada época. Esta modificación se conoce como el *incremento de los parámetros*.

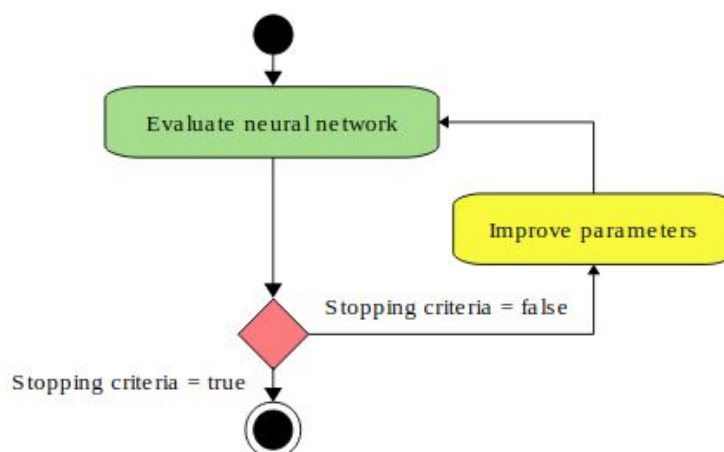
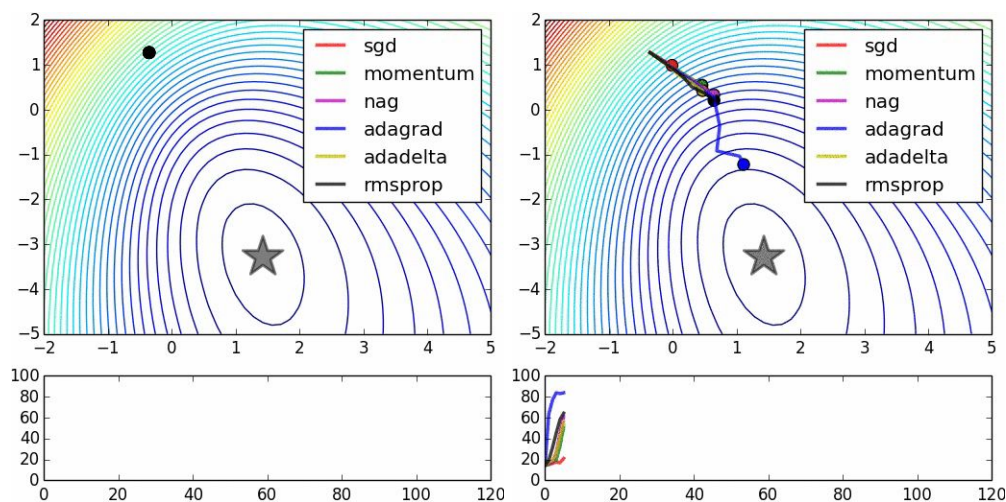
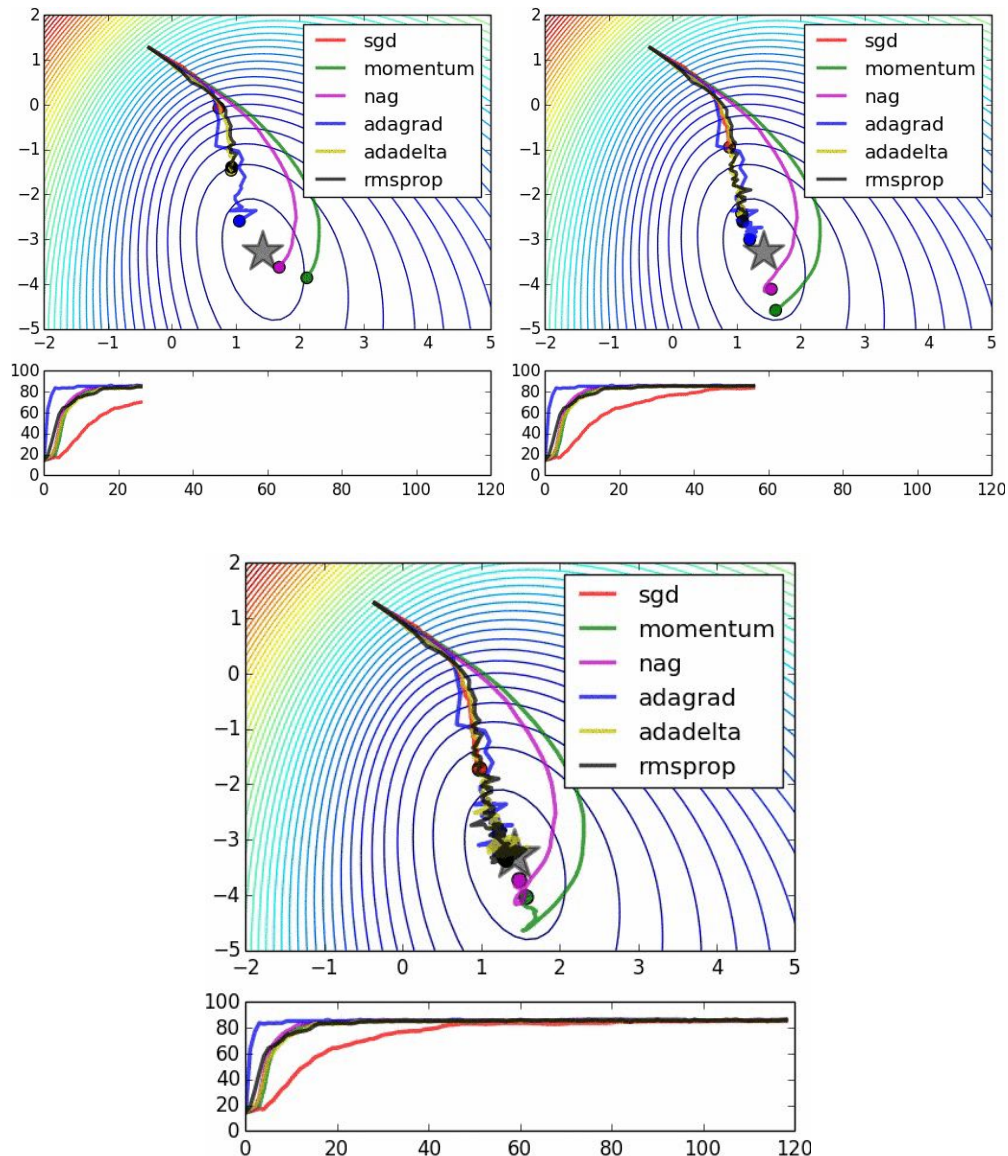


Figura 6. Esquema del entrenamiento de una red neuronal

Hay una variedad de estrategias de entrenamiento que difieren en la forma en la que se calculan los nuevos parámetros de la red. Los más utilizados, y los que están implementados en OpenNN, son:

- **Gradient descent (GD)**: es el algoritmo de entrenamiento más sencillo y consiste únicamente en actualizar los parámetros en cada época en la dirección negativa del gradiente del índice de pérdida.
- **Conjugate gradient (CG)**: es parecido a *gradient descent*, pero la búsqueda se realiza utilizando las direcciones conjugadas, lo que produce una convergencia de la actualización de los pesos más rápida que con *gradient descent*.
- **Quasi-Newton method (QNM)**: el método de Newton utiliza la *Hessiana* del índice de pérdida para calcular la dirección en la que se van a actualizar los pesos de la red. Cómo utiliza información de orden elevado (segundas derivadas de la función), se encuentran los mínimos del índice de pérdida con mucha precisión. El problema es que este método es muy costoso computacionalmente porque requiere calcular las segundas derivadas. Para solucionarlo se utiliza el método Quasi-Newton, que no requiere el cálculo de las segundas derivadas porque computa una aproximación de la inversa de la hessiana en cada época utilizando la información del gradiente.
- **Levenberg-Marquardt algorithm (LM)**: es un método de entrenamiento diseñado para alcanzar la precisión de segundo orden sin la necesidad de computar la hessiana, porque utiliza el gradiente de la matriz jacobiana del índice de pérdida. Se puede aplicar cuando el índice de pérdida tiene forma de una suma de cuadrados, como *Sum Squared Error (SSE)*, *Mean Squared Error (MSE)* y *Normalized Squared Error (NSE)*.
- **Stochastic gradient descent (SGD)**: este algoritmo consiste en actualizar los pesos de la red por lotes del conjunto de entrenamiento utilizando una aproximación estocástica (*aleatoria*) del gradiente.
- **Adaptive linear momentum (ADAM)**: es un algoritmo similar a *stochastic gradient descent*, pero implementa un método más sofisticado para calcular la dirección de entrenamiento que produce normalmente una convergencia más rápida.





Figuras 10 - 14. Comportamiento de varias estrategias de entrenamiento. [Gif aquí](#).

Arriba: recorrido en la función de *loss index*.

Abajo: porcentaje de acierto de la red.

El **índice de pérdida** (*loss index*) define la tarea que debe realizar la red neuronal y proporciona una medida de la calidad de la representación requerida. Está formado por dos términos: el de error y el de regulación.

El **término de error** mide la calidad de las predicciones de la red neuronal respecto a los datos del dataset. Se pueden medir los errores del entrenamiento y del test por separado. El error de entrenamiento se utiliza para actualizar los parámetros de la red, y el de test para comprobar el funcionamiento de esta una vez entrenada. Hay varias formas de medir estos errores, aunque las más utilizadas son:

- **Mean squared error (MSE)**: calcula la media del error al cuadrado entre las salidas de la red neuronal y las etiquetas del dataset.

- **Normalized squared error (NSE):** divide el error al cuadrado entre las salidas de la red neuronal y las etiquetas del dataset por el coeficiente de normalización. Si el error tiene un valor de cero significa que la predicción es perfecta.
- **Weighted squared error (WSE):** se utiliza para clasificaciones binarias en las que las clases de las dos clases en el dataset están muy desproporcionadas respecto al número de ejemplos. Pondera los errores según la clase a la que pertenecen.
- **Cross entropy error:** también se utiliza en problemas de clasificación en los que la etiqueta de la clase toma un valor de 0 o 1. Penaliza con un error muy grande cuando la predicción está muy alejada del valor de la clase a la que pertenece la instancia que se está probando.
- **Minkowski error (ME):** es la suma de la diferencia entre la salida de la red neuronal y la etiqueta de la clase elevada a un exponente conocido como el *parámetro Minkowski*, que varía entre 0 y 1.

Todas estas formas de calcular los errores permiten obtener el gradiente de forma analítica utilizando *backpropagation*.

El **término de regulación** mide los valores de los parámetros de la red neuronal, y consigue que la red neuronal esté más suavizada, ya que los pesos y el bias son más pequeños. Se dice que una solución es regular cuando pequeños cambios en las entradas producen pequeños cambios en las salidas. Este término está multiplicado por un parámetro, de forma que si la solución está muy suavizada el valor del parámetro disminuye, y si oscila mucho el valor del parámetro aumenta. Los dos principales tipos de regulación son:

- **Regulación L1:** consiste en la suma de los valores absolutos de todos los parámetros de la red neuronal.
- **Regulación L2:** consiste en la suma de los cuadrados de todos los parámetros de la red neuronal.

El índice de pérdida depende de la función representada por la red neuronal y se mide respecto al dataset. Podemos visualizarlo como una hipersuperficie que tiene los parámetros de la red como coordenadas.

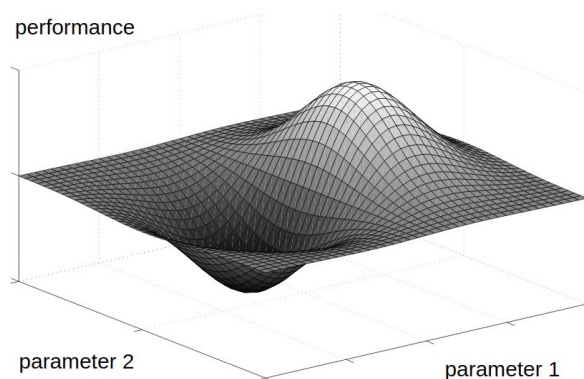


Figura 15. Hipersuperficie que representa el *loss index*.

Código

Para implementar el entrenamiento de la red vamos a usar dos clases principales: **TrainingStrategy** y **LossIndex**. Estas clases permiten definir cómo va a entrenar la red utilizando métodos que establecen la estrategia de entrenamiento y el índice de pérdida. Hay que tener en cuenta que una vez elegida una de las estrategias de entrenamiento y un índice de pérdida de entre las que están implementadas en OpenNN, existe una clase heredada para cada tipo de algoritmo que permite crear objetos de dicho algoritmo y establecer los parámetros propios del método utilizado.

Cuando se ha definido la estrategia de entrenamiento es cuando podemos utilizar la clase **ModelSelection** para encontrar la arquitectura de red que mejor va a funcionar con dicha estrategia.

En esta práctica partimos de redes neuronales de cero, es decir, que no están pre-entrenadas. Esto significa que la inicialización de los pesos de la red es aleatoria. Cómo realmente los computadores no son capaces de generar números aleatorios, sino que utilizan *Pseudo-Random Number Generator*, hay que establecer una semilla a partir de la cual se generan los números. Si queremos comparar varios modelos diferentes de forma coherente, es decir, que las pruebas sean **repetibles**, debemos empezar con los mismos parámetros de la red, por lo que la semilla de generación tiene que ser la misma. La semilla aleatoria para la generación de números aleatorios se puede establecer fácilmente utilizando la **librería estándar** de C++.

```
entrenarRed:
    establecer la semilla aleatoria

    definir la estrategia de entrenamiento:
        definir loss index
        definir training algorithm

    seleccionar la mejor arquitectura para la estrategia de entrenamiento
    definida (opcional)

    entrenar la red

fin entrenarRed
```

Analizar los resultados de la red entrenada

Fundamento teórico

Una vez la red ha terminado de entrenar, hay que evaluar su precisión con un conjunto de imágenes que no haya utilizado para el entrenamiento. Esto es debido a que puede producirse **overfitting**, es decir, que la red empieza a memorizar las imágenes del conjunto

de entrenamiento y pierde la capacidad de generalización. Cuando esto se produce, el error de entrenamiento disminuye mientras que el error de test y validación aumenta.

Aunque durante el entrenamiento se ha utilizado un conjunto de validación que ayuda a prevenir este error, sigue siendo obtener los resultados de la red con un conjunto de imágenes nuevo para poder tener los valores más fieles al funcionamiento de la red posible. Este conjunto de test es el que hemos separado aleatoriamente del dataset original con el que entrenamos y validamos la red.

El parámetro más obvio con el que podemos medir los resultados de nuestra red neuronal es con la **precisión**, que consiste simplemente en dividir el número de aciertos entre el número de muestras del test. El problema de usar este parámetro es que podemos encontrar ciertos sesgos en la red, es decir, puede tener una precisión relativamente buena pero tiende a confundir hombres con mujeres, por ejemplo. Por lo tanto, la precisión no aporta información individual acerca de cómo funciona la red respecto a cada clase.

Para recoger información de las predicciones de la red desglosadas por clases se utiliza la **matriz de confusión**. Esta matriz, que se podría entender también como una tabla, representa en filas y columnas las diferentes clases, de modo que normalmente las filas van a corresponder a las clases reales y las columnas a las clases predichas. En cada elemento de la matriz (i,j) se representa el número de veces que la red ha clasificado la clase i como clase j . De esta forma, cuando se cumple que $i=j$, es decir, en la diagonal principal de la matriz, tenemos los aciertos de la red.

		Predicho		
		Hombre	Mujer	Sin persona
Real	Hombre	a	d	g
	Mujer	b	e	h
	Sin persona	c	f	i

$a, b, c, d, e, f, g, h, i \in \mathbb{N}$

Figura 16. Esquema de la matriz de confusión para el problema dado

Normalmente, los problemas de clasificación no binaria de un número n de clases se pueden entender como n problemas de clasificación binaria a la hora de realizar el análisis. Cada clase puede entenderse como un problema independiente que consiste en detectar o no esa clase. Al reducir el problema a clasificación binaria para el análisis es cuando podemos detectar lo bien o mal que la red funciona con cada clase por separado.

En problemas de clasificación binaria tenemos dos tipos de error: **falso positivo**, en los que la clase se detecta (positivo) pero realmente la muestra no pertenece a esa clase; y **falso negativo**, en los que la clase no se detecta (negativo) pero realmente la muestra pertenece a esa clase. En según qué aplicaciones, los falsos positivos y los falsos negativos no tienen la misma importancia, un error es más grave que el otro. Por lo tanto, es muy importante definir otro tipo de métricas del error que permitan obtener información más específica sobre los tipos de error producidos en cada clase del problema.

A partir de la matriz de confusión obtenida podemos definir otros tipos de métricas para evaluar los resultados obtenidos:

- **Precisión:** mide el número de aciertos frente al total de predicciones sin distinguir entre clases.

$$precision = \frac{predicciones\ correctas}{total\ de\ predicciones} = \frac{a + e + i}{a + b + c + d + e + f + h + i}$$

- **Exactitud:** se tiene que calcular para cada clase. Representa de entre todas las muestras de la clase *x* que ha predicho la red cuántas pertenecen realmente a esa clase. Representa el acierto conseguido en la predicción positiva. La fórmula en el caso de la clase *hombre* es:

$$exactitud = \frac{predicciones\ de\ hombre\ correctas}{total\ de\ predicciones\ de\ hombre} = \frac{a}{a + b + c}$$

- **Sensibilidad:** también se tiene que calcular para cada clase. Representa de entre todas las muestras de la clase *x* reales cuántas han sido correctamente clasificadas. La fórmula en el caso de la clase *hombre* es:

$$sensibilidad = \frac{predicciones\ de\ hombre\ correctas}{total\ de\ muestras\ de\ hombre} = \frac{a}{a + b + c}$$

Código

Para implementar el test se utiliza la clase **TestingAnalysis**, que permite probar la eficacia de la red a partir del dataset y la estrategia de entrenamiento utilizadas para que la red aprenda. Tiene un método que directamente devuelve la matriz de confusión. A partir de la matriz de confusión, que se devuelve realmente en forma de vector ordenado por columnas, podemos obtener varios parámetros que definen la eficacia de la red. También hay métodos que directamente permiten obtener varios tipos de error.

Una vez calculados los errores y parámetros que deseamos para comprobar el funcionamiento de la red, se guardan en ficheros .csv o .xml el *dataset*, *neural network*, *training strategy* y la *matriz de confusión*.

```
analizarResultados:
    obtener la matriz de confusión de la red una vez entrenada
    obtener el error de la red
    calcular la precisión
    Para cada clase:
        calcular la exactitud
        calcular la sensibilidad
    fin Para
    calcular el tiempo de ejecución

    mostrar los resultados por pantalla

    guardar los resultados

fin analizarResultados
```

Resultados del entrenamiento y validación

Todos los resultados aquí: [Práctica 2 - Redes Neuronales](#)

El propósito de esta práctica es desarrollar distintas redes neuronales, variando los parámetros de entrenamiento (*training strategy* y *loss index*), la capa de escalado, el número de capas ocultas y el número de neuronas en cada una de ellas, etc.

Saber cuál es la mejor arquitectura y estrategia de entrenamiento para una aplicación dada es un problema complejo que requiere mucha investigación, ya que por ahora hay una serie de pautas generales y un gran trabajo de prueba-error. Para cada aplicación es necesario desarrollar diferentes escenarios y probarlos de forma empírica, para finalmente escoger el que mejor se comporte.

Escoger el modelo que mejor se comporta no es necesariamente el que consiga más precisión. Como vamos a analizar en este apartado, una alta precisión es debida a que reconoce muy bien una clase, pero la sensibilidad respecto a las otras es demasiado baja y no es viable utilizar esa red.

El número de pruebas y el tamaño de las redes está limitado en primer lugar por la memoria disponible en el computador en el que se ejecuta el código, ya que entrenar una red requiere de una memoria potente que sea capaz de efectuar todos los cálculos necesarios. Además, hay un factor temporal, ya que debido a la gran cantidad de posibles combinaciones no es realista poder probarlos todos variando además las arquitecturas.

Debido a estos factores, es entendible que no se obtengan resultados muy prometedores en el test. Aún así, es posible distinguir qué combinaciones son mejores que otras y en qué medida afecta la variación de ciertos parámetros de configuración de la red.

Las redes entrenadas se dividen en dos grandes grupos, en función de la estrategia de entrenamiento utilizada en cada una de ellas. Dentro de una misma estrategia de entrenamiento se entrena con diferentes arquitecturas y *loss index*.

Quasi-Newton Method

Este método aproxima la inversa de la *Hessiana* del índice de pérdida, que se utiliza para calcular la dirección en la que se actualizan los pesos de la red, utilizando información del gradiente, llegando a una precisión de segundo orden de derivada.

En los experimentos podemos comprobar que normalmente es un proceso relativamente costoso, ya que de media este método tarda unos 17 minutos en terminar de entrenar. En general, no se puede establecer una relación lineal entre los parámetros como la precisión y el tiempo de ejecución. A pesar de que el tiempo se mantiene estable, la precisión está influida principalmente por la arquitectura en sí. En este caso, manejamos arquitecturas

pequeñas debidas a las limitaciones de memoria del ordenador. Aún así, podemos comprobar los diferentes resultados en función de los distintos tipos de error.

Normalized Squared Error

Utilizando este tipo de error con regulación para configurar el índice de pérdida, podemos observar variaciones entre el tiempo de ejecución y la precisión obtenida que no tienen relación lineal. Sí podría concluirse que más capas van a reducir ligeramente el tiempo de entrenamiento respecto a una red con el mismo número de neuronas en una capa.

En este caso el mejor resultado obtenido es con arquitectura de dos capas ocultas, de tres y dos neuronas respectivamente, que llega a conseguir un **53'33%** de acierto. Sin embargo, no podemos quedarnos únicamente con el valor de la precisión para determinar la eficiencia de la red. En este caso, la exactitud de la misma para todas las clases está bien repartida, pero la sensibilidad a los hombres es muy baja. Esto significa que a la red le cuesta reconocer a los hombres por encima de las otras dos clases.

También hay otras arquitecturas que no reconocen bien otras clases. Por ejemplo, la 5-5 tiene un 0 en exactitud y sensibilidad para mujeres.

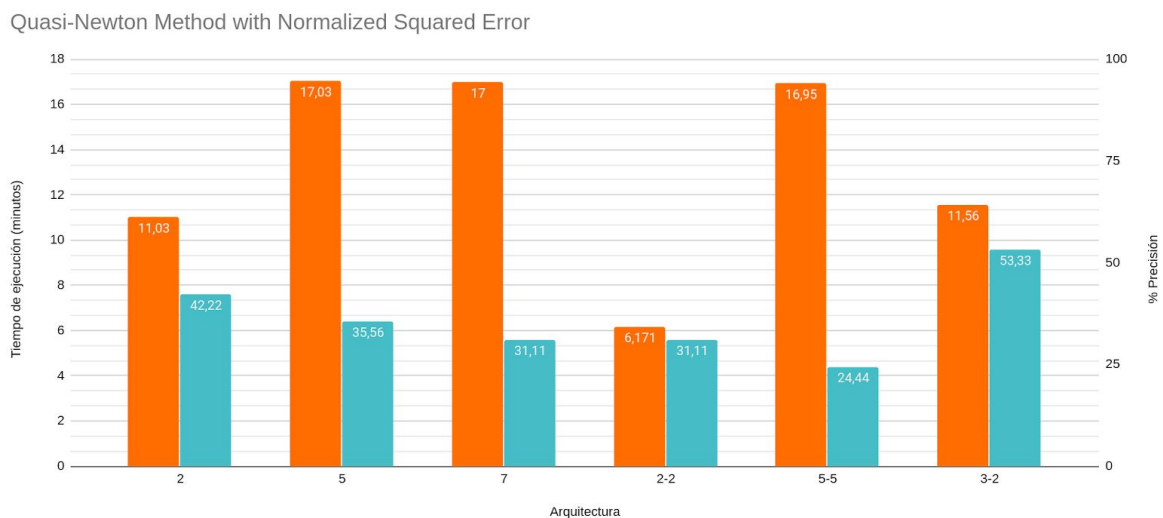


Figura 17. Gráfico de barras de los resultados obtenidos con Quasi-Newton y NSE

Minkowski Error

Utilizando esta función de error para el índice de pérdida lo primero que podemos percibir respecto a los resultados anteriores es que en este caso el tiempo es muy uniforme, independientemente de la arquitectura utilizada. También se puede comprobar que los valores de los distintos errores que obtenemos no tienen relación directa con la precisión. Hay redes que consiguen menor precisión pero con errores más bajos, y redes que aunque a pesar de tener mayor precisión funcionan peor porque tienen peor repartidas la sensibilidad y la exactitud de las clases, y por tanto, tienen más error.

Es por este motivo que elegimos la red que mejor se comporta en este caso la que tiene arquitectura 5-3, que produce un acierto del **42'22%**, a pesar de que hay otra red que tiene precisión de 44'44%. El problema de la red que tiene el mayor porcentaje de acierto es que no reconoce a las mujeres, ya que la sensibilidad y exactitud para esta clase es de 0. No podemos decir que una red es buena, por mucha precisión que pueda conseguir, si es incapaz de reconocer a toda una clase.

También se prueba la arquitectura de una capa de dos neuronas, la que consigue el 44'44% de acierto, pero añadiendo una capa de escalado de *Mean Standard Deviation*. Esta capa de escalado empeora considerablemente los resultados, ya que aparte de disminuir significativamente la precisión, la red deja de reconocer mujeres y sin persona y lo clasifica todo como hombres.

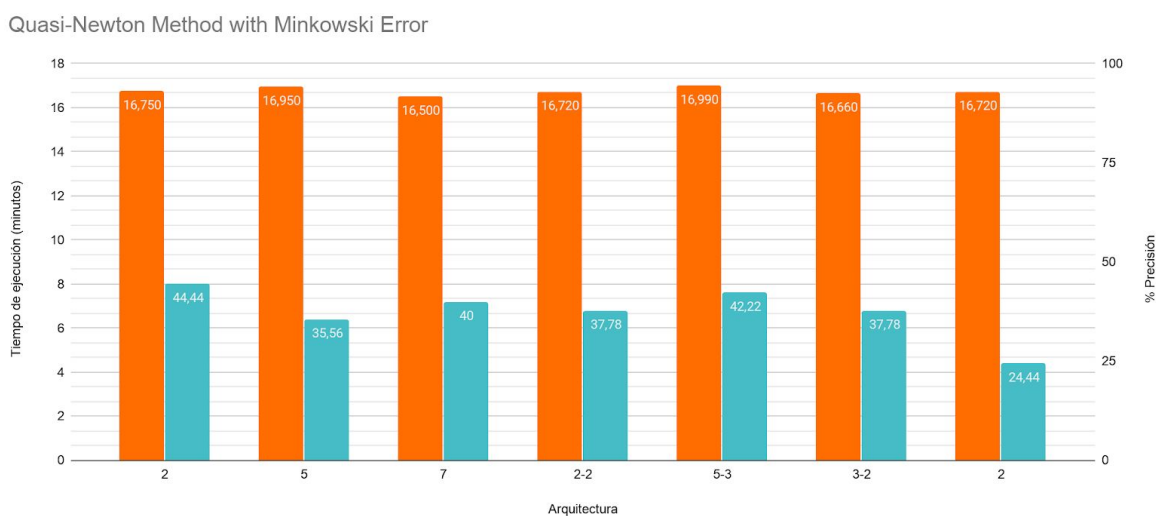


Figura 18. Gráfico de barras de los resultados obtenidos con Quasi-Newton y ME

Mean Squared Error

En este caso, utilizando el MSE, podemos comprobar que los resultados obtenidos en precisión son más dependientes de la arquitectura de la red. Obtenemos el mejor resultado para una arquitectura sencilla de una capa oculta con dos neuronas, que produce un **60%** de aciertos.

Al igual que en el caso anterior, los resultados del tiempo de entrenamiento son muy uniformes. La diferencia respecto a los dos tipos de error anteriores es que en este caso a mayor número de neuronas en la red, menor precisión. Es por eso que obtenemos el máximo de la precisión con una sola capa oculta de dos neuronas, que es el caso más básico.

Se introduce en las pruebas una capa de escalado para las entradas en la arquitectura de una capa con dos neuronas, para comprobar si se puede mejorar el resultado del 60% obtenido. El resultado, al igual que con el escalado anterior, es peor, ya que baja la precisión y la red sólo distingue a las mujeres, no reconoce ni la clase hombres ni la clase sin persona.

Quasi-Newton Method with Mean Squared Error

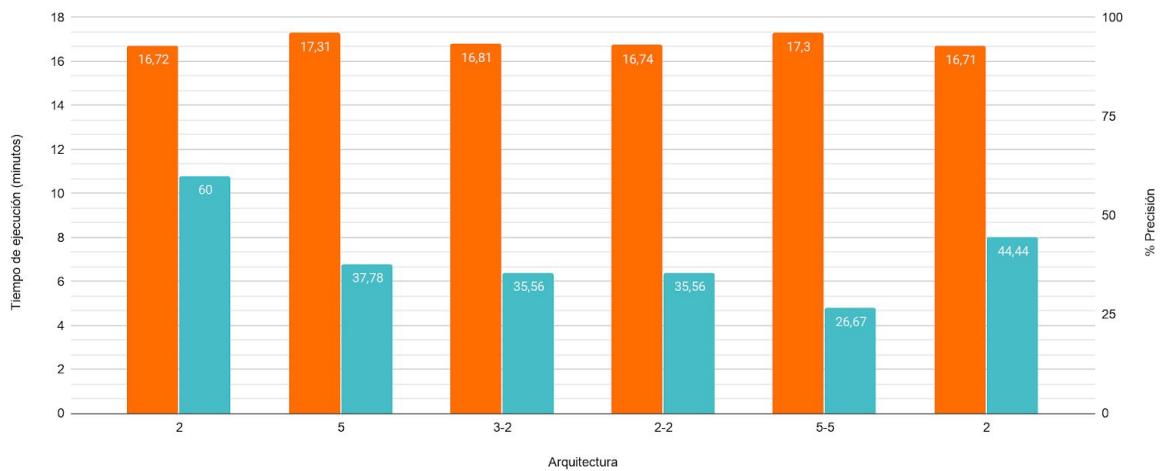


Figura 19. Gráfico de barras de los resultados obtenidos con Quasi-Newton y MSE

Weighted Squared Error

Los resultados que obtenemos si utilizamos WSE son bastante similares en tiempo al del resto de errores, pero en este caso obtenemos los mayores resultados con redes neuronales de dos capas. De las dos arquitecturas con mejores resultados, la mejor es la que tiene dos capas ocultas con tres y dos neuronas respectivamente, alcanzando un **53'33%** de acierto. La otra red que consigue la misma precisión se descarta porque no tiene ni sensibilidad ni exactitud para la clase hombre, por lo que no se puede considerar que una red funciona bien si hay una clase entera que no es capaz de reconocer, independientemente de la precisión que alcance.

Quasi-Newton Method with Weighted Squared Error

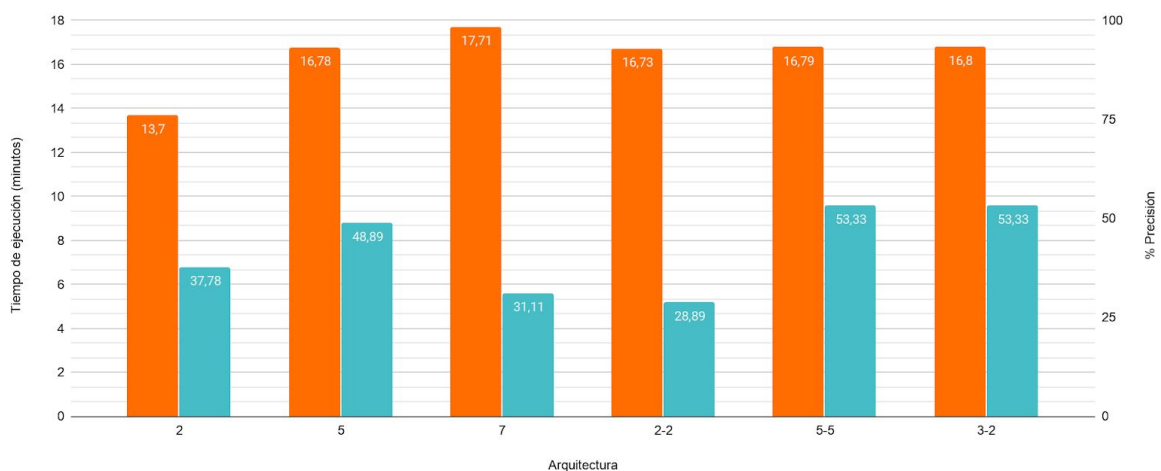


Figura 20. Gráfico de barras de los resultados obtenidos con Quasi-Newton y WSE

Tabla de resultados

Training strategy	Loss index	Capa de escalado	Arquitectura	Tiempo de ejecución (minutos)	% Precisión	Precisión	Exactitud			Sensibilidad		
							Hombres	Mujeres	Sin persona	Hombres	Mujeres	Sin persona
Quasi Newton	Normalized Squared Error	No	2	11,03	42,22	0,4222	0,2222	0,5556	0,5556	0,3636	0,25	0,7143
			5	17,03	35,56	0,3556	0,2273	1	0,4286	0,4545	0,1	0,6429
			7	17	31,11	0,3111	0,1667	0,5	0,4706	0,3636	0,1	0,5714
			2-2	6,171	31,11	0,3111	0,1786	0,6667	0,5	0,4545	0,1	0,5
			5-5	16,95	24,44	0,2444	0,1613	0	0,4615	0,4545	0	0,4286
			3-2	11,56	53,33	0,5333	1	0,5909	0,4545	0,0909	0,65	0,7143
	Minkowski Error	No	2	16,750	44,44	0,4444	0,3500	nan	0,5200	0,6364	0,00	0,9286
			5	16,95	35,56	0,3556	0,2857	nan	0,3684	0,1818	0	1
			7	16,5	40	0,4000	0,2500	0,4545	0,3684	0,09091	0,5	0,5
			2-2	16,72	37,78	0,3778	0,3182	nan	0,4348	0,6364	0	0,7143
			5-3	16,99	42,22	0,4222	0,3571	0,5	0,4286	0,4545	0,25	0,6429
			3-2	16,66	37,78	0,3778	0,2308	1	0,5556	0,5455	0,05	0,7143
		Mean St. Dev.	2	16,72	24,44	0,2444	0,2444	nan	nan	1	0	0
	Mean Squared Error	No	2	16,72	60	0,6	0,6667	0,6522	0,5263	0,1818	0,75	0,7143
			5	17,31	37,78	0,3778	0,25	1	0,4348	0,4545	0,1	0,7143
			3-2	16,81	35,56	0,3556	0,2609	nan	0,4545	0,5455	0	0,7143
			2-2	16,74	35,56	0,3556	0,2083	0,6	0,5	0,4545	0,15	0,5714
			5-5	17,3	26,67	0,2667	0,1724	0,5	0,4286	0,4545	0,05	0,4286
		Min - Max	2	16,71	44,44	0,4444	nan	0,4444	nan	0	1	0
	Weighted Squared Error	No	2	13,7	37,78	0,3778	0,2632	nan	0,4615	0,4545	0	0,8571
			5	16,78	48,89	0,4889	0	0,5714	0,4348	0	0,6	0,7143
			7	17,71	31,11	0,3111	0,1667	0,5	0,407	0,3636	0,1	0,5714
			2-2	16,73	28,89	0,2889	0,1935	nan	0,5	0,5455	0	0,5
			5-5	16,79	53,33	0,5333	0	0,6296	0,5	0	0,85	0,5
			3-2	16,8	53,33	0,5333	1	0,5909	0,4545	0,09091	0,65	0,7143

Tabla 1. Resultados experimentales utilizando Quasi-Newton Method

Gradient Descent

Este método es el algoritmo de entrenamiento más sencillo, por lo que se supone que el tiempo de entrenamiento va a ser significativamente inferior a los obtenidos utilizando Quasi-Newton. Consiste en obtener los nuevos parámetros de la red siguiendo la dirección negativa del gradiente de la función del índice de pérdida.

En este caso, tampoco podemos establecer una relación de dependencia lineal entre el tiempo de entrenamiento y la precisión obtenida. Además, se obtienen resultados mucho

menos uniformes que utilizando Quasi-Newton. Para poder comparar de forma coherente los resultados, vamos a utilizar las mismas arquitecturas que con Quasi-Newton.

Normalized Squared Error

Lo más llamativo de estos resultados respecto a los anteriores es la evidente dispersión de los tiempos de ejecución, que van a depender directamente de cada arquitectura concreta. Tenemos arquitecturas sencillas que obtienen buenos resultados en muy poco tiempo, como la que tiene una capa oculta con dos neuronas, que consigue un 51'11% de acierto.

La arquitectura que más acierto consigue es la que tiene dos capas ocultas, de cinco y tres neuronas respectivamente, que alcanza un porcentaje de precisión del **53'33%**. Además, la exactitud y la sensibilidad de las clases están debidamente equilibradas, por lo que más o menos diferencia las clases con la misma facilidad.

En este caso, repetimos dos arquitecturas para comprobar la eficacia de una capa de escalado de *minimum maximum*. Utilizamos una buena arquitectura para comprobar si los resultados mejoran más, y una arquitectura con peores resultados para comprobar si los resultados obtenidos son ligeramente mejores. En ninguno de los dos casos conseguimos mejorar la precisión, al contrario, empeora. Al igual que con los escalados de capas de ejecuciones anteriores, deja de reconocer dos clases para únicamente predecir una sola clase, por lo que este comportamiento es intolerable para una buena red neuronal.

Gradient Descent with Normalized Squared Error

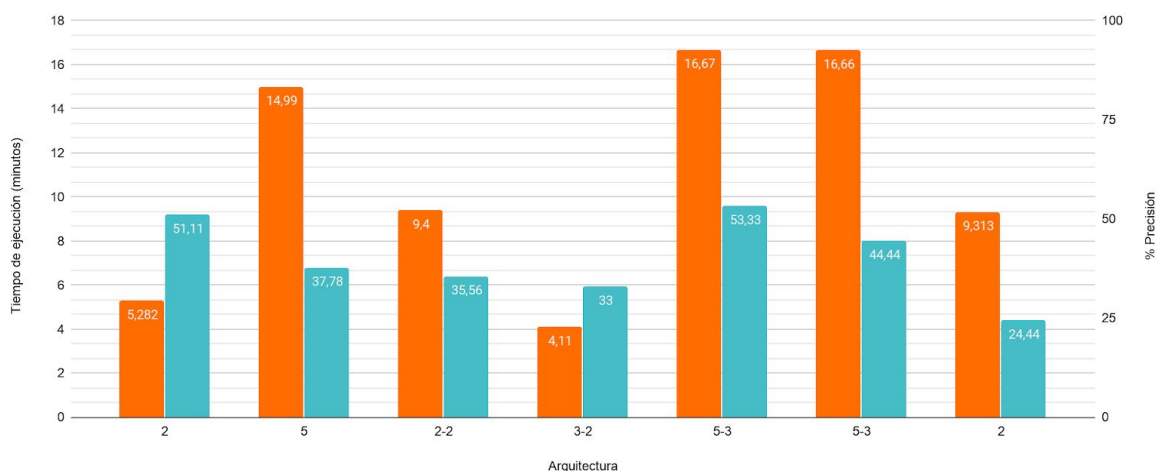


Figura 21. Gráfico de barras de los resultados obtenidos con Gradient Descent y NSE

Minkowski Error

Lo más llamativo de utilizar este tipo de error es que tenemos muchísima disparidad en los tiempos de ejecución, sin poder establecer relaciones entre este tiempo y el número total de neuronas. Por ejemplo, una arquitectura con un número de neuronas similar similar obtiene dos tiempos de ejecución totalmente dispares. Esto se puede deber a la inicialización aleatoria de la red y a la arquitectura de la misma.

Podemos comprobar también que los resultados obtenidos no son muy prometedores, ya que la mejor red obtiene un **42'22%** de precisión, y además muchas de las redes entrenadas no reconocen bien alguna de las clases porque para esa clase obtienen sensibilidad 0.

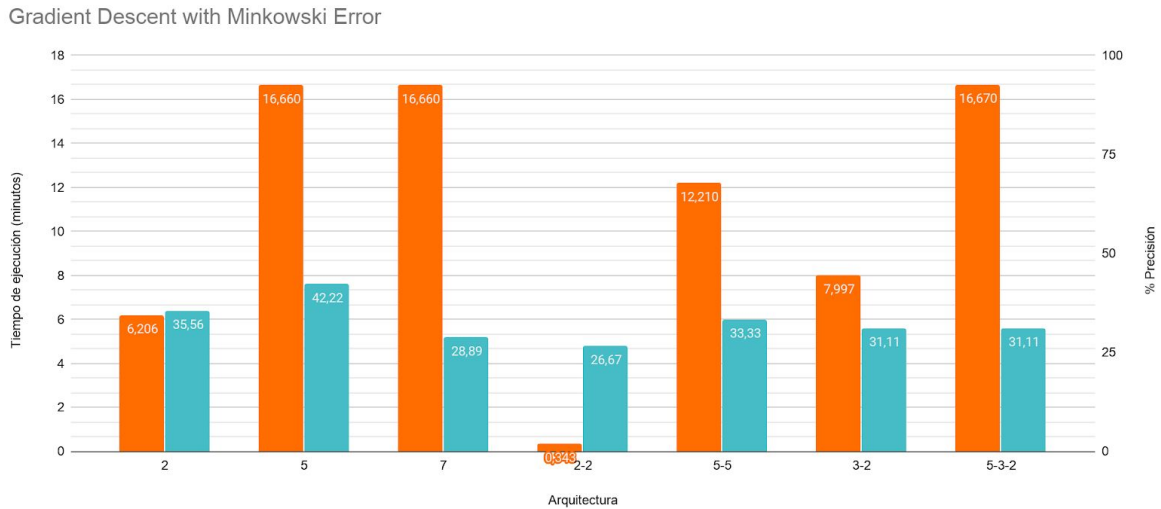


Figura 22. Gráfico de barras de los resultados obtenidos con Gradient Descent y ME

Mean Squared Error

Estos resultados son otro claro ejemplo de la dificultad que radica en intentar establecer relaciones lineales entre tiempo de ejecución, precisión obtenida y número de neuronas totales. Podemos comprobar que la distribución de las neuronas en varias capas ocultas en este caso no es muy relevante, al contrario que en ejemplos anteriores, ya que un mismo número de neuronas obtiene un resultado similar independientemente de las capas en las que se distribuyan, y el tiempo de entrenamiento también se mantiene.

Los resultados obtenidos no son muy eficaces, ya que la máxima precisión obtenida es del **40%**, con una arquitectura de cinco neuronas en una sola capa.

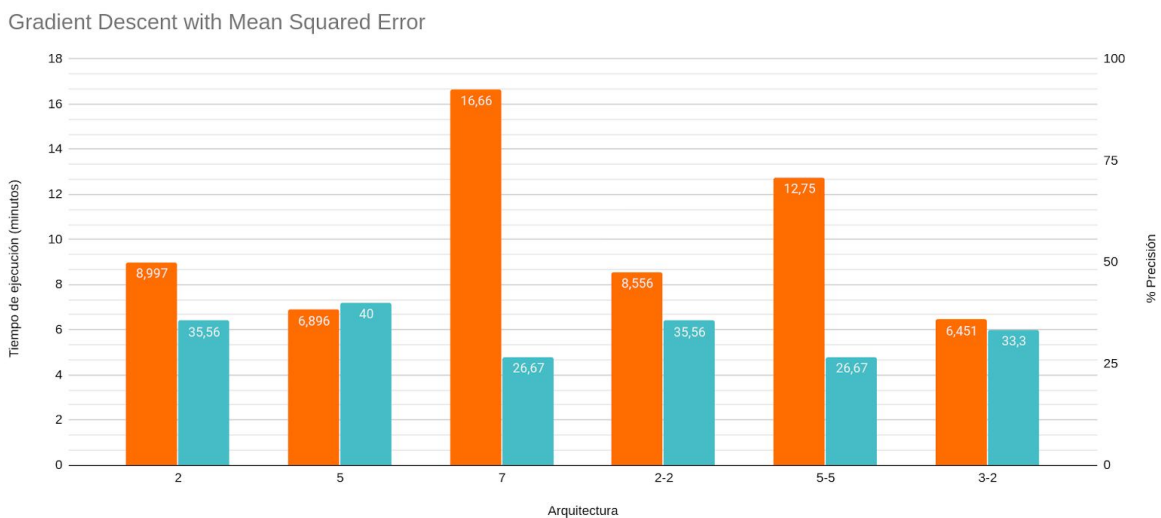


Figura 23. Gráfico de barras de los resultados obtenidos con Gradient Descent y MSE

Weighted Squared Error

Por último, comprobamos que con el WSE tampoco obtenemos buenos resultados. De hecho, es esta combinación la que peores resultados arroja, siendo el mejor resultado obtenido de un **37'78%** de precisión, con una arquitectura de 5 neuronas y una capa.

El tiempo de ejecución parece ser que depende del número total de neuronas y de la distribución de las mismas, ya que obtenemos menores tiempo cuantas menos neuronas y mayor número de capas ocultas.

A pesar de que los resultados obtenidos en precisión son bastante bajos, estas redes más o menos distinguen todas las clases, es decir, no hay ninguna arquitectura que tenga la exactitud o la sensibilidad de una clase a cero. Por lo general, las clases están bien distribuidas, aunque en algunos casos tenemos una sensibilidad bastante baja para la clase de mujeres.

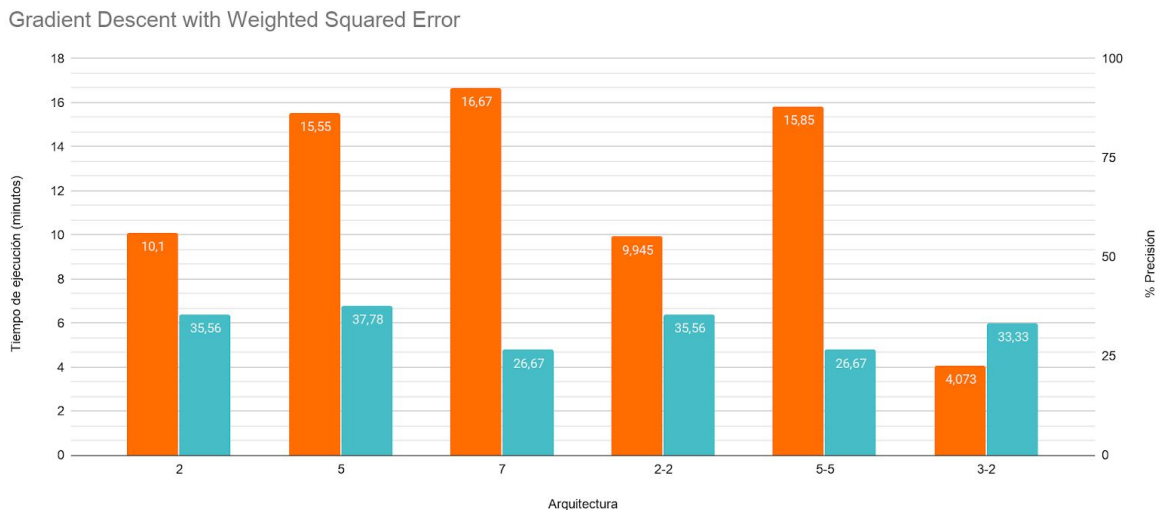


Figura 24. Gráfico de barras de los resultados obtenidos con Gradient Descent y WSE

Tabla de resultados

Training strategy	Loss index	Capa de escalado	Arquitectura	Tiempo de ejecución (minutos)	% Precisión	Precisión	Exactitud			Sensibilidad		
							Hombres	Mujeres	Sin persona	Hombres	Mujeres	Sin persona
Gradient Descent	Normalized Squared Error	No	2	5,282	51,11	0,5111	0,25	0,5909	0,5333	0,1818	0,65	0,5714
			5	14,99	37,78	0,3778	0,25	1	0,4091	0,4545	0,15	0,6429
			2-2	9,4	35,56	0,3556	0,1923	0,6667	0,5625	0,4545	0,1	0,6429
			3-2	4,11	33	0,33	0,2	0,5	0,5	0,4545	0,05	0,6429
			5-3	16,67	53,33	0,5333	0,3333	0,5833	0,5	0,0909	0,7	0,6429
		Min - Max	5-3	16,66	44,44	0,4444	nan	0,4444	nan	0	1	0
			2	9,313	24,44	0,2444	0,2444	nan	nan	1	0	0

	Minkowski Error	No	2	6,206	35,56	0,3556	0,2581	nan	0,5714	0,7273	0,00	0,5714
			5	16,66	42,22	0,4222	0	0,5263	0,375	0	0,5	0,6429
			7	16,66	28,89	0,2889	0,1923	0	0,4444	0,4545	0	0,5714
			2-2	0,3425	26,67	0,2667	0,1786	nan	0,4118	0,4545	0	0,5
			5-5	12,21	33,33	0,3333	0,2069	1	0,5	0,5455	0,1	0,5
			3-2	7,997	31,11	0,3111	0,1852	nan	0,5	0,4545	0	0,6429
			5-3-2	16,67	31,11	0,3111	0,15	0,4286	0,4444	0,2727	0,15	0,5714
	Mean Squared Error	No	2	8,997	35,56	0,3556	0,25	0,5	0,5385	0,6364	0,1	0,5
			5	6,896	40	0,4	0,25	1	0,4286	0,4545	0,2	0,6429
			7	16,66	26,67	0,2667	0,1739	0,25	0,3889	3,636	0,05	0,5
			2-2	8,556	35,56	0,3556	0,1923	0,6667	0,5625	0,4545	0,1	0,6429
			5-5	12,75	26,67	0,2667	0,1875	1	0,4167	0,5455	0,05	0,3571
			3-2	6,451	33,3	0,333	0,2	0,5	0,5	0,4545	0,05	0,6429
	Weighted Squared Error	No	2	10,1	35,56	0,3556	0,25	0,5	0,5385	0,6364	0,1	0,5
			5	15,55	37,78	0,3778	0,25	1	4,091	0,4545	0,15	0,6429
			7	16,67	26,67	0,2667	0,1667	0,25	0,4118	0,3636	0,05	0,5
			2-2	9,945	35,56	0,3556	0,1923	0,6667	0,5625	0,4545	0,1	0,6429
			5-5	15,85	26,67	0,2667	0,1875	1	0,4167	0,5455	0,05	0,3571
			3-2	4,073	33,33	0,3333	0,2	0,5	0,5	0,4545	0,05	0,6429

Tabla 2. Resultados experimentales utilizando Gradient Descent

Detección de caras

Una vez entrenada la red hay que desarrollar una aplicación en la que se pueda aplicar. Para ello, simplemente creamos una pequeña aplicación donde puedes cargar una foto y la red detecta si las caras de esta foto son de hombre o de mujer, o si no hay persona.

Detectar las caras en una foto cualquiera es una tarea bastante complicada que no corresponde con el *machine learning*. Para realizar una aproximación de la detección de la cara el programa divide la imagen en 3x3, es decir, en nueve cuadrados. Cada uno de esos cuadrados se pasa por la red, por lo que en una misma imagen se predice una clase una vez para cada ventana. Para poder hacer esto es necesario redimensionar las imágenes y tener en cuenta que el número de píxeles de la ventana que vamos a pasarle a la red es de 2500, que es el número de entradas que va a tener la arquitectura.

Código

Para poder implementar esta aplicación es muy importante poder trabajar con la clase **QImage** que ofrece Qt, que implementa todos los métodos necesarios para redimensionar y trabajar con imágenes. Además, también sirve de gran ayuda trabajar con la clases **QPixmap** para poder luego mostrar la imagen en la ventana de diálogo de ejecución del programa, y **QPainter**, que permite escribir texto y dibujar figuras geométricas encima de la imagen, para poder mostrar de forma gráfica los resultados obtenidos de la red.

Podemos definir la estructura del programa como:

```
detectarGenero:
    cargar la red neuronal entrenada y crear el objeto de clase asociado
    cargar la imagen y escalarla a 250x250

    dividir la imagen en sectores (5x5)

    Para cada sector:
        redimensionarlo
        definir el número de entradas y salidas de la red

        Para cada pixel
            obtener el color rgb
            obtener la información de un canal de un espacio de color
            escribir la información en el vector de entradas
        fin Para

        calcular las predicciones de la red
        dibujar los resultados en la imagen global
    fin Para

    mostrar y guardar la imagen

fin detectarGenero
```

Resultados

Para probar los resultados obtenidos vamos a utilizar la red que mejor comportamiento presenta, que es la red que tiene estrategia de entrenamiento **Quasi-Newton Method** con índice de pérdida de **Mean Squared Error** y una arquitectura de **una capa oculta con dos neuronas**.

A pesar de que aparentemente esta red da buenos resultados en el test, cuando la probamos con otro tipo de fotos diferente a las que se han utilizado para el dataset de entrenamiento, obtenemos unos resultados bastante malos. El principal problema de estos resultados es que la red no es sensible a los hombres y las mujeres, es decir, que le cuesta reconocerlos. Por lo tanto, la mayor parte de predicciones de la red se corresponden a la clase sin persona.

Como se ha comentado antes, esto es un claro ejemplo de por qué la precisión solo no es una buena medida del funcionamiento de la red. Podemos obtener una alta precisión porque tenemos una gran cantidad de aciertos para una clase en cuestión, pero si la red no es capaz de detectar correctamente el resto de clases el resultado no funciona.

Si analizamos bien los resultados del test, podemos comprobar que la red normalmente no se equivoca diciendo que hay una persona en un sector en el que no hay ninguna cara de persona, porque la red ha entrenado muy bien para detectar la clase sin persona. El

problema viene porque la red no predice las caras de las personas cuando hay caras de personas.

Este problema tiene origen en varios puntos. En primer lugar, puede ser porque este tipo de redes necesitan un dataset más amplio que contenga más imágenes de mujeres y de hombres para que pueda aprender mejor esas clases. También puede deberse a que las caras de las personas no coinciden con la separación de los sectores, por lo que la red no puede distinguir si es un hombre o una mujer con una cara a medias, y simplemente predice que no hay persona.

Otro aspecto que podemos discernir es que la red no puede únicamente probarse con imágenes que pertenecen al dataset para comprobar su eficacia, ya que los resultados reales distan mucho de los obtenidos con el dataset de test.

Por ejemplo, en esta imagen las caras no están completas, por lo que no lleva a cabo una detección de las caras. Sin embargo, como los hombres normalmente llevan camisas en el dataset de entrenamiento, detecta la camisa del fondo como si fuera un hombre.



Figura 25. Imagen 1 de test

En otra imagen, por ejemplo, detecta como hombre dos sectores de la imagen que tienen pelo corto y una barba, características del aspecto físico de un hombre según las imágenes del dataset. También detecta un rostro humano que asocia directamente a la clase hombre.

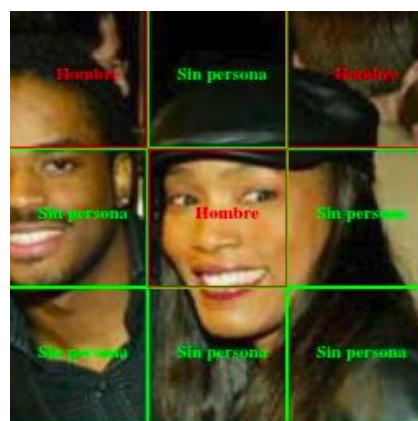
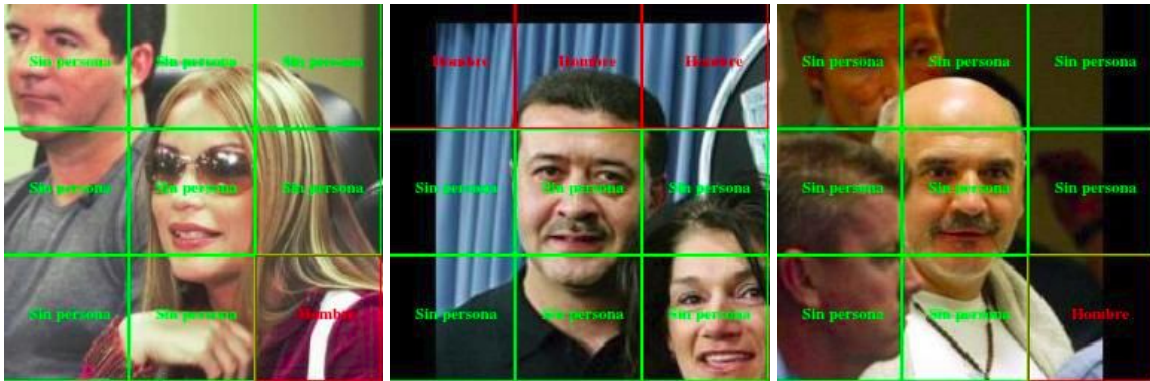


Figura 25. Imagen 5 de test

En otros casos, a pesar de que las caras de la imagen coinciden bastante bien en el sector, la red no reconoce al hombre o mujer y la predicción es que no hay persona, como en estos ejemplos. En concreto, en la imagen 11 del test reconoce como hombre el atributo de pelo corto, y falla en los dos laterales que detecta hombre cuando no hay persona. En las otras dos fotos también hay predicciones erróneas de hombre cuando no hay persona.



Figuras 26-28. Imágenes 6, 11 y 17 del test

Si no fijamos en otro ejemplo, podemos pensar que lo que está aprendiendo la red es el vestuario asociado a cada clase según las imágenes del dataset, por lo que no está clasificando en función de las caras sino en función de la ropa. En este ejemplo detecta que la corbata y la camisa es un hombre, en vez de detectar claramente la cara.

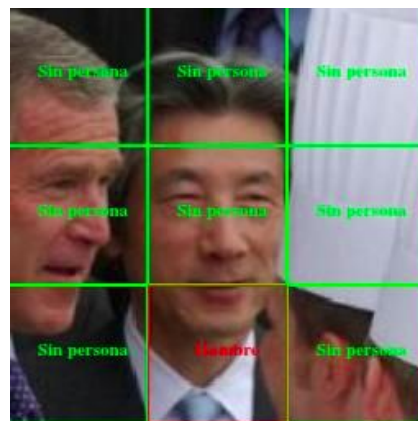


Figura 29. Imagen 16 del test

Tal y como se ha podido comprobar, los resultados obtenidos son erróneos en su totalidad, porque a pesar de que la detección de las no personas funciona relativamente bien, la red no detecta las caras y mucho menos es capaz de asignarles un género, por lo que la finalidad principal de la aplicación no se lleva a cabo y es totalmente inútil.

Conclusiones

Los resultados obtenidos no son ninguna sorpresa. Este tipo de redes neuronales, pertenecientes al llamado *machine learning*, se utilizan desde el siglo pasado. Durante muchos años los investigadores trataron de llevar a cabo correctamente tareas de detección de género y reconocimiento de personas, dedicando mucho esfuerzo, tiempo y recursos, sin llegar a obtener grandes resultados.

La forma de mejorar los resultados llegó en la década de 2010, cuando se inventaron las **redes neuronales convolucionales**. Estas redes neuronales están dentro de la disciplina del *deep learning*, que utiliza redes neuronales más complejas que las redes de clasificación formadas por perceptrones. En este tipo de redes neuronales, se ha conseguido resolver el problema de la detección de género utilizando inteligencia artificial. Estas redes ya no codifican las imágenes utilizando números, sino que directamente aprenden con las imágenes en su totalidad. Es un campo muy interesante que está en desarrollo actualmente y que ofrece una gran cantidad de posibilidades para el futuro.

Sin embargo, durante la realización de esta práctica *a la vieja usanza*, una de las lecciones que mejor se ha aprendido es lo difícil que es encontrar la arquitectura adecuada para un único problema. Los expertos en *machine learning* han tenido que dedicar mucho tiempo a realizar pruebas que se pueden considerar tediosas para únicamente conseguir la estructura que mejor funciona en relación a un único problema. Por eso es importante conocer bien toda la teoría de clasificadores, para que a la hora de implementar una red sea posible seguir unas pautas que puedan ahorrar tiempo en la resolución del problema.

Bibliografía

- [1] [QImage Class | Qt GUI 5.15.0](#)
- [2] [Neural Networks Tutorial](#)
- [3] [4. El perceptrón](#)
- [4] [Tema 8. Redes Neuronales](#)
- [5] [Neural networks tutorial: Neural network](#)
- [6] [NeuralNetwork Class Reference](#)
- [7] [The training strategy class](#)
- [8] [Neural networks tutorial: Training strategy](#)
- [9] [TestingAnalysis Class Reference](#)
- [10] [LossIndex Class Reference](#)
- [11] [Machine Learning a tu alcance: La matriz de confusión](#)
- [12] [Confusion Matrix](#)
- [13] [Neural networks tutorial: Testing analysis](#)