Introducción a las redes neuronales artificiales

UNIVERSIDAD COMPLUTENSE DE MADRID

GRADO EN INGENIERÍA MATEMÁTICA

Trabajo fin de grado

Juan Miguel Sierra Ramos
Tutor: Antonio López Montes

Facultad de Matemáticas
Departamento de Análisis Matemático y Matemática Aplicada

Curso: 2021/2022
27 de febrero de 2022

Resumen

En la actualidad las redes neuronales son una de las herramientas con mayor número y variedad de aplicaciones a todos los campos de la ciencia , la ingeniería, la medicina, la arquitectura.... A diario aparecen en los medios de comunicación noticias que relacionan a las redes neuronales con aplicaciones novedosas, sorprendentes y muy prometedoras.

El ánimo que nos mueve en la realización de este trabajo es profundizar en el conocimiento y las aplicaciones de las redes neuronales.

La estructura del trabajo es la siguiente: en este primer apartado hablamos sobre la historia de las redes neuronales artificiales, describimos los componentes básicos de las redes neuronales, las neuronas artificiales, y de donde surgen. Así como algunos conceptos básicos que nos serán útiles en el desarrollo de los modelos. En el segundo apartado presentamos los primeros modelos de redes neuronales artificiales junto a sus algoritmos de aprendizaje y algunos ejemplos. En el tercer y último apartado, desarrollamos algunos ejemplos de redes neuronales artificiales multicapa y tratamos las redes neuronales convolucionales.

Palabras clave

Adaline, Backpropagation, Hopfield, Perceptrón, Red Neuronal artificial, Red neuronal convolucional.

Abstract

Nowadays, artificial neural networks are a tool with lot of applications in many fields of the science such as engineering, medicine, architecture ....Daily we see news that link artificial neural networks to astonising results in various fields.

The objetive of this proyect is making a reference where you can dive into the knowledge and the applications of artificial neural networks.

The proyect is structured as follow: on the first section we talk about the history of artificial neural networks, their basic components and how all of them emerged. The second section presents the firsts models and their learning process with a few examples. On the third and last section we develop more examples and we talk about the convolutional neural networks.

Keywords

Adaline, Artificial Neural Network, Backpropagation, Convolutional Neural Network, Hopfield, Perceptron.

Índice

Índice de figuras

1. Fundamentos

1.1. Un poco de historia

El primer modelo de una red neuronal artificial surge en 1943, de la mano de Warren McCulloch y Walter Harry Pitts. Pretende simular el funcionamiento de una neurona en el cerebro humano. Es un modelo básico y que no se llegó a implementar físicamente debido a las limitaciones técnicas de la época.

Hebb [6] escribe "The Organization of Behavior" libro que establece una conexión entre psicología y fisiología. Postula que la información se representa en el cerebro mediante un conjunto de neuronas activas o inactivas y que el aprendizaje se localiza en las conexiones entre las neuronas. Da lugar a la teoría hebbiana que sentó las bases que todavía se usan en el desarrollo de redes neuronales artificiales.

En 1958 Rosenblatt [21] elabora el modelo del perceptrón, modelo formado por una única neurona artificial que posee una salida binaria y que es capaz de variar sus pesos. El Perceptrón puede resolver problemas de clasificación cuya solución presenta una separación lineal, por ejemplo la puerta AND o la puerta OR. Sin embargo el Perceptrón es incapaz de resolver problemas que no presentan una solución lineal, como sería el ejemplo de la puerta XOR.

Dos años más tarde se desarrolla el modelo Adaptative Linear Neuron (ADALINE) de la mano de Bernard Widrow y Ted Hoff, red neuronal artificial con una estructura también sencilla pero que en lugar de presentar una salida binaria genera salidas reales y utiliza el método de la regla delta en su aprendizaje. La red ADALINE incorpora más neuronas, aunque sigue teniendo una sola capa.

Marvin and Seymour [12] publican en su libro "Perceptrons" las deficiencias de los modelos monocapa, lo que provoca una ralentización en el desarrollo tanto teórico como práctico de las redes neuronales hasta mediados de la década de los ochenta [4]. Durante este periodo surgen modelos alternativos a los establecidos como por ejemplo la red Hopfield.

Es, con la publicación del algoritmo "Backpropagation" de Rumelhart, Hinton, and Williams [22] en 1986, cuando se produce el resurgir del desarrollo de las redes neuronales artificiales. Este algoritmo presenta una nueva forma de aprendizaje para las redes neuronales, mediante la retropropagación de errores desde las últimas capas y haciendo uso del descenso del gradiente para corregir este error.

El desarrollo tecnológico de las últimas décadas, así como la aparición de grandes conjuntos de datos, como los grandes conjuntos de imágenes de uso libre que veremos, ha permitido la elaboración de nuevas y mas eficientes estructuras en las redes neuronales artificiales como son las redes neuronales convolucionales o las redes neuronales recurrentes entre otras.

Cabe destacar la aparición en los últimos años de redes neuronales pre-entrenadas, las cuales se enfocan en la realización de una tarea específica con enormes cantidades de datos y una estructura compleja. Un ejemplo de estas redes son VGG16 o VGG19 dos redes neuronales convolucionales pre-entrenadas en la tarea de clasificación de imágenes.

En los últimos años, han surgido multitud de aplicaciones de las redes neuronales artificiales con resultados prometedores. Por citar algunas de las últimas estructuras tenemos alphafold y alphafold2 aplicadas en el campo de la biomedicina y que lograron resolver el problema planteado en la competición CASP[1][1] una competición para decidir el software que predice mejor la estructura de proteínas a partir de sus secuencias de aminoácidos. TCAV un método que permite visualizar como actuan las capas intermedias u ocultas de las redes neuronales [9]. VQGAN red neuronal capaz de producir imágenes a partir de texto[2].

[1] Critical Assessment of Protein Structure Prediction

1.2. Introducción teórica

1.2.1. La neurona

La neurona es un tipo de célula diferente del resto. Forma parte del sistema nervioso, que es el responsable de todas las funciones cognitivas. Está formada por 3 partes fundamentales: el núcleo o soma , las dendritas y el axón. Las señales son recibidas a través de las dendritas , estas señales pueden provenir de un sistema sensorial externo o de otra neurona. El núcleo procesa esta información y la envía por el axón el cual posee varios terminales. Una neurona puede recibir información de miles de otras neuronas y enviar información a otros miles de ellas.

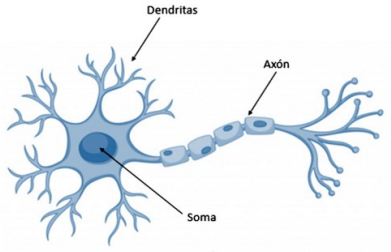
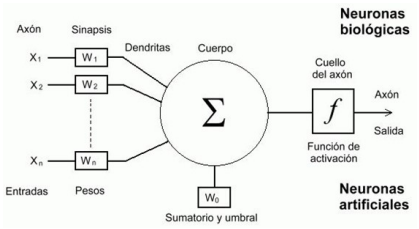


Figura 1.1: Representación de una neurona

Se estima que en el cerebro tenemos unas 10^{11} neuronas con 10^{15} conexiones, gracias a estar tan altamente conectadas es como consiguen una gran capacidad de procesamiento y realización de tareas complejas. La comunicación entre neuronas no es física sino que se hace a través de la sinapsis, un espacio ocupado por sustancias químicas denominadas neurotransmisores. Estos son los que se encargan de bloquear o dejar pasar las señales. Dichas señales se procesan en el núcleo de la neurona y bajo ciertas condiciones apropiadas(activación) se transmiten a través del axón.

1.2.2. La neurona artificial

La pieza básica sobre la que se desarrollan las redes neuronales es la neurona artificial(o simplemente neurona), la cual, puede ser entendida no como una estructura física sino como una función que trata de modelar matemáticamente el funcionamiento de una neurona del cerebro humano.



La información recibida por la neurona se representa como el vector $\vec{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$. Para modelar la sinapsis otorgamos unos pesos sinápticos w_i a cada entrada x_i para todo $i = 1, \ldots, n$. Estos dan lugar al vector de pesos $\vec{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n$. La neurona artificial realiza un producto escalar entre los vectores $\vec{x}$ y $\vec{w}$. A este producto escalar

se le añade el umbral(bias) w_0 de la neurona, que representa el valor a partir del cual se considerará que la neurona se activa. La neurona realiza una suma ponderada de las entradas por los pesos asignados a lo que suma el bias:

$$EntradaNeta =< \vec{x}, \vec{w} > +w_0 = \sum_{i=1}^{n} x_i w_i + w_0 = x_1 w_1 + \ldots + x_n w_n + w_0 \quad (1.1)$$

A esta suma se le llama también entrada neta de la neurona, la activación o no de la neurona artificial vendrá además determinada por la función de activación. El uso de esta función de activación nos permite la concatenación de neuronas, pues la sucesión de múltiples regresiones lineales sería equivalente a realizar solo una regresión lineal. La salida de la neurona vendrá determinada por la función de activación del valor de la cantidad neta 1.1:

$$y = f(EntradaNeta) = f(x_1 w_1 + \ldots + x_n w_n + w_0) \quad (1.2)$$

Podemos ver un análisis más profundo de como la neurona artificial surge del modelado de la neurona biológica en [2], [20].

La versatilidad en las redes neuronales reside en la concatenación de muchas neuronas. Esto permitirá al modelo poder tener en consideración tanto los elementos de nuestro problema como la relación entre los mismos. No obstante, como comentamos antes, la concatenación de funciones lineales da como resultado una función lineal. Tenemos entonces que aplicar una transformación no lineal, es aquí donde entran las funciones de activación.

1.3. Funciones de activación

Pasamos a describir las principales funciones de activación (f) usadas en las redes neuronales artificiales 1.2. Recordemos que la entrada de estas funciones es la suma ponderada de los valores de entrada de las neuronas por sus pesos, sumando el parámetro de Bias 1.1. Describiremos brevemente las funciones de activación más usadas e importantes.

1.3.1. Función de Salto Binaria

La primera de las funciones de activación que usaron las redes neuronales artificiales, está presente en las estructuras del Perceptrón y Hopfield que veremos más adelante. Haciendo uso del Bias o parámetro de sesgo, desplazaremos esta función según se requiera. Pueden tomarse tanto el valor 0 como 1 en la discontinuidad de la función. Es una función que anula todos los valores que estén a la izquierda de la discontinuidad y establece en 1 todos aquellos que estén a la derecha.

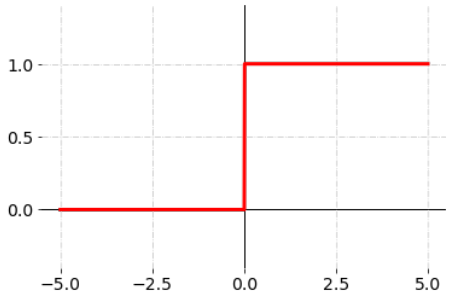


Figura 1.2: Función salto binario

1.3.2. Unidad Rectificada Uniforme (RELU)

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

Mediante esta función de activación consideramos solo los valores positivos para la salida, es la más usada pues tiene un menor coste computacional que otras funciones de activación definidas más adelante.

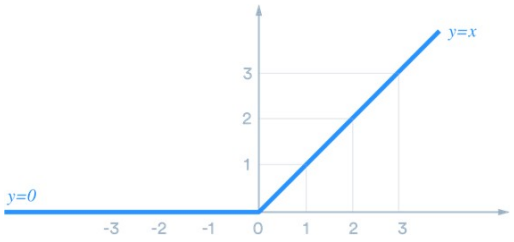


Figura 1.3: Función RELU

1.3.3. Función SoftMax

$$\alpha : \mathbb{R}^K \to [0,1]^K$$

$$\alpha(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad j = 1, \ldots, K$$

Su coste computacional es bastante más elevado que el de la RELU. Devuelve valores en el intervalo $[0, 1]$ que suman 1 en total, por lo que muchas veces se usa para asignar probabilidades en problemas categóricos.

1.3.4. Función sigmoide

$$P(t) = \frac{1}{1 + e^{-t}}$$

Otra función bastante usada al final de las redes neuronales refleja muy bien la curva de aprendizaje de cualquier red, penalizando aquellos valores cercanos a cero o a uno. Su coste computacional es bastante elevado lo que hace que para estructuras elaboradas de redes neuronales no se use dicha función de activación en las capas intermedias.

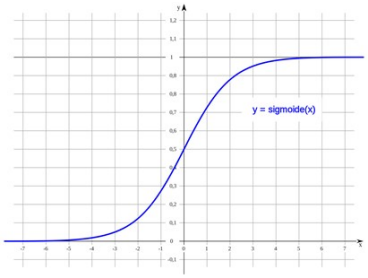


Figura 1.4: Función sigmoide

1.3.5. Tangente hiperbólica (Tanh)

$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

Actúa de manera similar a las 2 funciones de activación anteriores, devolviendo los valores en el intervalo (-1,1) en lugar de (0,1). Al igual que la sigmoide refleja muy bien la curva de aprendizaje del modelo.

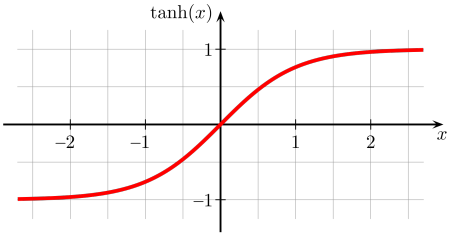


Figura 1.5: Función tangente hiperbólica

1.4. Aprendizaje supervisado y aprendizaje no supervisado

Nuestros datos se representan mediante un vector con n componentes, donde cada componente es una variable que nuestro problema contempla. Así, representamos con $\vec{x_j} = (x_1^j, \ldots, x_n^j)$ la observación j-ésima de nuestras variables $j = 1, \ldots, P$. Siendo P el número total de observaciones con las que trabajaremos.

Podemos tener una o varias variables que se desean estudiar, a las que llamaremos variables respuestas y que denotaremos por $\vec{d_j} = (d_1^j, \ldots, d_m^j)$ con $j = 1, \ldots, P$. En el aprendizaje supervisado tenemos las respuestas esperadas($\vec{d_j}$) para esas P observaciones. Lo que nos permitirá entrenar a las redes neuronales artificiales. Así formamos patrones de entrenamiento donde cada patrón j se define como el par $\{\vec{x_j}|\vec{d_j}\}$ $j = 1, \ldots, P$, es decir, cada patrón está formado por una observación y las salidas esperadas para dicha observación. Si por el contrario se presentan los datos de entrada $\vec{x_j}$ en la red estaremos trabajando con técnicas de aprendizaje no supervisado.

El uso de una red neuronal requiere del diseño de una estructura para la red. Esta puede ser elaborada o se puede emplear una estructura ya creada. Se debe decidir tanto el número de capas que posee la red como el número de neuronas de cada capa. No existe ningún criterio para los mismos, no obstante, se suele poner un número de neuronas más elebado en las primeras capas que en las posteriores [25].

1.5. Overfitting y underfitting

Cuando trabajamos con cualquier método relacionado con el tratamiento de datos para realizar una tarea de aprendizaje debemos tener cuidado a la hora de entrenar. Se pueden presentar 2 situaciones durante el entrenamiento que debemos evitar. La primera es cuando nuestro modelo durante la fase de entrenamiento no da buenos resultados, es decir, no aprende o tiene mucho sesgo [2] [19]. Si suponemos que estamos desarrollando un modelo para diferenciar fotos de perros y gatos, nuestro modelo tendría underfitting si durante el entrenamiento no llegase nunca a poder diferenciar entre ambos conjuntos de datos.

La otra situación, denominada overfitting, que se puede presentar se detecta con datos externos a los usados durante el entrenamiento. En este caso, nuestro modelo sí ha obtenido un buen desempeño en la fase de entrenamiento con las fotos proporcionadas, no obstante, al usar una foto de un perro o un gato externa al conjunto de entrenamiento nuestro modelo no diferencia de forma correcta. Esto puede ocurrir por varias razones , pero la principal es una falta de generalización en el propio conjunto de datos de entrenamiento, por ejemplo si durante la fase de entrenamiento usásemos solo fotos de razas grandes de perros, al presentar una nueva foto con razas de perros pequeñas el modelo podría equivocarse.

[2] Diferencia entre la respuesta obtenida y la esperada

2. Los primeros modelos

En este apartado vamos a introducir los modelos básicos de redes neuronales en el orden cronológico en que se desarrollaron. Describiremos sus procesos de aprendizaje y como estos mismos han ido evolucionando para solventar problemas que iban surgiendo. La combinación de estructuras complejas y entrenamientos eficientes han dado lugar a la evolución que se ha vivido en el mundo de la inteligencia artificial.

2.1. El Perceptrón

El perceptrón fue el primer modelo de una red neural artificial. Es capaz de resolver problemas con una superficie de separación lineal. Su estructura es sencilla, posee varias entradas $(x_1, x_2, \ldots, x_n)$ con pesos $(w_1, w_2, \ldots, w_n)$ y una única salida (y) con una función de activación binaria como vemos en la imagen 2.1, donde hemos supuesto que w_0 es el bias de la neurona.

$$f(w_1x_1+w_2x_2+\ldots+w_nx_n+w_0) = \begin{cases} 1 & \text{si } w_1x_1 + w_2x_2 + \ldots + w_nx_n + w_0 \geq 0 \\ 0 & \text{si } w_1x_1 + w_2x_2 + \ldots + w_nx_n + w_0 \leq 0 \end{cases}$$

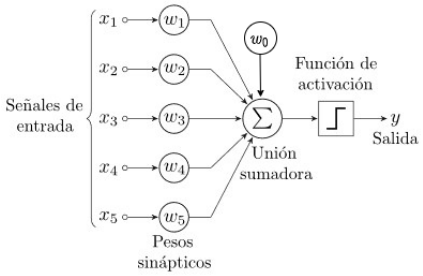


Figura 2.1: Estructura de un Perceptrón

Si suponemos un perceptrón con 2 entradas (x_1, x_2) de pesos (w_1, w_2) y sesgo w_0. La función de activación crea una separación en el plano formado por el par (x_1, x_2).

$$x_1 w_1 + x_2 w_2 + w_0 = 0$$

Que corresponde a la recta de ecuación:

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

Si ampliamos una dimensión más considerando una tercera entrada x_3, obtendremos que la función de activación separa el espacio con el plano:

$$x_3 = -\frac{w_1}{w_3} x_1 - \frac{w_2}{w_3} x_2 - \frac{w_0}{w_3}$$

En general, si tenemos n entradas la función de activación generará una división en el espacio euclídeo n-dimensional cuya frontera es un hiperplano[3].

2.1.1. Algoritmo de aprendizaje

En primer lugar, asignamos unos pesos $(w_1, w_2, \ldots, w_n)$ a las entradas de la neurona, así como elegimos un bias o sesgo w_0 [1]. Dichos valores se suelen tomar en el intervalo $[-1, 1]$ e irán cambiando a lo largo del entrenamiento en caso de ser necesario. Suponemos que tenemos un conjunto de vectores $X = \{\vec{x_1}, \vec{x_2}, \ldots, \vec{x_p}\}$ correspondientes a los P patrones de entrenamiento de salidas esperadas $D = \{\vec{d_1}, \vec{d_2}, \ldots, \vec{d_p}\}$ donde cada $\vec{d_i} \in \mathbb{R}$ al tratarse de un perceptrón con una sola salida. La salida del perceptrón viene dada por:

$$y = f(\sum_{i=1}^{n} (w_i x_i) + w_0)$$

Siendo f la función de activación definida en 2.1. Podemos pues, calcular la diferencia entre la salida deseada d_i para el vector de entrada $\vec{x_i}$ y el valor obtenido por el perceptrón con esas entradas y_i. Esto nos da el error cometido para ese patrón de entrenamiento:

$$e_j = d_j - y_j \quad para \quad j = 1, \ldots, P$$

A continuación, se actualizan los pesos de las entradas siguiendo la siguiente fórmula, tenemos en cuenta que nos encontramos con el patrón de entrenamiento $j (j = 1 \ldots, P)$ formado por el par $\{\vec{x_j} | d_j\}$ y tenemos un error cometido e_j(debido a la forma de la función de activación este error tomará los valores 0,1 o -1):

$$w_i(t+1) = w_i(t) + \alpha e_j x_i^j \quad Para \;\; i = 1, \ldots, n$$
$$w_0(t+1) = w_0(t) + \alpha e_j$$

[1] Puede verse que el sesgo sirve para que la recta que separará el plano no quede anclada en el origen

Donde $w_i(t)$ es el valor del peso i-ésimo en la fase de entrenamiento t , $w_i(t+1)$ el nuevo valor que le damos, $w_0(t)$ es el sesgo en la misma fase y α es el factor de aprendizaje. Actualizamos los pesos de las entradas y el sesgo hasta que todas las salidas del perceptrón sean iguales a las salidas deseadas o hasta que llegamos a un número determinado de iteraciones. En el último caso deberemos tratar de cambiar los parámetros iniciales, aumentar el número de iteraciones o tratar de buscar una estructura de red neuronal más elaborada.

El Perceptrón realiza su proceso de entrenamiento en base a la diferencia entre las entradas obtenidas y las deseadas, sin buscar ningún óptimo en la función de costes como algunas estructuras que veremos en secciones posteriores. Esto provoca que el algoritmo de aprendizaje pueda darnos distintos resultados, todos válidos, para diferentes configuraciones de peso iniciales. Asimismo, hace a esta red neuronal artificial vulnerable a los datos con ruido.

2.1.2. Ejemplo:Clasificación de conjuntos linealmente separables

En el siguiente ejemplo usaremos el perceptrón para separar dos conjuntos de puntos distintos presentados de la siguiente forma.

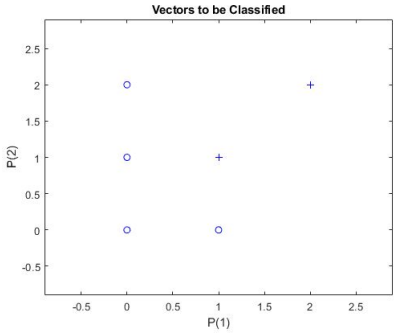


Figura 2.2: Nube de puntos a dividir

Dada una recta cuyos parámetros iniciales los establecemos de forma aleatoria, mediante el uso del perceptrón, se variarán dichos parámetros para dividir de forma correcta el plano separando ambas nubes de puntos.

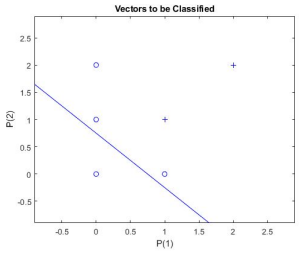


Figura 2.3: Recta inicial sin entrenar

Haciendo uso del perceptrón, se entrenarán los parámetros de la recta hasta que la misma separe de forma correcta los puntos. A diferencia de otras redes neuronales artificiales que veremos más adelante, con la salida binaria del perceptrón solo se puede determinar si una salida es correcta o incorrecta no pudiendo cuantificar cuanto difieren. El resultado final se muestra en la siguiente imagen:

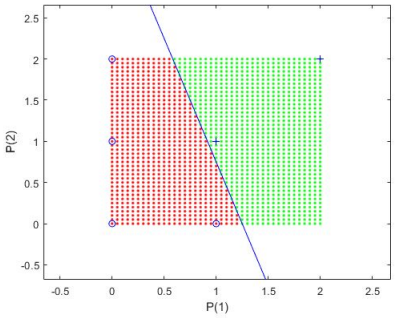


Figura 2.4: Recta entrenada

Puede encontrarse el código implementado en Matlab en 3.3.7. Tanto la formulación teórica como el ejemplo aparecen en [2].

2.2. ADALINE

La red ADALINE o Adaptative Linear Neuron es similar al perceptrón, es una red monocapa con varias entradas conectadas totalmente. Su salida es un número real, no binaria como en el caso del Perceptrón.

Mientras que en el Perceptrón se definía simplemente si la red tiene una salida errónea o correcta durante el aprendizaje, la red ADALINE toma en cuenta cuanto difiere la salida obtenida de la esperada.Tanto el desarrollo teórico de esta red como el ejemplo presentado de la misma pueden encontrarse tratados con mayor extensión en [24].

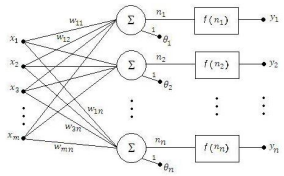


Figura 2.5: Estructura de la red ADALINE

2.2.1. Algoritmo de aprendizaje

La red ADALINE emplea en su proceso de aprendizaje un algoritmo todavía muy presente en las redes neuronales actualmente, la llamada Regla Delta que fue la precursora del algoritmo Backpropagation. Este algoritmo se basa en la idea del empleo de las derivadas parciales para, dado un punto, hacerlo avanzar hacia el mínimo de la función. Para ello definimos el error total producido en los P patrones de entrenamiento como:

$$J = \frac{1}{2}\sum_{j=1}^{P}\sum_{i=1}^{m}(d_{ij} - y_{ij})^2 = \frac{1}{2}\sum_{j=1}^{P} e_j \tag{2.1}$$

Es decir, hacemos uso del error cuadrático. Donde $\vec{d_j} = (d_{1j}, \ldots, d_{mj}) \in \mathbb{R}^m$ es el valor de salidas deseadas para el patrón de entrenamiento j e $\vec{y_j} = (y_{1j}, \ldots, y_{mj}) \in \mathbb{R}^m$ es el vector de salidas de la red en el patrón de entrenamiento. El objetivo del algoritmo es encontrar el conjunto de pesos que haga mínimo J. En cada patrón de entrenamiento $j \in \{1, \ldots, P\}$ se ajustan los pesos w_{ik} entre la entrada i-ésima y la neurona k-ésima de la siguiente forma:

$$w_{ik}(t+1) = w_{ik}(t) + \triangle w_{ik}(t) \tag{2.2}$$

$$w_{0k}(t+1) = w_{0k}(t) + \triangle w_{0k}(t) \tag{2.3}$$

Con $\triangle w_{ik}(t) = -\alpha\frac{\partial J}{\partial w_{ik}}$ $\quad con\ \ i = 0, \ldots, n(numero\ de\ entradas\ en\ la\ primera\ capa)\ \ k = 1, \ldots, m(numero\ de\ Neuronas)$

Aplicando la regla de la cadena tenemos:

$$\frac{\partial J}{\partial w_{ik}} = \frac{\partial J}{\partial e_j}\frac{\partial e_j}{\partial y_{kj}}\frac{\partial y_{kj}}{\partial w_{ik}}$$

Por 2.1 tenemos:

$$\frac{\partial J}{\partial e_j} = \frac{1}{2}$$

Como $e_j = (d_{1j} - y_{1j})^2 + \ldots + (d_{kj} - y_{kj})^2 + \ldots + (d_{mj} - y_{mj})^2$:

$$\frac{\partial e_j}{\partial y_{kj}} = -2(d_{kj} - y_{kj})$$

Por último, $y_{kj} = w_{1k}x_1 + \ldots + w_{ik}x_i + \ldots + w_{nk}x_n + w_{0k}$:

$$\frac{\partial y_{kj}}{\partial w_{ik}} = x_i \quad i = 1, \ldots, n$$
$$\frac{\partial y_{kj}}{\partial w_{0k}} = 1$$

Así:

$$\frac{\partial J}{\partial w_{ik}} = -(d_{kj} - y_{kj})x_i \ \ , \triangle w_{ik}(t) = \alpha(d_{kj} - y_{kj})x_i$$
$$i = 1, \ldots, n \ \ k = 1, \ldots, m$$
$$\frac{\partial J}{\partial w_{0k}} = -(d_{kj} - y_{kj}) \ \ , \triangle w_{0k}(t) = \alpha(d_{kj} - y_{kj})$$

Con lo que la fórmula del aprendizaje 2.2 nos queda:

$$w_{ik}(t+1) = w_{ik}(t) + \alpha(d_{kj} - y_{kj})x_i \ , \ w_{0k}(t+1) = w_{0k}(t) + \alpha(d_{kj} - y_{kj})$$

Repetimos el proceso hasta que el valor del error total J es menor que un valor determinado o hasta que lleguemos a un número de iteraciones.

La elección del factor o ratio de aprendizaje α es crucial, si lo tomamos demasiado pequeño el algoritmo avanzará demasiado lento pero si lo escogemos muy grande el algoritmo tendrá inestabilidad alrededor del mímimo de J. A diferencia del Perceptrón, diferentes valores de pesos iniciales nos darán el mismo resultado tras el entrenamiento. Pese a esto, la red ADALINE solo es capaz de separar clases que tienen una separación lineal. Al introducir un valor tras la fase de entrenamiento la red propagará los valores obtenidos multiplicandolos por los pesos w_{ij} entrenados produciendo así las salidas y_j, el hecho de converger hacia el mínimo de J hace a esta red más robusta analizando datos con ruido.

2.2.2. Ejemplo:Decodificador

Un decodificador tomará números escritos en binario y los convertirá en número decimales, nuestro objetivo en este ejemplo es emplear a la red ADALINE con este propósito. Cuando la red reciba cada una de las entradas buscamos, mediante el aprendizaje, obtener las diversas salidas. La red estará formada por una única neurona con 3 entradas(x_1, x_2 *y* x_3).

x1	x2	x3	Salida
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Iniciamos con un valor en el factor de aprendizaje $\alpha = 0{,}3$ y una precisión de parada requerida de 0,001. Los pesos iniciales tomados son $w = (3'12, 2, 1'86)$, la red irá variando estos pesos de forma proporcional a la diferencia entre la salida obtenida y

la esperada. De esta forma obtenemos la siguiente tabla donde se refleja como los pesos varían minimizando el error en cada paso hasta obtener un valor menor que la precisión requerida.

Iteración	Pesos	Error
1	3.6084984 1.9855184 1.4244324	0.3116599
2	3.8234569 1.9823698 1.20311109	0.06892449
3	3.92060785 1.98678987 1.09678491	0.01502946
4	3.9644138 1.99157779 1.04595917	0.00327488
5	3.98410452 1.99505004 1.02175287	0.00071314

Cuadro 2.1: Ajuste de los pesos en el proceso de entrenamiento

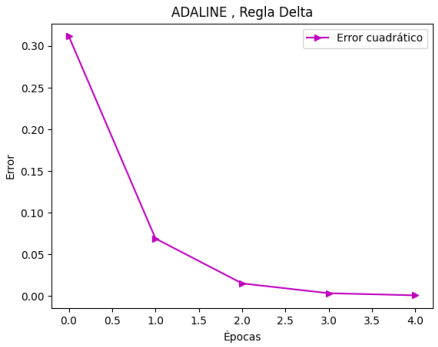


Figura 2.6: Error cuadrático medio a lo largo del entrenamiento

Este proceso está implementado en python y puede encontrarse en 3.3.7.

2.3. Algoritmo del Gradiente Descendente

Método iterativo, dado un punto inicial lo desplaza siguiendo el valor negativo del gradiente hasta un punto crítico que será un mínimo local de la función. Este algoritmo es local pues encontrar el mínimo global de la función sería computacionalmente muy exigente. Es bastante popular para grandes problemas de optimización por su fácil implementación, así como su reducción en el costo de procesamiento a medida que el algoritmo avanza. La principal desventaja es que puede converger muy lento o incluso no converger dependiendo del tamaño del ratio de aprendizaje que elijamos o si el punto inicial es un máximo local de la función.

Dada una función escalar diferenciable $f(\vec{x}) : \mathbb{R}^n \to \mathbb{R}$ y un punto inicial $\vec{x_0} = (x_1^0, \ldots, x_n^0)$, el algoritmo del descenso del gradiente mueve de forma iterativa dicho punto inicial tomando pasos cuya longitud viene determinada por el ratio de aprendizaje α en la dirección del gradiente negativo $- \bigtriangledown f(x_0)$. Dicha

dirección es la que debe tomar x_0 para hacer descender lo más rápido posible el valor de f. Así definimos el siguiente punto del algoritmo como:

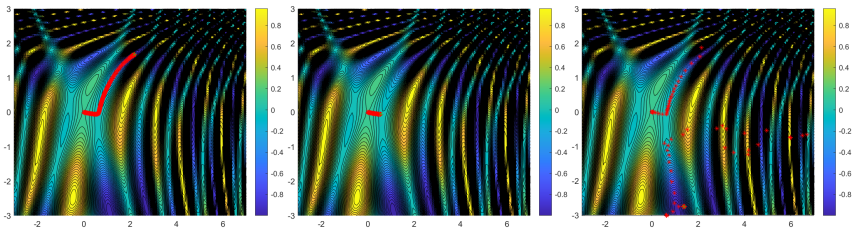
$$\overrightarrow{x_t} = \overrightarrow{x_{t-1}} - \alpha \bigtriangledown f(\overrightarrow{x_{t-1}})$$

Detendremos el algoritmo cuando, dados $\epsilon, \epsilon' > 0$ fijados previamente tengamos que:

- $|| \bigtriangledown f(\overrightarrow{x_{t-1}})|| < \epsilon$ en cuyo caso diremos que el algoritmo converge en un punto crítico.
- $||\overrightarrow{x_{t-1}} - \overrightarrow{x_t}|| < \epsilon'$ en cuyo caso diremos que el algoritmo converge en $\overrightarrow{x_t}$.
- $f(\overrightarrow{x_t}) > f(\overrightarrow{x_{t-1}})$ el algoritmo converge.
- Se alcance el número máximo de iteraciones en cuyo caso variaremos el parámetro α de forma adecuada.

El ratio de aprendizaje debe fijarse antes de iniciar el algoritmo, su valor debe escogerse con cuidado, pues si es muy pequeño el algoritmo tardará mucho en converger y si es muy grande no alcanzará nunca un punto crítico.

A continuación, vemos un ejemplo del uso de este algoritmo sobre la función $f(\theta_1, \theta_2) = sin(\frac{\theta_1^2}{2} - \frac{\theta_2^2}{4} + 3)cos(2\theta_1 + 1 - e^{\theta_1})$:



(a) Valor adecuado (b) Valor inferior (c) Valor superior

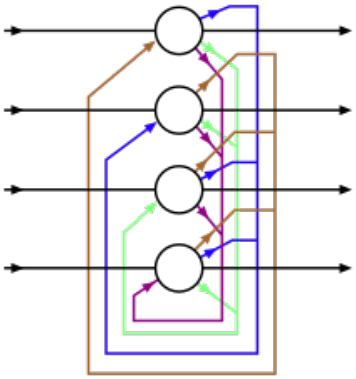
Figura 2.7: Situaciones en función del valor α

La elección del parámetro α es crucial, si el ratio de aprendizaje es muy pequeño el algoritmo tardará mucho en converger hacia un mínimo de la función, como vemos en la imagen central, donde se refleja en rojo las posiciones que adquiere en cada iteración el punto y en verde el punto que se devuelve como mínimo hallado. Si el ratio de aprendizaje es demasiado grande, como en la tercera imagen, el algoritmo no converge en ningún punto crítico. Para un valor adecuado del parámetro de aprendizaje el algoritmo moverá el punto hacia el mínimo de la función como ocurre en la imagen de la izquierda. Podemos ver el código que permite visualizar el entrenamiento del descenso del gradiente en 3.3.7(Python) o en 3.3.7(Matlab). Así como una descripción mas detallada del algoritmo del descenso del gradiente en [5], [16].

2.4. Redes Hopfield

Esta red neuronal fue propuesta por John Hopfield en 1982. Surge un poco antes del modelo Backpropagation, con una arquitectura de red algo distinta a las propuestas hasta entonces. Hasta ese momento las redes que se habían planteado, el Perceptrón y la red ADALINE, eran redes neuronales artificiales que propagaban la información de las primeras capas a las últimas (feedforward) independientemente en cada patrón de entrenamiento, sin embargo, la estructura de red propuesta por Hopfield no sigue este principio.

En la red Hopfield, cada neurona envía su salida como nueva entrada de datos a todas las neuronas de la red salvo a sí misma. Este proceso recibe el nombre de recursividad. Además es una red autoasociativa, durante la etapa de entrenamiento varios patrones se almacenan en la red, luego asocia las nuevas entradas con esta información almacenada. Esto hace que se suela usar en aprendizaje no supervisado, donde se presentan los datos directamente a la red sin tener información sobre la salida esperada.



El desarrollo teórico de las redes de Hopfield puede verse con mayor extensión en [7], [13].

2.4.1. Estructura de la red

La red Hopfield es una red monocapa formada por N neuronas [2], cada neurona se conecta con todas las demás salvo consigo misma. Sea w_{ij} el peso de la conexión entre la neurona i y la j. Estos pesos suponemos que son simétricos $w_{ij} = w_{ji} \quad \forall i, j$ y $w_{ij} = 0 \quad \forall i, j$ con $i = j$.

El primer modelo presentado por Hopfield tenía una función de activación discreta con salidas binarias entre -1/1 o 0/1 haciendo uso de la función escalonada 1.3.1.

$$f(x) = \begin{cases} 1 & si \quad x > \theta_i \\ x & si \quad x = \theta_i \\ -1 & si \quad x < \theta_i \end{cases}$$

La función de activación no modificará el valor recibido si $x = \theta_i$.

θ_i es el umbral de la neurona definido anteriormente, en este caso nos permitirá movernos en el eje de las abscisas. Se suele tomar como valor de θ_i:

$$\theta_i = K \sum_{j=1}^{N} w_{ji} \quad \forall i$$

[2] Número de elementos que componen la información a procesar

Para $k = 0$, $f(x)$ toma los valores $-1/1$, mientras que si $k = \frac{1}{2}$ toma $0/1$.

Los datos se deben presentar como vectores de N componentes. Estas componentes tomarán valores binarios entre 0 y 1 o -1 y 1. Cada combinación recibe el nombre de estado de la red.

Recordemos que la red de Hopfield es autoasociativa. Primero se almacenaran en la red unos valores o estados $\{\overrightarrow{e^1}, \ldots, \overrightarrow{e^M}\}$. A continucación se le presentará una nueva entrada $\vec{e} = (e_1, \ldots, e_N)$ y la red devolverá el valor almacenado que más se parezca a dicha nueva entrada. El proceso matemático que sigue para encontrar dicho valor es el siguiente:

1. Se define $S_i(t = 0) = e_i \quad 1 \leq i \leq N$ y con e_i el elemento i-ésimo de la nueva entrada.

2. Se repite el siguiente proceso hasta que se estabilice, es decir, hasta que $S_i(t + 1) = S_i(t) \quad \forall i$:

$$S_i(t+1) = f(\sum_{j=1}^{N} w_{ji} S_j(t) - \theta_i)$$

   Con f la función de activación definida anteriormente.

Una vez se estabilice, tendremos unos valores $(S_1, S_2, \ldots, S_N)$ que corresponderán al valor almacenado que más se parece a la nueva entrada. Se debe tener en cuenta si todas las neuronas actualizan su salida a la vez o en distintos tiempos. Supondremos que se actualizan de la primera forma.

2.4.2. Aprendizaje

La red Hopfield tiene una primera etapa de aprendizaje antes de introducir los datos a procesar. En esta primera etapa se definen los pesos de las conexiones entre las neuronas i y j mediante el producto de las componentes i-ésima y j-ésima del patrón a almacenar. Suponemos que tenemos M patrones todos presentados en forma de vector N-dimensional, la red de Hopfield discreta asigna los pesos de las conexiones de la siguiente forma:

$$w_{ij} = \begin{cases} \sum_{k=1}^{M} e_i^k e_j^k & si \quad 1 \leq i, j \leq N; i \neq j \\ \\ 0 & si \quad 1 \leq i, j \leq N; i = j \end{cases}$$

O bien si tomamos valores de la función de activación entre $0/1$:

$$w_{ij} = \begin{cases} \sum_{k=1}^{M} (2e_i^k - 1)(2e_j^k - 1) & si \quad 1 \leq i, j \leq N; i \neq j \\ \\ 0 & si \quad 1 \leq i, j \leq N; i = j \end{cases}$$

Definimos la matriz de pesos:

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \ldots & w_{1N} \\ w_{21} & w_{22} & \ldots & w_{2N} \\ \ldots & \ldots & \ldots & \ldots \\ w_{N1} & w_{N2} & \ldots & w_{NN} \end{bmatrix}$$

Con $w_{ij} = w_{ji} \quad \forall i, j$ y $w_{ij} = 0$ si $i = j$, por tanto será una matriz simétrica con una diagonal formada por ceros. Alternativamente podemos definir esta matriz mediante la ecuación:

$$\mathbf{W} = \sum_{k=1}^{M} [E_k^T E_k - I_N]$$

Donde $E_k = (e_1^k, e_2^k, \ldots, e_N^k)$ es el patrón k de entrenamiento, E_k^T es su traspuesta e I_N es la matriz identidad de orden N.

2.4.3. Función de Energía

Introducimos en este apartado la función de energía, muy usada para el desarrollo teórico de estas redes y cuya fórmula es:

$$E = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} S_i S_j + \sum_{i=1}^{N} \theta_i S_i$$

Considerando el espacio de dimensión N donde cada punto representa un estado de la red podemos asignar a cada uno de estos puntos el valor de la función de energía. Una vez se han almacenado los M estados en la primera fase, queremos que estos representen los mínimos de la función de energía E. Al presentar una nueva entrada esta evolucionará hasta alcanzar uno de estos mínimos.

Así, nuestro objetivo en la red de Hopfield autoasociativa es que los patrones a memorizar se sitúen en los mínimos de la función de energía. Esto lo conseguiremos asignando los pesos $w_{ii} = 0 \quad \forall i$ y $w_{ij} = \sum S_j S_i$ (Ver [7]).

2.4.4. Problema del viajante con redes Hopfield

Las redes Hopfield permiten el tratamiento de problemas más complejos que las redes Perceptrón y ADALINE. Una de las aplicaciones de las redes de Hopfield es su uso en la resolución de problemas de optimización. En el siguiente ejemplo resolveremos el problema del viajante, introduciendo de forma breve el desarrollo teórico de la función de energía para este problema.

El problema del viajante se anuncia como sigue: Dadas N ciudades, nuestro objetivo es visitarlas todas una vez, partiendo de una de ellas cualquiera, y recorriendo la menor distancia posible. Este problema se puede abordar haciendo uso de una red Hopfield con NxN neuronas donde cada neurona representa la posibilidad de llegar a la ciudad N en un instante. Se suelen representar mediante una matriz cuadrada de orden N donde las filas indican la ciudad y las columnas el orden en que se visitan. Así, tenemos:

$$S_{ij} = \begin{cases} 1 & si \quad se \quad llega \quad a \quad la \quad ciudad \quad i \quad en \quad el \quad instante \quad j \\ 0 & en \quad otro \quad caso \end{cases}$$

Buscamos una expresión de la función objetivo a minimizar del problema que queremos resolver. Debemos incluir como términos en dicha función las restricciones pues más adelante enlazaremos nuestro problema con la función de energía de la red de Hopfield.

La función objetivo se divide en 4 sumatorios, el primero de ellos penaliza si se toma más de una vez cualquier ciudad. El segundo lo hace cuando se toma más de una ciudad al mismo momento, añadiendo así la restricción de pasar por una sola ciudad en cada momento. El tercer sumatorio asegura que se tomen N puntos en total y el cuarto representa la distancia que deseamos minimizar.

$$F = \frac{A}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{\substack{l=1\\ l\neq j}}^{N} S_{ij}S_{il} + \frac{B}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{\substack{k=1\\ k\neq i}}^{N} S_{ij}S_{kj} + \frac{C}{2}(\sum_{i=1}^{N}\sum_{j=1}^{N} S_{ij} - N)^2 + \frac{D}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{\substack{k=1\\ k\neq i}}^{N} d_{ij}(S_{ij}S_{kj+1} + S_{ij}S_{kj-1})$$

Los valores A, B, C, D son pesos usados para determinar la importancia de cada sumatorio y d_{ij} representa la distancia entre la ciudad i y la ciudad j. Se debe relacionar esta función de coste con la función de energía de la red de Hopfield para este problema:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}\sum_{l=1}^{N} w_{ij\ kl}S_{ij}S_{kl} - \sum_{i=1}^{N}\sum_{j=1}^{N}\theta_{ij}S_{ij}$$

Para que sean equivalentes los pesos de las conexiones de 2 neuronas situadas en ij y en kl deben ser:

$$w_{ij\ kl} = -A\delta_{ik}(1-\delta_{jl}) - B\delta_{jl}(1-\delta_{ik}) - C - D\delta_{ik}(\delta_{jl+1} + \delta_{jl-1})$$

Donde $\delta_i k$ denota la delta de Kronecker, es decir:

$$\delta_{ik} = \begin{cases} 1 \quad si \quad i = k \\ 0 \quad en \quad otro \quad caso \end{cases}$$

El desarrollo en detalle de la red de Hopfield para el problema del viajante podemos encontrarlo en [8]

2.4.5. Ejemplo:Problema del viajante

Para el siguiente ejemplo tomamos un total de 10 ciudades cuyas posiciones son:

- A:(0.25,0.16)
- B:(0.85,0.35)

- C:(0.65,0.24)
- D:(0.70,0.50)
- E:(0.15,0.22)
- F:(0.25,0.78)
- G:(0.40,0.45)
- H:(0.90,0.65)
- I:(0.55,0.90)
- J:(0.60,0.25)

Por tanto nuestra red tendrá $NxN = 100$ neuronas y 9900 conexiones(Recordemos que en la red Hopfield cada neurona se conectaba con todas las demás salvo consigo misma). Haciendo uso del código que se muestra en 3.3.7 obtenemos la siguiente matriz W con el momento en el que se visita cada ciudad.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Por tanto el orden en el que se recorren todas las ciudades una vez y a mínima distancia es: F-I-D-B-H-C-G-E-A-J

2.5. Perceptrón multicapa

En una red neuronal artificial una capa es una agrupación de neuronas que reciben las mismas entradas provenientes de otras neuronas o directamente de los datos, estas capas se representan en los diagramas de redes neuronales en forma de columna y de forma secuencial con otras capas según la dirección de los datos. El perceptrón multicapa es un modelo de red neuronal artificial basado en una estructura que posee al menos 3 capas. La capa que recibe los datos recibe el nombre de capa de entrada, la que da los resultados de la red capa de salida y la capa o capas intermedias entre ambas se llaman capas ocultas pues no tienen contacto directo ni con los datos ni con los resultados , permaneciendo invisibles desde fuera de la red neuronal [3].

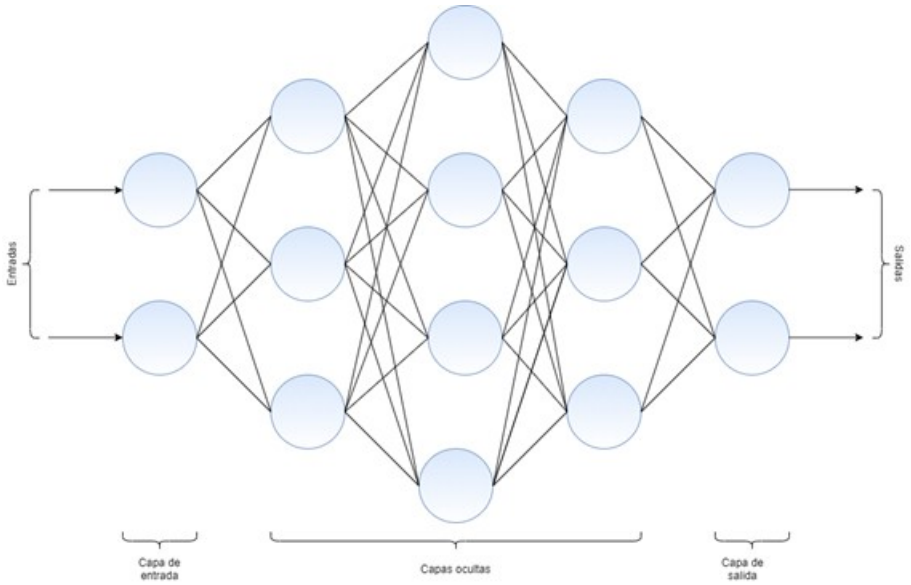


Figura 2.8: Estructura de un perceptrón multicapa

Denotamos por $\overrightarrow{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ las entradas de la red neuronal artificial por la capa de entrada y por $\overrightarrow{y} = (y_1, \ldots, y_n) \in \mathbb{R}^m$ las salidas de la red por la capa de salida. El perceptrón multicapa está completamente conectado, es decir, existen conexiones entre todas las neuronas de una capa y todas las neuronas de la siguiente capa. Dichas conexiones tienen unos pesos que denotaremos por w_{ij}^l siendo el peso que une la neurona i de la capa $l-1$ con la neurona j de la capa l.

Rosenblatt, el creador del perceptrón, ya intuía que el problema señalado por Misky y Papert sobre su modelo podía corregirse añadiendo más capas en serie. Pero esto presentaba un nuevo inconveniente. Hasta ese momento se

[3] Esquema elaborado mediante Draw.io

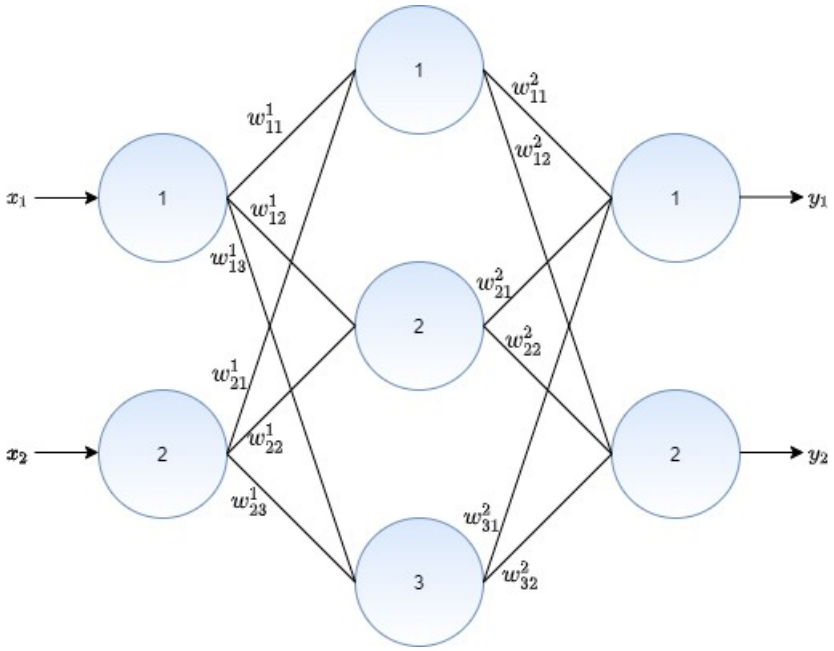


Figura 2.9: Nomenclatura

sabía como entrenar los pesos de la capa de salida, pero no se tenía ningún método computacionalmente asequible que permitiese entrenar los pesos de las capas ocultas. Dicho método llegó en los setenta cuando Paul Werbs propone el algoritmo Backpropagation en su tesis doctoral. Este algoritmo permite entrenar al perceptrón multicapa y abre su aplicación a una gran variedad de problemas de alta complejidad.

2.6. Algoritmo Backpropagation

2.6.1. Introducción

Las redes neuronales monocapa no pueden resolver problemas que requieran una separación no lineal. Tras la publicación de "Perceptrons"[12] en 1969 por Marvin Lee Minsky y Seymour Papert este hecho provocó una ralentización en el desarrollo de las redes neuronales artificiales que terminó gracias a la creación del algoritmo backpropagation. Este algoritmo establece un método mediante el cual las redes neuronales multicapa [4] pueden entrenarse. Se basa en propagar el error desde las últimas capas hacia las primeras mediante el error imputado de una capa. El aprendizaje de las redes neuronales multicapa haciendo uso del backpropagation se hace con conjuntos de datos supervisados y con redes

[4]Con aprendizaje forward y función de activación f continua

neuronales totalmente conectadas. Podemos encontrar un desarrollo más extenso del método Backpropagation en [17], [14] y un desarrollo con ejemplos en [18].

2.6.2. Algoritmo

Tomamos el patrón de entrenamiento p-ésimo con $p \in 1, \cdots, P$, formado por el par $(\vec{x_p}, \vec{d_p})$, donde $\vec{x_p} = (x_{1p}, x_{2p}, \cdots, x_{np}) \in \mathbb{R}^n$ es la entrada p-ésima de datos a la red y $\vec{d_p} = (d_{1p}, d_{2p}, \cdots, d_{mp}) \in \mathbb{R}^m$ es la salida deseada para esa entrada de datos. Tenemos entonces una red con n entradas y m salidas, es decir, tenemos m neuronas en la capa de salida, suponemos esta red neuronal está completamente conectada. Sea y_{ip}^l la componente i-ésima en la salida de la capa l para el patrón de entrenamiento p, definiendo $\vec{y_p^0} = \vec{x_p}$ las entradas del patrón de entrenamiento p. Al estar la red completamente conectada esta salida se pasará como dato de entrada a todas las neuronas de la siguiente capa $l+1$ salvo en el caso que $l = L$ que obtendremos la salida de la red, calculamos el error cometido en dicho patrón mediante el error cuadrático en lo que definimos como la función de costes para p:

$$E_p = \frac{1}{2} \sum_{i=1}^{m} (d_{ip} - y_{ip}^L)^2 \tag{2.4}$$

Es común repetir el proceso de entrenamiento con los P patrones varias veces en lo que se conocen como épocas, para cuantificar el error cometido en las distintas épocas se utiliza el error cuadrático medio de los P patrones de entrenamiento:

$$E = \frac{1}{P} \sum_{p=1}^{P} E_p$$

Nuestro objetivo es minimizar la función de coste E_p(2.4) en cada patrón de entrenamiento. Esto se consigue variando los pesos de las conexiones de las neuronas haciendo uso del algoritmo del descenso del gradiente. Para ello necesitamos calcular las derivadas de la función de costes (2.4) respecto a los pesos de las conexiones, los cuales definimos de la siguiente forma, w_{ij}^l representa el peso de la conexión que va de la i-ésima neurona en la capa $l-1$ a la j-ésima neurona en la capa l. La salida de la capa L-ésima, es decir , la última capa viene definida por la ecuación:

$$y_{ip}^L = f(Neta_{ip}^{L-1}) = f(w_{1i}^L y_{1p}^{L-1} + \cdots + w_{ni}^L y_{np}^{L-1} + \theta^L) \quad i = 1, \ldots, m \tag{2.5}$$

Donde f es la función de activación (lineal o sigmoidal) y $Neta_{ip}^{L-1}$ es la suma ponderada de las entradas a la neurona por los pesos de las conexiones. Por tanto la función de costes es la composición de las siguientes funciones:

$$E_p(f(Neta_{ip}^{L-1}(w_{1i}^L, w_{2i}^L, \ldots, w_{ni}^L, \theta^L)))$$

Luego si queremos calcular $\frac{\partial E_p}{\partial w_{ji}^L}$ las derivadas de la función de coste respecto a los pesos de la última neurona debemos de usar la regla de la cadena:

$$\frac{\partial E_p}{\partial w_{ji}^L} = \frac{\partial E_p}{\partial f}\frac{\partial f}{\partial Neta_{ip}^{L-1}}\frac{\partial Neta_{ip}^{L-1}}{\partial w_{ji}^L} \tag{2.6}$$

Desarrollando cada una de las derivadas tenemos:

$$\frac{\partial E_p}{\partial f} = \frac{\partial E_p}{\partial y_{ip}^L} = -\frac{1}{2}2(d_{ip} - y_{ip}^L) = -(d_{ip} - y_{ip}^L) \tag{2.7}$$

$$\frac{\partial f}{\partial Neta_{ip}^{L-1}} = f' \tag{2.8}$$

$$\frac{\partial Neta_{ip}^{L-1}}{\partial w_{ji}^L} = \frac{\partial w_{1i}^L y_{1p}^{L-1} + \cdots + w_{ji}^L y_{jp}^{L-1} + \cdots + w_{ni}^L y_{np}^{L-1} + \theta^L}{\partial w_{ji}^L} = y_{jp}^{L-1} \tag{2.9}$$

Así, tenemos que la derivada de la función de costes (2.6) es el producto de (2.7) por la derivada de la función de activación (2.8) y por la salida j-ésima de la capa anterior (2.9), correspondiente a la neurona j-ésima de la capa anterior.

Respecto a la derivada de la función de activación (2.8) esta debe ser derivable, por ello tomaremos siempre una función de activación lineal o sigmoidal, si la función de activación es lineal tenemos entonces que $f'(Neta_{ip}^{L-1}) = 1$ y si es sigmoidal $f'(Neta_{ip}^{L-1}) = \frac{e^{-Neta_{ip}^{L-1}}}{(1+e^{-Neta_{ip}^{L-1}})^2}$.

En la ecuación (2.9) obtenemos los valores de la salida de la capa anterior para la derivada de los pesos de las conexiones y obtenemos el peso de la entrada del sesgo, es decir uno, derivando por el bias θ^L . Por lo tanto la ecuación (2.6) queda:

$$\frac{\partial E_p}{\partial w_{ji}^L} = -(d_{ip} - y_{ip}^L)f' y_{jp}^{L-1}$$

$$\frac{\partial E_p}{\partial \theta^L} = -(d_{ip} - y_{ip}^L)f'$$

Definimos el error imputado a la neurona i-esima de la capa L como:

$$\delta_i^L = -(d_{ip} - y_{ip}^L)f' = \frac{\partial E_p}{\partial f}\frac{\partial f}{\partial Neta_{ip}^{L-1}}$$

Así las ecuaciones anteriores nos quedan:

$$\frac{\partial E_p}{\partial w_{ji}^L} = \delta_i^L y_{jp}^{L-1}$$

$$\frac{\partial E_p}{\partial \theta^L} = \delta_i^L$$

Si ahora encontramos una ecuación que relacione las derivadas parciales respecto a los pesos de las conexiones de una capa en función de las derivadas

de los pesos de la capa siguiente podríamos calcular todas las derivadas parciales de todas las conexiones de la red propagando el error hacia las neuronas de las capas anteriores. Queremos calcular entonces $\frac{\partial E_p}{\partial w_{kj}^{L-1}}$, para ello tenemos en cuenta que E_p es la composición de las siguientes funciones:

$$E_p(f^L(Neta_{ip}^{L-1}(f^{L-1}(Neta_{jp}^{L-2}(w_{1j}^{L-1}, w_{2j}^{L-1}, \ldots, w_{n'j}^{L-1})))))$$

Donde f^L denota la función de activación de la capa L y f^{L-1} la función de activación de la capa L-1, luego para calcular $\frac{\partial E_p}{\partial w_{kj}^{L-1}}$ aplicamos de nuevo la regla de la cadena:

$$\frac{\partial E_p}{\partial w_{kj}^{L-1}} = \frac{\partial E_p}{\partial f^L} \frac{\partial f^L}{\partial Neta_{ip}^{L-1}} \frac{\partial Neta_{ip}^{L-1}}{\partial f^{L-1}} \frac{\partial f^{L-1}}{\partial Neta_{jp}^{L-2}} \frac{\partial Neta_{jp}^{L-2}}{\partial w_{kj}^{L-1}} \tag{2.10}$$

No obstante , como $\frac{\partial E_p}{\partial f^L} \frac{\partial f^L}{\partial Neta_{ip}^{L-1}} = \delta_i^L$ la ecuación (2.10) nos queda:

$$\frac{\partial E_p}{\partial w_{kj}^{L-1}} = \delta_i^L \frac{\partial Neta_{ip}^{L-1}}{\partial f^{L-1}} \frac{\partial f^{L-1}}{\partial Neta_{jp}^{L-2}} \frac{\partial Neta_{jp}^{L-2}}{\partial w_{kj}^{L-1}}$$

Tenemos que $\frac{\partial f^{L-1}}{\partial Neta_{jp}^{L-2}} = (f^{L-1})'$ es la derivada de la función de activación de la capa L-1, esta será lineal o sigmoidal, derivadas que ya vimos anteriormente.Por otro lado:

$$\frac{\partial Neta_{jp}^{L-2}}{\partial w_{kj}^{L-1}} = \frac{\partial w_{1j}^{L-1} y_{1p}^{L-2} + \cdots + w_{kj}^{L-1} y_{kp}^{L-2} + \cdots + w_{n'j}^{L-1} y_{n'p}^{L-2} + \theta^{L-1}}{\partial w_{kj}^{L-1}} = y_{kp}^{L-2}$$

$$\frac{\partial Neta_{jp}^{L-2}}{\partial \theta^{L-1}} = \frac{\partial w_{1j}^{L-1} y_{1p}^{L-2} + \cdots + w_{kj}^{L-1} y_{kp}^{L-2} + \cdots + w_{n'j}^{L-1} y_{n'p}^{L-2} + \theta^{L-1}}{\partial \theta^{L-1}} = 1$$

Por último:

$$\frac{\partial Neta_{ip}^{L-1}}{\partial f^{L-1}} = \frac{\partial Neta_{ip}^{L-1}}{\partial y_{jp}^{L-1}} = \frac{\partial w_{1i}^{L} y_{1p}^{L-1} + \cdots + w_{ji}^{L} y_{jp}^{L-1} + \cdots + w_{ni}^{L} y_{np}^{L-1} + \theta^{L}}{\partial y_{jp}^{L-1}} = w_{ji}^{L}$$

Que es el peso de la conexión entre la neurona j de la capa L-1 y la neurona i de la capa L, reescribiendo (2.10):

$$\frac{\partial E_p}{\partial w_{kj}^{L-1}} = \delta_i^L w_{ji}^L (f^{L-1})' y_{kp}^{L-2}$$

$$\frac{\partial E_p}{\partial \theta^{L-1}} = \delta_i^L w_{ji}^L (f^{L-1})'$$

Denotamos entonces el error imputado a neurona j-ésima de la capa L-1 como:

$$\delta_j^{L-1} = \delta_i^L w_{ji}^L (f^{L-1})'$$

En resumen, el error imputado a la neurona i de la última capa viene dado por:

$$\delta_i^L = \frac{\partial E_p}{\partial f} \frac{\partial f}{\partial Neta_{ip}^{L-1}} \tag{2.11}$$

Las derivadas parciales de la función de costes respecto a los pesos de las conexiones y el parámetro de sesgo en la capa l-1 son:

$$\frac{\partial E_p}{\partial w_{kj}^{l-1}} = \delta_j^{l-1} y_{kp}^{l-2} \tag{2.12}$$

$$\frac{\partial E_p}{\partial \theta^{l-1}} = \delta_j^{l-1} \tag{2.13}$$

Siendo δ_j^{l-1} el error imputado a la capa l-1, calculada respecto al error de las capas posteriores y el peso de la conexión correspondiente:

$$\delta_j^{l-1} = \delta_i^l w_{ji}^l (f^{l-1})' \tag{2.14}$$

Con las ecuaciones 2.11, 2.12,2.13 y 2.14 podemos calcular las derivadas parciales de la función de costes en cualquier neurona localizada en cualquier capa. Por 2.11 sabemos calcular el error imputado en la última capa, para ver el error imputado en alguna capa anterior simplemente aplicamos la fórmula 2.14 hasta llegar a dicha capa. Por último, las derivadas de la función de costes las calculamos usando 2.12 y 2.13 según queramos calcular la derivada en función de un peso o un sesgo respectivamente.

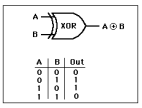
3. Aplicaciones de redes multicapa

En este apartado desarrollaremos las principales aplicaciones que tienen las redes neuronales multicapa, introduciendo en primer lugar desarrollos prácticos del perceptrón multicapa en tareas de reconocimiento de clases y aproximación de funciones. Posteriormente, describiremos algunos modelos más avanzados de redes neuronales multicapa específicos tales como las redes neuronales convolucionales, muy usadas actualmente para el tratamiento de imágenes. Estas redes neuronales añaden algunas funciones a las definidas en el perceptrón multicapa aunque sus principios estructurales se basan en los vistos anteriormente.

3.1. Perceptrón Multicapa:Clasificación de Patrones

La primera de las aplicaciones más comunes del perceptrón multicapa que veremos se trata de la clasificación de elementos en una serie de clases numerables. Supongamos que tenemos unas entradas $\vec{x_1}, \ldots, \vec{x_n} \in \mathbb{R}^n$ a las cuales se les asigna una variable de salida y. Cuyos posibles valores denotan las distintas clases a considerar. En la mayoría de los casos, esta variable respuesta es binaria, reflejando la presencia o ausencia de un factor. Para una salida binaria se necesitará únicamente una neurona en la capa de salida, en general, con m neuronas se puede realizar una clasificación en 2^m clases distintas.

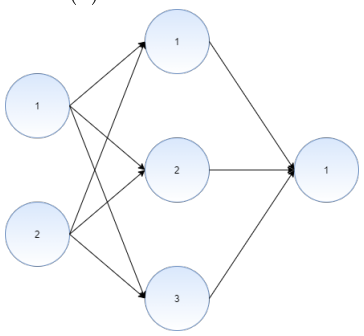
Como vimos anteriormente tanto el Perceptrón como la red ADALINE no podían resolver problemas de clasificación que requiriesen una separación no lineal entre sus clases. Un ejemplo sería la separación de las salidas producidas por una puerta XOR, esta devolverá el valor 0 cuando ambas entradas sean iguales y el valor 1 cuando sean distintas.



Podemos resolver este problema haciedo uso de una red multicapa, añadiendo una capa intermedia en el perceptrón multicapa la red es capaz de realizar una división de forma correcta. En el siguiente ejemplo, introducimos los siguientes datos en formato Excel en la red neuronal cuya estructura mostramos en la imagen, con pesos entre las conexiones y bias aleatorios (Código del ejemplo en 3.3.7).

Entrada A	Entrada B	Salida
0	0	0
1	0	1
0	1	1
1	1	0

(a) Entradas a la red



(b) Estructura de la red

Figura 3.1: Ejemplo puerta XOR

Vemos en la siguiente gráfica como la red va reduciendo el error[1] a medida que aumentan las épocas.

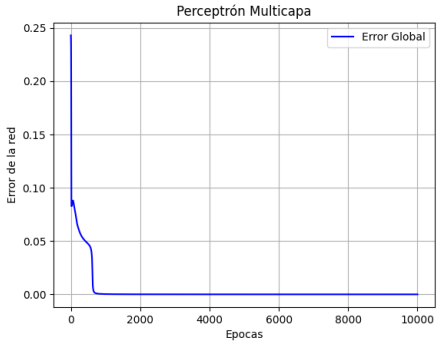


Figura 3.2: Error durante el entrenamiento

[1] Reflejamos el error total E en dicha gráfica

Tras el entrenamiento, vemos como la red obtiene salidas satisfactorias al pasar los distintos casos de la puerta XOR:

Caso	Salida
0 0	0
1 0	1
0 1	1
1 1	0

Cuadro 3.1: Resultados obtenidos

El perceptrón multicapa puede resolver cualquier problema de clasificación binaria siempre que una de las clases se encuentre en una región convexa y cerrada [18]. El número de neuronas y capas dependerá de la complejidad de dicha región.

3.2. Perceptrón multicapa: Curvas de Funciones

Otra aplicación del perceptrón multicapa son los problemas de aproximación de funciones. Supongamos que tenemos un conjunto de entradas y salidas de una función desconocida cuyo comportamiento queremos predecir. Haciendo uso del perceptrón multicapa podemos aproximar el comportamiento de dicha función. Si g es la función de activación de las neuronas del perceptrón multicapa, suponemos que g es continua [2] entonces haciendo uso de una sola capa oculta podemos aproximar cualquier función f real y continua [18]. En concreto:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} \lambda_i g_i^1(Neta_i^1)$$

Donde g_i^1 son las funciones de activación de las neuronas de la capa intermedia, λ_i son constantes que determinarán el peso de cada función de activación y $Neta_i^1$ son las entradas ponderadas por los pesos tal y como se definieron en 1.2.2.

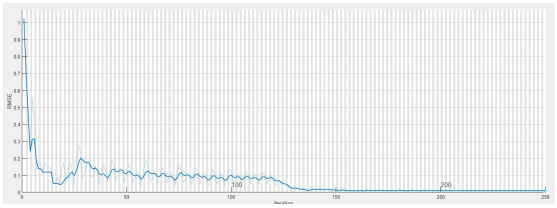
$$Neta_i^1 = \sum_{j=1}^{n} w_{ji}^1 x_j - w_0^i$$

Usaremos la identidad como función de activación en la última capa que posee una única neurona y determinaremos los pesos de las funciones como los pesos de las conexiones entre las neuronas de la capa intermedia y la neurona de la capa de salida, es decir, $\lambda_i = w_{1i}^2$.

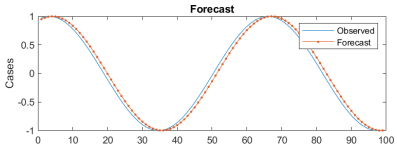
En el siguiente ejemplo (3.3.7), utilizaremos el perceptrón multicapa para aproximar la función $f(x) = sin(x)$. Para ello, tomaremos todos los enteros en el intervalo $[0, 200]$ como datos de entrenamiento y los enteros del intervalo

[2] Generalmente se usará la función sigmoide o la función tangente hiperbólica

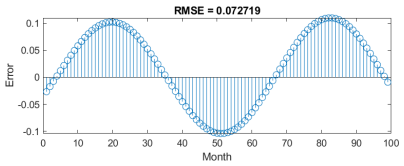
$[201, 300]$ como datos de test. En la siguiente gráfica podemos ver el proceso de entrenamiento de la red que realiza Matlab [3].



Como vemos, el error se reduce a medida que el entrenamiento avanza, esto ajustará los pesos λ_i. Por último las siguiente gráficas muestran el desempeño de nuestra red en los 100 puntos del conjunto de test y el error cometido en los mismos.



(a) Aproximación frente a valor real de la función



(b) Error cometido en cada punto del intervalo

Como vemos el RMSE tiene un comportamiento sinusoidal y un valor relativamente bajo lo que indica que el resultado fue bastante preciso en este ejemplo.

[3] Mediante la función train del paquete NeuralToolbox

3.3. Redes neuronales convolucionales

3.3.1. Introducción

Es un tipo de red neuronal artificial con aprendizaje supervisado, surgió en 1998 de la mano de Yann Lecun y su principal utilidad radica en la capacidad de "Analizar imágenes" que otorga a las redes neuronales. Los datos de entrada de este tipo de redes suelen ser imágenes en forma de matrices, donde cada valor representa un píxel. Estas entradas estarán formadas por una sola matriz, es decir, tendrán un solo canal si se trata de una imagen en escala de grises o 3 canales para el formato RGB, donde cada píxel estará representado por 3 valores que corresponden al nivel de saturación de cada uno de los colores rojo, azul y verde.

Usar imágenes como datos de entrada presenta una complicación añadida a lo visto hasta el momento. Al procesar las matrices correspondientes a los píxeles de las imágenes se debe tener en cuenta no solo la información que obtenemos del propio píxel , sino el entorno que rodea al mismo. Por ejemplo, una línea de píxeles de color negro rodeados de píxeles de color blanco en una imagen en escala de grises podría indicar un margen de una figura. Nuestra red neuronal artificial debe ser capaz de analizar esto y para ello, se hace uso de una nueva operación matemática llamada convolución, que definiremos en la siguiente sección.

Otro problema es el coste computacional que supone el tratar con imágenes. Para solventar esto, las redes neuronales convolucionales utilizan una operación denominada Pooling. Su utilidad radica en la generalización de características para la reducción del costo. Introduciremos esta operación en segundo lugar.

Podemos consultar de forma más extensa información teórica sobre las redes convolucionales en [3] o [11]. Así mismo, la guía de matlab[1] presenta una serie de ejemplos de uso de su paquete DeepLearningToolbox.

3.3.2. Convolución

Decimos que una red neuronal artificial es convolucional si al menos en una de sus capas aplica una operación denominada convolución.

En el caso continuo, dadas las funciones I y K definimos la convolución como:

$$I * K(t) = \int_{-\infty}^{+\infty} I(a)K(t-a)da \tag{3.1}$$

En el caso discreto la operación de convolución se define como sigue:

$$I * K(t) = \sum_{a=-\infty}^{+\infty} I(a)K(t-a) \tag{3.2}$$

En las redes neuronales convolucionales usaremos la expresión 3.2, siendo I los datos de entrada, generalmente píxeles de una imagen representados en matrices. K el filtro o núcleo de la convolución, que se aplicará a lo largo de toda

la entrada. Al resultado lo llamaremos mapa de características de la entrada para dicho filtro y lo denotaremos por S. Una convolución puede tomar como entrada tanto a una imagen como a un conjunto de m mapas de carecterísticas producidos por alguna convolución anterior.

Denotaremos por N al número de canales o mapas de características de la entrada, N=1 en escala de grises, y N=3 en formato RGB si estamos con los datos de entrada de la red. Cada una de las longitudes de la entrada se denotará por $i_j \quad j = 1, 2$, respectivamente $k_j \quad j = 1, 2$ serán las longitudes del filtro, donde $k_j \leq i_j \quad j = 1, 2$.

Así, expresamos matemáticamente la convolución (para $N = 1$) de los datos de entrada I haciendo uso del filtro K como:

$$S(i,j) = \sum_{m=1}^{k_1} \sum_{n=1}^{k_2} I(i-m+1, j-n+1) K(i,j)$$

O de forma equivalente [11]:

$$S(i,j) = \sum_{m=1}^{k_1} \sum_{n=1}^{k_2} I(i+m-1, j+n-1) K(m,n)$$

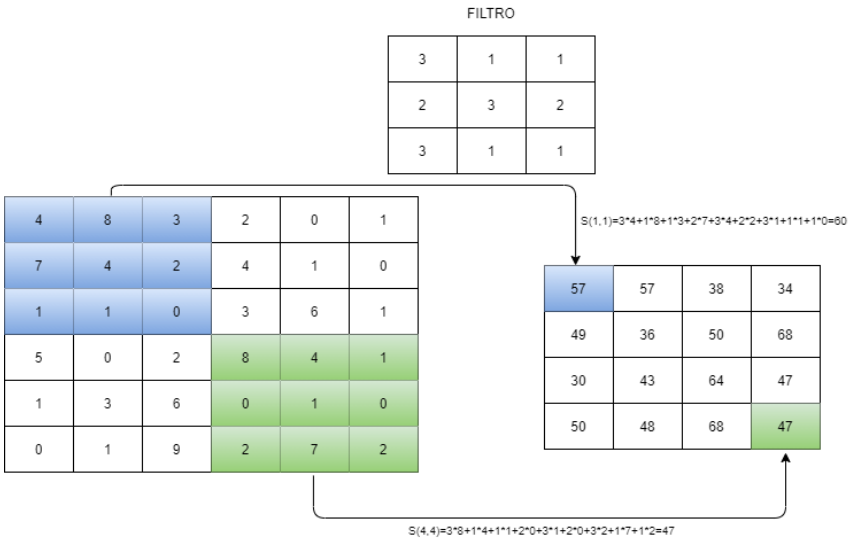
Si $N > 1$ el proceso es idéntico salvo en la definición del filtro, este deberá tomar siempre todos los canales al recorrer la imagen en cada salto, para ello, definiremos N matrices para el filtro K. Sea $I^l(m,n)$ el valor de la entrada a la convolución en la posición de altura m y anchura n del canal l-ésimo con $l = 1, \ldots, N$, $K^l(m,n)$ el mismo elemento para el canal l-ésimo del filtro, tenemos:

$$S(i,j) = \sum_{m=1}^{k_1} \sum_{n=1}^{k_2} \sum_{l=1}^{N} I^l(i+m-1, j+n-1) K^l(m,n)$$

El número de canales de la salida de la convolución vendrá determinado por el número de filtros distintos que apliquemos, es decir, los canales serán iguales a la cantidad de mapas de características que definamos para una misma entrada I pero con distintos núcleos K [3]. De esta forma, al ir avanzando una imagen en la red neuronal convolucional iremos perdiendo altura y anchura en la misma pero esta irá ganando mayor profundidad gracias al uso de múltiples filtros.

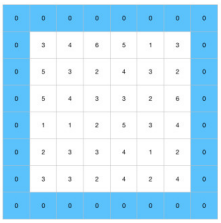
En el siguiente ejemplo aplicamos una convolución a una imagen en escala de grises, N=1. Las dimesiones de la imagen son $i_1 = 6$ que denota la altura de la imagen e $i_2 = 6$ que denota la anchura. El nucleo que usamos tiene dimensiones $k_1 = 3$ de altura y $k_2 = 3$ de anchura. En general, se suelen tomar imágenes cuadradas en las entradas así como filtros cuadrados en las convoluciones [4].

[4] Imagen del ejemplo creada en draw.io



Como vemos, la convolución se aplica a lo ancho y alto de la imagen, desplazando el filtro hasta que nos quedamos sin píxeles en la imagen de entrada. En el ejemplo anterior el desplazamiento del filtro tenía un ancho de 1 píxel tanto en la altura como en la anchura, no obstante se pueden definir convoluciones con saltos superiores, para ello denotamos como s_j con $j = 1, 2$ los saltos que realiza el filtro en cada dimensión.

La salida de una convolución producirá una reducción en todas las dimensiones de la entrada. Así mismo, al tener definida de esta forma la convolución, se pierde la información de los píxeles situados en los bordes. Para solventar este problema, se suelen añadir filas de ceros en todas las dimensiones de la entrada en lo que se conoce como zero-padding. El filtro podrá recorrer posiciones extra y al tener estos nuevos elementos el valor 0 no aportarán ninguna información extra al resultado. Denotaremos por p_j con $j = 1, 2$ el zero-padding que se añade respectivamente en la altura y anchura de la entrada[5].



Ejemplo zero-padding

El uso de los núcleos K es vital en las redes neuronales convolucionales, en cada convolución se hacen uso de varios núcleos distintos para la misma entrada. Esto permite obtener diversas caracterísitcas de la entrada como las

[5]Si tenemos N>1 se añaden filas de ceros en todos los canales.

lineas verticales que aparecen en la misma, las lineas horizontales, los contornos o incluso la información de una sección de la entrada solo. El número de filtros que se usan en una convolución definirá el número de canales en la salida, es decir, si en una convolución de una imagen en escala de grises se usan 3 filtros distintos, la salida de la misma tendrá dimensiones $i_1 \times i_2 \times 3$.

A continuación vemos un ejemplo de como un filtro permite obtener diversas características de una imagen, en la siguiente imagen [6] aplicamos dos filtros distintos usados para detectar tanto lineas horizontales como verticales.

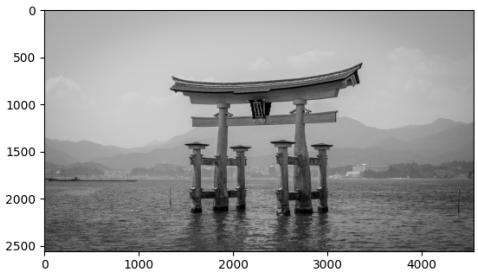


Figura 3.4: Imagen original en escala de grises

La imagen se encuentra en escala de grises, por tanto el filtro que aplicamos tendrá que ser considerado para un único canal. En caso de tener varios canales en la imagen habría que definir varias matrices para el filtro y aplicar cada una a un canal sumando los resultados para obtener un único valor de salida para cada posición del filtro. En este ejemplo vamos a usar los filtros llamados filtro Sobel [23] que permite encontrar bordes verticales y horizontales.

$$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Filtro Sobel para detectar bordes horizontales

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 2 & 1 \end{pmatrix}$$

Filtro Sobel para detectar bordes verticales

[6] Sacada de Unsplash, página que provee imágenes de uso libre.

Aplicamos ambos filtros a la matriz producida por los píxeles de la imagen anterior en escala de grises, desplazando un píxel a la derecha el filtro hasta llegar al borde derecho de la imagen para desplazarlo 1 píxel hacia abajo y repetir hasta llegar al final de la misma. Obtenemos entonces las siguientes salidas.

(a) Vertical (b) Horizontal

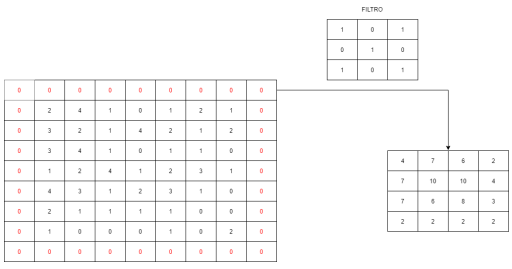
Figura 3.5: Filtro Sobel 3.3.7

Como comentamos anteriormente, los saltos de píxeles producen una reducción en la salida de la convolución. Si suponemos que tenemos una convolución cuya entrada tiene dimensión i, si añadimos P filas de ceros en ambas direcciones y tomamamos los filtros saltando S píxeles. La salida de la convolución tendrá dimensiones o, donde:

$$o = \left[\frac{i + 2p - k}{s}\right] + 1$$

Donde los corchetes denotan a la operación floor, que asigna a la entrada el mayor entero menor que el valor introducido.

Por ejemplo, si tomamos una imagen cuyas dimensiones son 8×8, a la que además añadimos una zero-padding de tamaño 1 y aplicamos en la misma un filtro de tamaño 3×3 que salta 2 píxeles en cada desplazamiento nuestras dimensiones en la salida serían 4×4.



Una convolución viene seguida de una función de activación, generalmente se suele usar la función RELU. Estas actúan de una forma similar a como lo hacían con las redes neuronales multicapa. Se aplica sobre valores de todos los canales justo después de haberse realizado una convolución. Así, al aplicar una función de activación RELU no se cambian las dimensiones de salida. Se pueden usar otras funciones de activación aunque se debe tener cuidado si las mismas tienen un coste computacional elevado.

Basándose de nuevo en principios biológicos, muchas redes neuronales convolucionales usan en su estructura una normalización en las diferentes salidas de los filtros de la convolución que les permite comparar entre diversos filtros

en las capas posteriores. Esta normalización puede aplicarse a todos los filtros mediante la fórmula:

$$b_i = \frac{a_i}{(k + \alpha \sum_i a_i^2)^\beta} \quad i = 1, \ldots, N$$

Donde i denota el filtro, a_i la salida original del mismo en una posición (x, y), b_i la salida normalizada y k, α, β son hiperparámetros que variarán según la red. En los último años han surgido normalizaciones que dividen los filtros en lotes, no obstante, en las redes neuronales convolucionales que usaremos no intervienen este tipo de normalizaciones.

3.3.3. Pooling

La última de las operaciones de las redes neuronales convolucionales es el pooling. Esta operación va tomando regiones de tamaño $P_q \times P_q$ en todos los canales de forma independiente y recorriendo toda la entrada de forma similar a como lo hacían los filtros en las convoluciones. Solo que en lugar de calcular el producto interno, esta operación devolverá el máximo de todos los píxeles en dicha región(max-pooling) o la media de los mismos(Average-pooling). Aquí toman mayor protagonismo los desplazamientos entre aplicaciones del filtro anteriormente definidos pues ayudan a reducir redundancias. Estos cambios se aplican a cada canal de forma independiente, por tanto, los pooling no modifican la profundidad de la entrada.

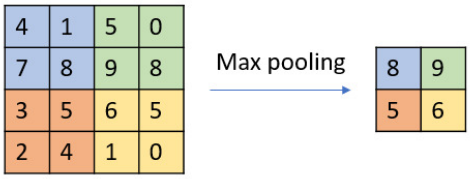


Figura 3.6: Max-pooling con desplazamiento de 2

La estructura que suelen tomar las redes neuronales convolucionales es ir iterando convoluciones y funciones de activación RELU. Intercalando cada cierto número de combinaciones una capa de pooling para reducir así el coste y eliminar redundancias.

Las redes neuronales convolucionales utilizan un aprendizaje jerarquizado en su estructura. Las primeras capas de convolución se encargan de detectar pequeños elementos como líneas, círculos o cruces. Mientras que las capas convolcionales posteriores se encargan de combinar esta información para procesar estructuras más complejas como puede ser una rueda, una cara o un perro.

Al final de la red se ponen una serie de capas con neuronas completamente conectadas entre sí de la misma forma que se vio en el perceptrón multicapa. El número de capas completamente conectadas dependerá del problema que queramos resolver y se usarán para obtener una salida que se pueda interpretar en la red. La última de estas capas suele ser una capa softmax para problemas con respuestas categóricas y cuyo número de neuronas viene determinado por el número de categorías. Para conectar estas capas con las capas finales de las convoluciones se enlazan todas las salidas de la última convolución con todas

las neuronas de la primera de las capas totalmente conectadas. Debido a esto, la última parte de las redes convolucionales suele tener una gran cantidad de parámetros a entrenar, pero estos se pueden entrenar mediante el método de Backpropagation visto en 2.6.

Para entrenar los parámetros de los filtros de las convoluciones se usará de igual forma el algoritmo Backpropagation pero teniendo en cuenta una serie de particularidades que veremos seguidamente.

3.3.4. Entrenamiento

Para el entrenamiento de las redes neuronales convolucionales se usará el algoritmo Backpropagation explicado anteriormente. Esto plantea una serie de cuestiones, se sabe como aplicar este algoritmo en el perceptrón multicapa, no obstante, en las redes convolucionales entran en juego nuevas operaciones como es el pooling y la convolución.

Al aplicar el algoritmo Backpropagation en el pooling debemos tener en cuenta como propagar el error hacia atrás en esta operación, es decir, cuales fueron las neuronas de la capa i que generaron esa salida en la capa $i+1$ al introducir los datos. En el caso del max-pooling [7], si no tenemos solapamiento se establece la propagación del error en 0 para aquellos elementos en la región del filtro de la capa i que no coincidan con el valor máximo asignado en la capa $i+1$, propagando así el error en aquellos que sean iguales al máximo solo.

Solo queda establecer la forma en la que determinar el error producido en las convoluciones. Para ello definimos la convolución de forma matricial, supongamos que tenemos una convolución con desplazamiento de 1 y sin zero-padding. Cuya entrada tiene dimensiones $L_q \times B_q$ y un solo canal, el filtro tiene dimensiones $F_q \times F_q$. Suponemos también que tenemos solo un canal en la salida, en caso de considerar más, simplemente repetimos el proceso siguiente para todos los filtros. La salida tendrá dimensiones $(L_q - F_q + 1) \times (B_q - F_q + 1) \times 1$. En primer lugar tenemos que transformar la matriz de entrada [8] en un vector de longitud $A_0 = L_q \times B_q$ al que llamaremos $\overline{f}$, para ello proyectamos cada fila en orden descendente. Aplicar un filtro en una posición determinada corresponderá a multiplicar $\overline{f}^T$ por la izquierda por un vector de dimensión A_0 con ceros en todas sus posiciones salvo en las que se esté aplicando el filtro. Si tomamos todas las posibles posiciones del filtro tendremos $A_1 = (L_q - F_q + 1) \times (B_q - F_q + 1)$ vectores con los que formaremos la matriz de la convolución a la que llamaremos C que tendrá dimensiones $A_1 \times A_0$. El resultado será un vector de A_1 elementos que corresponderán a la salida de la convolución para ese filtro.

[7] La mayoría de las estructuras predefinidas usan max-pooling en lugar de average-pooling
[8] Este proceso se aplica para cada filtro en caso de tener varios

Por ejemplo, si tenemos la matriz $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ como entrada, esta se transformará en el vector $\overline{f} = (1, 2, 3, 4, 5, 6, 7, 8, 9)$, si queremos ahora aplicarle el filtro $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ a todas las posiciones posibles en las condiciones definidas antes debemos de usar la matriz $C = \begin{pmatrix} a & b & 0 & c & d & 0 & 0 & 0 & 0 \\ 0 & a & b & 0 & c & d & 0 & 0 & 0 \\ 0 & 0 & 0 & a & b & 0 & c & d & 0 \\ 0 & 0 & 0 & 0 & a & b & 0 & c & d \end{pmatrix}$. Para aplicar la convolución multiplicamos:

$$C * \overline{f}^T = \begin{pmatrix} a & b & 0 & c & d & 0 & 0 & 0 & 0 \\ 0 & a & b & 0 & c & d & 0 & 0 & 0 \\ 0 & 0 & 0 & a & b & 0 & c & d & 0 \\ 0 & 0 & 0 & 0 & a & b & 0 & c & d \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{pmatrix} = \begin{pmatrix} a + 2b + 4c + 5d \\ 2a + 3b + 5c + 6d \\ 4a + 5b + 7c + 8d \\ 5a + 6b + 8c + 9d \end{pmatrix}$$

Para entrenar los parámetros de los filtros mediante Backpropagation debemos tomar como entrada el vector de dimensiones A_1 siguiendo con la notación anterior, a este vector lo llamaremos $\overline{g}$. Para propagar el error hacia atrás debemos tener en cuenta en cuantos lugares intervino la celda cuando se ejecutó de forma normal, esto se obtiene multiplicando a $\overline{g}$ por C^T por la izquierda [9]. En el caso de tener una profundidad en la salida mayor a 1, realizamos el proceso anterior en cada capa y sumamos:

$$\sum_{k=1}^{d} C_k^T \overline{g}_k^T$$

Donde C_k^T es la matriz asociada al filtro k-ésimo de la convolución y $\overline{g_k}$ es la salida que produjo.

[9] Puede encontrarse un desarrollo más extenso de como usar Backpropagation en convolucion en [3]

3.3.5. Redes pre-entrenadas

Aunque es posible elaborar y entrenar una red neuronal convolucional, el entrenamiento de las mismas suele ser bastante lento y costoso debido a la cantidad de parámetros que existen. Cuantas más capas tenga la red neuronal artificial más costará entrenarla [15], [25]. No obstante, existe una serie de arquitecturas para redes neuronales convolucionales creadas y que han sido entrenadas en el reconocimiento de una gran variedad de objetos a través de imágenes. Estas redes surgen como propuestas de solución a los problemas de la competición ILSVRC [10] que se organiza anualmente. Esta competición provee de amplios datasets de imágenes que deben ser clasificados según una serie de clases. Los métodos que mejores soluciones han obtenido en dicha competencia en los últimos años han sido precisamente las redes neuronales convolucionales.

A continuación se presenta una de las redes neuronales convolucionales pre-entrenadas. Esta red adquirió el mejor porcentaje de acierto en la competencia de 2012 del ILSVRC. Dicha estructura recibió el nombre de Alexnet [10] y aunque en los últimos años se han presentado otras estructuras con mejores resultados, el impacto que tuvo Alexnet en el desarrollo de las redes neuronales convolucionales fue bastante notable y su estructura relativamente sencilla la hace perfecta para presentarla como una aproximación a este tipo de redes pre-entrenadas. Para acceder a ella hacemos uso del paquete DeepToolbox de Matlab que permite entre otras cosas importar redes pre-entrenadas.

3.3.6. Alexnet

Esta red recibe imágenes en formato RGB , es decir, que su entrada tendrá 3 canales, está diseñada para imágenes de dimensiones $227 \times 227 \times 3$. Como vemos en la imagen, se compone de 8 capas en total, 5 convoluciones con sus respectivas capas de activación RELU así como algún max-pooling y alguna normalización en algunas de ellas. La primera convolución aplica 96 filtros de tamaño $11 \times 11 \times 3$ y con un stride[11] de 4, luego se aplica una función de activación RELU y una normalización [12]. Seguido a esto se realiza un max-pooling de tamaño 3×3 y stride de 2, por tanto este tendrá solapamiento. La segunda convolución aplica 256 filtros de tamaño $5 \times 5 \times 96$ viene seguida también de una función de activación RELU y una normalización, así como un max-pooling de tamaño 3×3 y stride 2. La tercera y cuarta capa convolucional vienen solo seguidas por una función de activación RELU y aplican 384 filtros de tamaño $3 \times 3 \times 256$ y 384 de tamaño $3 \times 3 \times 384$ respectivamente. La quinta y última de las convoluciones utiliza 256 filtros de tamaño $3 \times 3 \times 384$ y viene seguida por una función de activación RELU y un último max-pooling de tamaño 3×3 y stride 2. A continuación vienen 3 capas totalmente conectadas, donde las 2 primeras tienen 4096 neuronas en total y vienen seguidas por una función de activación RELU. La última de las capas totalmente conectadas tiene 1000 neuronas que

[10] ImageNet Large Scale Visual Recognition Challenge

[11] Forma inglesa de referirse al número de píxeles que salta el filtro

[12] Las normalizaciones en Alexnet tienen como hiperparámetros $k = 1, \beta = 0{,}75, \alpha = 0{,}0001$

corresponden a las 1000 clases diferentes en las que es capaz de clasificar Alexnet, viene seguida de una función de activación Softmax que otorgará la probabilidad de pertenencia de cada clase.

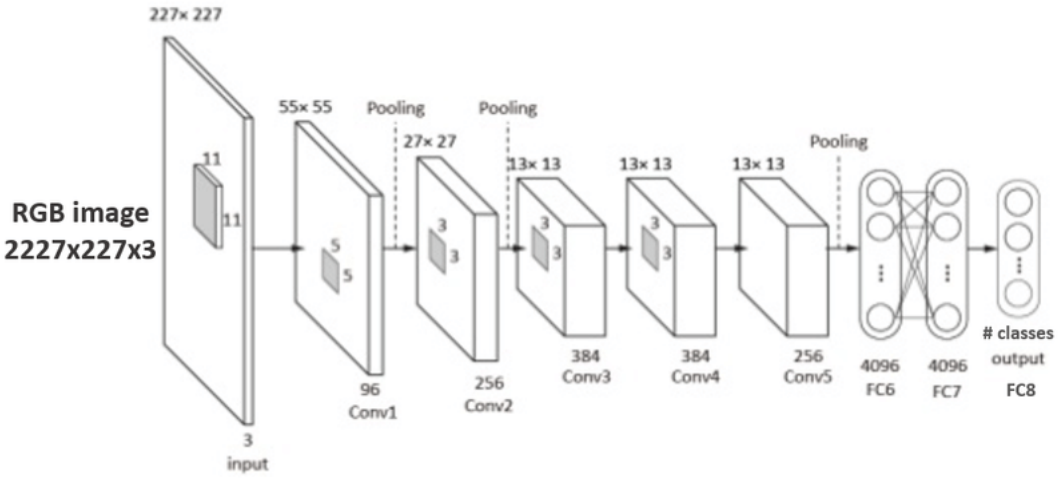
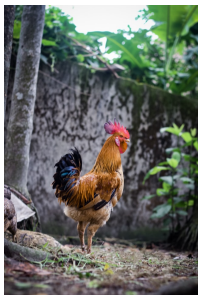


Figura 3.7: Estrucura de Alexnet

3.3.7. Ejemplo:Alexnet

En el siguiente ejemplo haremos uso de la red pre-entrenada Alexnet para clasificar las siguiente imágenes [13]



En este ejemplo simplemente deberemos importar la red con los valores de los pesos entrenados, escalar las imágenes para que tengan dimensiones $227 \times 227 \times 3$ y pasarlas por la red para ver si las clasifica correctamente así como el grado de especificación que tiene Alexnet. Al usar pesos pre-entrenados de antemano el

[13] Sacadas de Unsplash.

coste computacional del programa será considerablemente más bajo que el que tendríamos si entrenáramos la red para este ejemplo.

Vemos los resultados de la clasificación por parte de la red neuronal convolucional artificial Alexnet:

La red no solo fue capaz de clasificar el perro y la flor como tal, sino que pudo identificar la raza de perro y el tipo de flor que se mostraban en la imagen. En concreto Alexnet es capaz de diferenciar con bastante exactitud un total de 1000 clases distintas.

Podemos encontrar el código para importar la red y realizar la clasificación en 3.3.7 haciendo uso de la herramienta Deeptoolbox de Matlab, que permite importar y manejar redes pre-entrenadas de una forma sencilla y con una interfaz clara. Además Matlab cuenta con una guía de usuario para este paquete que es actualizada anualmente [1]. Destacar, del mismo modo, algunos paquetes que permiten elaborar modelos de redes neuronales artificiales en Python[4] como son Keras o Tensorflow[26].

En el anexo [3.3.7], podemos encontrar un ejemplo haciendo uso de la red diseñada por Google donde se puede apreciar como actuan las distintas capas convolucionales a lo largo de la red.

Bibliografía

[1] Mark Hudson Beale, Martin T Hagan, and Howard B Demuth. Neural network toolbox. *User's Guide, MathWorks*, 2:77–81, 2010.

[2] Eduardo Francisco Caicedo Bravo and Jesús Alfonso López Sotelo. *Una aproximación práctica a las redes neuronales artificiales*. Programa Editorial UNIVALLE, 2009.

[3] C Aggarwal Charu. *Neural Networks and Deep Learning*. 2018.

[4] Joshua Eckroth. *Python artificial intelligence projects for beginners: Get up and running with artificial intelligence using 8 smart and exciting AI applications*. Packt Publishing Ltd, 2018.

[5] Kris Hauser. "b553 lecture 4: Gradient descent, 2012.

[6] Donald Olding Hebb. The organization of behavior; a neuropsycholocigal theory. *A Wiley Book in Clinical Psychology*, 62:78, 1949.

[7] José Ramón Hilera González, Víctor José Martínez Hernando, et al. *Redes neuronales artificiales: fundamentos, modelos y aplicaciones*. 2000.

[8] John J Hopfield and David W Tank. "neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

[9] Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, et al. Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav). In *International conference on machine learning*, pages 2668–2677. PMLR, 2018.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[11] G.P. Martinsanz, P.J.H. Caro, and E.B. Portas. *Aprendizaje profundo*. RC Libros, 2021. ISBN 9788412106985.

[12] Minsky Marvin and A Papert Seymour. Perceptrons, 1969.

[13] Damián Jorge Matich. Redes neuronales: Conceptos básicos y aplicaciones. *Universidad Tecnológica Nacional, México*, 41:12–16, 2001.

[14] James L McClelland, David E Rumelhart, PDP Research Group, et al. *Parallel Distributed Processing, Volume 2: Explorations in the Microstructure of Cognition: Psychological and Biological Models*, volume 2. MIT press, 1987.

[15] Umberto Michelucci. *Advanced applied deep learning: convolutional neural networks and object detection*. Springer, 2019.

[16] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[17] Nikolay Nikolaev and Hitoshi Iba. *Adaptive learning of polynomial networks: genetic programming, backpropagation and Bayesian methods*. Springer Science & Business Media, 2006.

[18] IVAN NUNES and HERNANE SPATTI DA SILVA. *Artificial neural networks: a practical course*. Springer, 2018.

[19] Cathy Oñeil. *Weapons of math destruction: How big data increases inequality and threatens democracy*. Broadway Books, 2016.

[20] Kevin L Priddy and Paul E Keller. *Artificial neural networks: an introduction*, volume 68. SPIE press, 2005.

[21] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[22] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[23] I Sobel and G Feldman. An isotropic 3x3 image gradient operator. presentation at stanford ai project, 1968.

[24] Pedro Isasi Viñuela and Inés M Galván León. *Redes de neuronas artificiales: un enfoque práctico*. Pearson Prentice Hall, 2003.

[25] Steven Walczak and Narciso Cerpa. Heuristic principles for the design of artificial neural networks. *Information and software technology*, 41(2): 107–117, 1999.

[26] Giancarlo Zaccone, Md Rezaul Karim, and Ahmed Menshawy. *Deep learning with TensorFlow*. Packt Publishing Ltd, 2017.

Referencias Web

1. https://www.youtube.com/watch?v=Uz7ucmqjZ08&ab_channel=DotCSV
2. https://www.youtube.com/watch?v=90QDe6DQXF4&ab_channel=DotCSV
3. https://www.youtube.com/watch?v=gicnvDwdDxY&list=WL&index=65&t=344s&ab_channel=HackeandoTec
4. https://towardsdatascience.com/history-of-the-first-ai-winter-6f8c2186f80b

Anexo:GoogleNet

Introducción

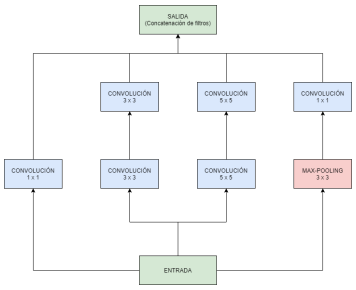
La red GoogleNet fue desarrollada por el equipo de investigadores de Google. Obtuvo el mejor resultado en la competición ILSVRC en el año 2014. Sus aplicaciones abarcan desde la detección de objetos en sistemas de conducción autónoma hasta reconocimiento facial a través de imágenes.

Esta estructura de red está compuesta por 20 capas entre las que se incluyen capas Inception. Las capas Inception surgen bajo el objetivo de aumentar la anchura en las capas de las redes convolucionales sin disparar el coste computacional de las mismas. Como vimos anteriormente, este coste es bastante elevado sobre todo cuando empezamos a trabajar con estructuras de redes más complejas.

Capas Inception

Las capas Inception, a diferencia de las capas convolucionales explicadas hasta ahora, aplican varias convoluciones de forma paralela con filtros de distinto tamaño. Suponemos que la entrada tiene dimensiones $m \times n \times p$, donde p es el número de mapas de características o canales de la entrada y m, n la altura y anchura. La capa Inception aplicará convoluciones de distinto tamaño de forma paralela en esta entrada para luego juntar todas estas salidas concatenando el total de filtros usados.

En concreto, las capas Inception usadas en la red GoogleNet aplican 4 convoluciones de forma paralela en al entrada. La primera de ellas tiene como tamaño de filtro $k = 1$, las 2 siguientes aplican 2 convoluciones seguidas de tamaño $k = 3$ y $k = 5$ respectivamente. La última de las convoluciones paralelas($k = 1$) se aplica tras haber realizado un Max-Pooling de tamaño $k = 3$. Todas las convoluciones tienen en su salida una función de activación RELU. Gracias a la computación paralela en una misma capa conseguimos una mayor anchura en la salida, lo cual se traduce en un aprendizaje



Capa Inception usada en GoogleNet

mejor para la red.

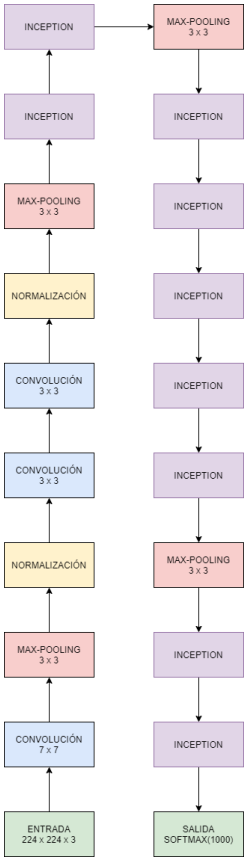
Existen versiones más avanzadas de capas Inception que utilizan filtros no cuadrados y normalización, no obstante estas capas no son usadas en la red GoogleNet por lo que no las trataremos. Puede encontrarse más información sobre las capas Inception en [11].

Red GoogleNet

Pasamos ahora a analizar la estructura de la red GoogleNet, como comentamos antes esta red está formada por 20 capas en total. De esas 20 capas, 9 son capas Inception cuya estructura comentamos en el apartado anterior. Esta red obtuvo el mejor resultado en la competición ILSVRC en el año 2014 con una tasa de error del 6 %, muy cercana al nivel humano.

La red recibe imágenes de tamaño 224×224 en RGB, luego el número de canales iniciales será $p = 3$. Aplica una primera convolución[14] con $k = 7$ y 64 filtros en total seguida de un Max-Pooling cuyo tamaño de filtro es $k = 3$. Luego aplica una normalización como la vista en la red Alexnet. A continuación aplica otras 2 convoluciones ambas con $k = 3$ pero con 64 y 192 filtros cada una. Una última normalización y un Max-Pooling($k = 3$) se hace antes de pasar a ejecutar las 9 capas Inception de forma secuencial. Entre la segunda y tercera capa Inception se realiza un Max-Pooling con $k = 3$, así como entre la séptima y la octava. Tras ejecutar todas las capas Inception se realiza un Average-Pooling($k = 3$). Ya en último lugar tenemos una capa totalmente conectada con 1000 neuronas y con una función de activación Softmax que nos devolverá la probabilidad de pertenencia de la imagen en las 1000 clases en las que es capaz de clasificar la red GoogleNet.

Recordemos que después de cada convolución se aplica una función de activación, en concreto la función de activación RELU.



Estructura red GoogleNet

[14] Seguida de una función de activación RELU

Ejemplo: Red GoogleNet

En el siguiente ejemplo (Código en [3.3.7]) usaremos la red GoogleNet para clasificar la siguiente imagen:

Chow(96 %)

Esta imagen ya fue clasificada haciendo uso de la red Alexnet en un ejemplo anterior. En este ejemplo visualizaremos la salida obtenida por las convoluciones a lo largo de la red GoogleNet, de esta forma podremos ver como trabajan las redes neuronales convolucionales a lo largo del proceso de clasificación de imágenes.

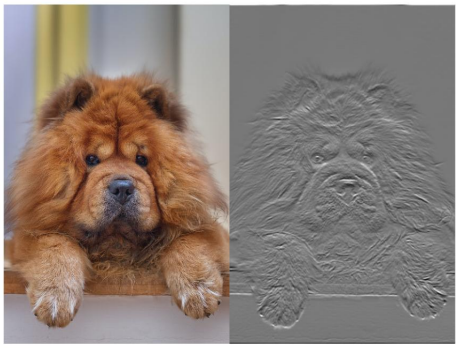
Mostrar todos los filtros de todas las convoluciones llevaría un alto coste computacional así como generaría una cantidad enorme de imágenes. Por tanto, mostraremos algunos filtros de cada convolución así como el filtro que mayor valor ha dado en la salida. Las imágenes resultantes se pasan a imágenes en blanco y negro donde el color blanco indica una fuerte activación en la convolución mientras que el color negro indica una baja activación.

A continuación vemos 64 filtros de la primera de las convoluciones que se aplican en esta red. Estas primeras convoluciones suelen emplearse para detectar elementos básicos como son los bordes, las líneas o los contrastes en el la imagen.

Filtros primera convolucion

Si mostramos el filtro que obtuvo una mayor activación vemos que este está

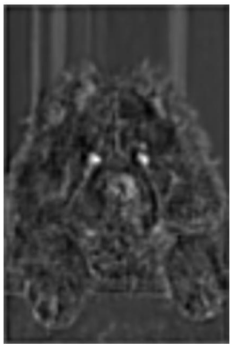
centrado en la detección de bordes en la imagen. A este nivel en la red convolucional todavía no se ha distorsionado la imagen.



Resultado con mayor activación

Veamos ahora los resultados de 64 filtros de una convolución situada en la parte media de la red. La imagen ya comienza a ser distorsionada por la aplicación consecutiva de filtros.

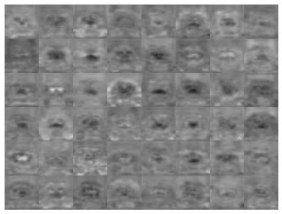
Filtros convolución



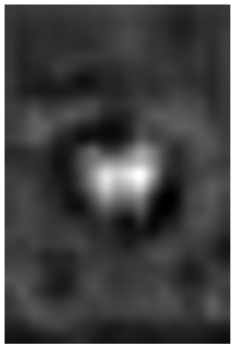
Filtro con mayor activación

Si miramos el filtro que obtuvo una mayor activación en la convolución vemos como este se centra en la detección de ojos en la figura.

Los filtros de las últimas convoluciones buscan figuras más complejas mediante la combinación de los mapas de características obtenidos hasta el momento. El resultado al observas su desempeño son imágenes bastante distorsionadas con zonas grandes donde se activan estas convoluciones.



Filtros convolución



Filtro con mayor activación

El filtro con mayor activación en esta convolución final muestra el enfoque que realiza la red en la parte del morro y los ojos del perro para identificarlo así como una ligera atención en la parte del pelaje.

Anexo:Códigos

Código en Python descenso de gradiente

```
import numpy as np
import scipy as sc
import matplotlib.pyplot as plt

#Funcion anonima en python
func = lambda th: np.sin(1/2 * th[0] ** 2
- 1/4 * th[1] ** 2 + 3)*np.cos(2 * th[0]
+ 1 - np.e ** th[1])


res = 100
_X = np.linspace(-2, 2, res)
_Y = np.linspace(-2, 2, res)

#Evaluaremos en esos 100x100 puntos
_Z = np.zeros((res, res))

for ix, x in enumerate(_X):
    for iy, y in enumerate(_Y):
        _Z[iy, ix] = func([x, y])

#Graficamos
plt.contourf(_X, _Y, _Z, 100)
plt.colorbar()


#Generamos un punto aleatorio
Theta = np.random.rand(2) * 4 - 2

####EMPEZAMOS A APLICAR EL DESCENSO DEL GRADIENTE######

_T = np.copy(Theta)
```

```
h = 0.001 #Paso de la derivada

lr = 0.001 #Ratio de aprendizaje

plt.plot(Theta[0], Theta[1], "o", c="white")

grad = np.zeros(2)

#Calculamos las derivadas parciales
for _ in range(10000):
    for it,th in enumerate(Theta):
        _T = np.copy(Theta)

        _T[it] = _T[it] + h

        deriv = (func(_T) - func(Theta)) / h

        grad[it] = deriv

    Theta = Theta - lr * grad

    if(_%100 == 0):
        plt.plot(Theta[0], Theta[1], ".", c="red")

plt.plot(Theta[0], Theta[1], "o", c="green")
plt.show()
```

Código en Matlab descenso de gradiente

```
res = 100;
X = linspace(-3,7,res);
Y = linspace(-3,3,res);
Z = zeros(res,res);
for i = 1:length(X)
    item_x = X(i);
    for j = 1:length(Y)
        item_y = Y(j);
        Z(j,i) = fun(item_x,item_y);
    end
end

%Theta = rand(1,2) * 4 - 1
Theta = [0,0];
hold on
contourf(X,Y,Z,50)
```

```
colorbar

h = 0.001; %Pasos en la derivada
lr = 0.5; %hiperparametro del aprendizaje
grad = zeros(2); %Vector gradiente
%Aplicamos descenso del gradiente
for i=1:10000
    for j=1:length(Theta)
        T = Theta;
        T(j) = T(j) + h;
        deriv=(fun(T(1),T(2))-fun(Theta(1),Theta(2))) / h;
        grad(j) = deriv;
    end
    Theta = Theta - lr*grad;
    plot(Theta(1),Theta(2),'r*')
end
plot(Theta(1),Theta(2),'g*')

function z = fun(x,y)

    z = sin(1/2*x^2 -1/4*y^2 +3)*cos(2*x + 1 - exp(y));

end
```

Código en Python problema del viajero

```
import numpy as np
from random import randint
from random import uniform

class Hopfield:
    def __init__(self, cities, d, alpha):
        self.cities = cities
        self.neurons = cities**2
        self.d = d
        self.alpha = alpha

        self.w = np.zeros([self.neurons, self.neurons])

    def f(self, x):
        return 0.5*(1.0+np.tanh(self.alpha*x))

    def train(self, u, A, B, C, D, sigma):
        n = self.cities

        for iteration in range((n**2)):
```

```
            x = randint(0, n - 1)
            i = randint(0, n - 1)
            tmpA = 0
            for j in range(n):
                if i != j:
                    tmpA += u[x][j]
            tmpA *= -A
            tmpB = 0
            for y in range(n):
                if x != y:
                    tmpB += u[y][i]
            tmpB *= -B
            tmpC = 0
            for y in range(n):
                for j in range(n):
                    tmpC += u[y][j]
            tmpC -= (n+sigma)
            tmpC *= -C
            tmpD = 0
            for y in range(n):
                if 0 < i < n - 1:
                    tmpD += self.d[x][y]*(u[y][i+1] +
                    u[y][i-1])
                elif i > 0:
                    tmpD += self.d[x][y]*(u[y][i-1])
                elif i < n-1:
                    tmpD += self.d[x][y]*(u[y][i+1])
            tmpD *= -D
            u[x][i] = self.f(tmpA + tmpB + tmpC + tmpD)
        return u

    def predict(self, A, B, C, D, sigma, max_iterations):
        u = np.zeros([self.cities, self.cities])
        for i in range(self.cities):
            for j in range(self.cities):
                u[i][j] = uniform(0, 0.03)

        prev_error = self.calc_error(u, A, B, C, D, sigma)
        repeated = 0
        max_repeat = 10
        for iteration in range(max_iterations):
            u = self.train(u, A, B, C, D, sigma)
            error = self.calc_error(u, A, B, C, D, sigma)
            if error == prev_error:
                repeated += 1
            else:
```

```
                repeated = 0

            if repeated > max_repeat:
                break
            prev_error = error
        return u

    def calc_error(self, u, A, B, C, D, sigma):
        tmpA = 0
        n = self.cities
        for x in range(n):
            for i in range(n):
                for j in range(n):
                    if i != j:
                        tmpA += u[x][i]*u[x][j]
        tmpA *= (A/2.0)

        tmpB = 0
        for i in range(n):
            for x in range(n):
                for y in range(n):
                    if x != y:
                        tmpB += u[x][i] * u[y][i]
        tmpB *= (B/2.0)

        tmpC = 0
        for x in range(n):
            for i in range(n):
                tmpC += u[x][i]
        tmpC -= ((n+sigma)**2)
        tmpC *= (C/2.0)

        tmpD = 0
        for x in range(n):
            for y in range(n):
                for i in range(n):
                    if 0 < i < n - 1:
                        tmpD += self.d[x][y]
                        *u[x][i]*(u[y][i+1]+u[y][i-1])
                    elif i > 0:
                        tmpD += self.d[x][y]
                        *u[x][i]*(u[y][i-1])
                    elif i < n - 1:
                        tmpD += self.d[x][y]
                        *u[x][i]*(u[y][i+1])
        tmpD *= (D/2.0)
```

```
        return tmpA+tmpB+tmpC+tmpD


def calc_d(cities):
        n = cities.shape[0]
        d = np.zeros([n, n])
        for i in range(n):
                for j in range(n):
                        d[i][j] = np.sqrt(
                        np.square(cities[i][0] -
                        cities[j][0]) +
                         np.square(cities[i][1]
                        - cities[j][1]))
        return d

city[0] = (0.25, 0.16)
city[1] = (0.85, 0.35)
city[2] = (0.65, 0.24)
city[3] = (0.70, 0.50)
city[4] = (0.15, 0.22)
city[5] = (0.25, 0.78)
city[6] = (0.40, 0.45)
city[7] = (0.90, 0.65)
city[8] = (0.55, 0.90)
city[9] = (0.60, 0.25)

d = calc_d(city)

hp = Hopfield(n, d, 50.0)
v = hp.predict(A=100.0, B=100.0, C=90.0, D=100.0,
 sigma=1, max_iterations=1000)
print(v)
```

Código en Python decodificación con red ADALINE

```
import numpy as np
import matplotlib.pyplot as plt

class ADALINE():

    #   Constructor
    def __init__(self, d, xi, n_muestras, wi, fac_ap,
     epochs, precision, w_ajustado):
        self.d = d
        self.xi = xi
```

```
        self.n_muestras = n_muestras
        self.wi = wi
        self.fac_ap = fac_ap
        self.epochs = epochs
        self.precision = precision
        self.w_ajustado = w_ajustado
        self.y = 0 # Salida de la red

    def Entrenamiento(self):
        E = 1    # Error de salida
        E_ac = 0 # Error total
        Error_prev = 0 # Error anterior
        Ew = 0 # Error cuadratico medio
        E_red = [] # Error de la red
        E_total = 0 # Error total

        while np.abs(E) > self.precision:
            Error_prev = Ew
            for i in range(self.n_muestras):
                #Calculo salida de la red
                self.y = sum(self.xi[i, :]*self.wi)
                # Calculo del error
                E_ac = (self.d[i] - self.y)
                # Reajuste pesos
                self.wi += self.fac_ap*E_ac*self.xi[i, :]
                E_total = E_total + (E_ac**2)

            print('Pesos= ', self.wi)

            # Calculo error cuadratico medio
            Ew = ((1/self.n_muestras) * E_total)
            E = (Ew - Error_prev) #Error de la red
            E_red.append(np.abs(E)) # Almacenamos errores
            self.epochs += 1
            print('error=',E)
        return self.wi, self.epochs, E_red

    def F_operacion(self):
        salida = []
        for j in range(self.n_muestras):
            self.y = sum(self.xi[j, :] * self.w_ajustado)
            salida.append(self.y)
        return salida
```

```
# Ciclo principal
if __name__=="__main__":
    # Datos de entrada
    xi = np.array([[0,0,0],[0, 0, 1], [0, 1, 0],
     [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0],
      [1, 1, 1]])

    # Salidas esperadas
    d = np.array([[0],[1], [2], [3], [4], [5], [6], [7]])

    # Longitud
    n_muestras = len(d)

    # Establecemos los pesos
    wi = np.array([3.12, 2.0, 1.86])

    # Factor de aprendizaje
    fac_ap = 0.3

    # Epocas
    epochs = 0

    # Precision
    precision = 0.001

    w_ajustado = []

    # Inicializar la red ADALINE
    red = ADALINE(d,xi,n_muestras,wi,fac_ap,epochs,
    precision,w_ajustado)
    w_ajustado, epocas, error = red.Entrenamiento()
    # Grafica
    plt.ylabel('Error')
    plt.xlabel('Epocas')
    plt.title('ADALINE , Regla Delta')
    x = np.arange(epocas)
    plt.plot(x, error, 'm->', label="Error cuadratico")
    plt.legend(loc='upper right')
    plt.show()
    print("Pesos ajustados", w_ajustado)
```

Código en Matlab clasificación perceptrón

```
clear all
clc

TABLA_DATOS=[
0  0  1  1  0  2;
0  1  0  1  2  2;
0  0  0  1  0  1];


ENTRADA=[TABLA_DATOS(1,:);TABLA_DATOS(2,:)];
SALIDA=[TABLA_DATOS(3,:)];

hold on
plotpv(ENTRADA,SALIDA)
hold off
clf

MI_RED=newp(minmax(ENTRADA),1);
view(MI_RED)

MI_RED.iw{1,1}=[1  1];
MI_RED.b{1}=-0.75;

hold on
plotpv(ENTRADA,SALIDA)
plotpc(MI_RED.iw{1,1},MI_RED.b{1})
hold off
clf
MI_RED=train(MI_RED,ENTRADA,SALIDA);
plotpc(MI_RED.iw{1,1},MI_RED.b{1})

xrojo=[];
yrojo=[];
xazul=[];
yazul=[];
for k=0:0.05:2
    for j=0:0.05:2
xpunto=k;
ypunto=j;
valor_salida=sim(MI_RED,[xpunto;ypunto]);
if valor_salida==0
    xrojo=[xrojo,xpunto];
    yrojo=[yrojo,ypunto];
```

```
end
if valor_salida==1
    xazul=[xazul,xpunto];
    yazul=[yazul,ypunto];
end
    end
    hold on
    plot(xrojo,yrojo,'r.')
    plot(xazul,yazul,'g.')
    plotpv(ENTRADA,SALIDA)
    plotpc(MI_RED.iw{1,1},MI_RED.b{1})
    hold off
end
```

Código en Python aplicación filtros Sobel

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import cv2
import numpy as np

image = mpimg.imread('foto.jpg')
plt.imshow(image)
plt.show()


gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
plt.imshow(gray, cmap='gray')
plt.show()

# horizontal
sobel_y = np.array([[ -1, -2, -1],
                    [ 0, 0, 0],
                    [ 1, 2, 1]])
# vertical
sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])

filtered_image1 = cv2.filter2D(gray, -1, sobel_x)
filtered_image2 = cv2.filter2D(gray, -1, sobel_y)
f, ax = plt.subplots(1, 2, figsize=(15, 4))
ax[0].set_title('Deteccion de borde horizontal')
ax[0].imshow(filtered_image1, cmap='gray')
ax[1].set_title('Deteccion de borde vertical')
```

```
ax[1].imshow(filtered_image2, cmap='gray')
plt.show()
```

Código Imágenes Alexnet

```
net = alexnet;

Img1 = imread('ruta')
imshow(Img1)
Img_r1 = imresize(Img1,[227,227])
label1 = classify(net,Img_r1);
imshow(Img_r1)
title(string(label1))

Img2 = imread('ruta')
imshow(Img2)
Img_r2 = imresize(Img2,[227,227])
label2 = classify(net,Img_r2);
imshow(Img_r2)
title(string(label2))

Img3 = imread('ruta')
imshow(Img3)
Img_r3 = imresize(Img3,[227,227])
label3 = classify(net,Img_r3);
imshow(Img_r3)
title(string(label3))
```

Código Ejercicio Perceptrón Multicapa:Clasificación

```
import numpy as np
import pandas as pd
from numpy import random
import matplotlib.pyplot as plt


class MLP():
    # Constructor
    def __init__(self, xi, d, w_1, w_2, us, uoc, precision
    , epocas, fac_ap, n_ocultas, n_entradas, n_salida):
        self.xi = np.transpose(xi)
        self.d = d
        self.w1 = w_1
        self.w2 = w_2
        self.us = us
```

```
        self.uoc = uoc
        self.precision = precision
        self.epocas = epocas
        self.fac_ap = fac_ap
        self.n_entradas = n_entradas
        self.n_ocultas = n_ocultas
        self.n_salida = n_salida
        # Varaibles de aprendizaje
        self.di = 0 # Salida deseada en la
        iteracion actual
        self.error_red = 1 # Error total
        de la red en un conjunto de entrenamiento
        self.Ew = 0 # Error cuadratico medio
        self.Error_prev = 0 # Error anterior
        self.Errores = []
        self.Error_actual = np.zeros((len(d)))
        self.Entradas = np.zeros((1,n_entradas))
        self.un = np.zeros((n_ocultas,1))
        self.gu = np.zeros((n_ocultas,1))
        self.Y = 0.0
        self.y = 0.0
        self.epochs = 0
        # Varaibles de retropropagacion
        self.error_real = 0
        self.ds = 0.0   # Delta de salida
        self.docu = np.zeros((n_ocultas, 1))
    def Operacion(self):
        respuesta = np.zeros((len(self.d), 1))
        for p in range(len(self.d)):
            self.Entradas = self.xi[:, p]
            self.Propagar()
            respuesta[p, :] = self.y
        return respuesta.tolist()

    def Aprendizaje(self):
        Errores = [] # Almacenamos los errores
        while np.abs(self.error_red) > self.precision:
            for i in range(len(d)):
                self.Entradas = self.xi[:,i]
                self.di = self.d[i]
                self.Propagar()
                self.Backpropagation()
                self.Propagar()
                self.Error_actual[i] = (0.5)
                 * ((self.di - self.y)**2)
            # Error global de la red
```

```
            self.Error()
            Errores.append(self.error_red)
            self.epochs += 1
            # Si se alcanza un numero mayor de epocas
            if self.epochs > self.epocas:
                break
        # Regresar
        return self.epochs, self.w1, self.w2,
                        self.us, self.uoc, Errores


    def Propagar(self):
        # Operaciones en la primera capa
        for a in range(self.n_ocultas):
            self.un[a, :] = np.dot(self.w1[a, :],
             self.Entradas) + self.uoc[a, :]

        # Calcular la activacion de las neuronas de
         la capa oculta
        for o in range(self.n_ocultas):
            self.gu[o, :] = tanh(self.un[o, :])

        # Calcular Y potencial de activacion de
         las neuronas de salida
        self.Y = (np.dot(self.w2, self.gu) + self.us)

        # Calcular la salida de la neurona de salida
        self.y = tanh(self.Y)


    def Backpropagation(self):
        # Calcular el error
        self.error_real = (self.di - self.y)
        # Calculamos ds(delsta de salida)
        self.ds = (dtanh(self.Y) * self.error_real)
        # Ajustamos pesos sinapticos w2
        self.w2 = self.w2 + (np.transpose(self.gu)
        * self.fac_ap * self.ds)
        # Ajustamos umbral us
        self.us = self.us + (self.fac_ap * self.ds)
        # Calcular docu
        self.docu = dtanh(self.un) * np.transpose(self.w2)
        * self.ds
        # Ajustar los pesos w1
        for j in range(self.n_ocultas):
            self.w1[j, :] = self.w1[j, :]
```

```
            + ((self.docu[j, :]) * self.Entradas
             * self.fac_ap)

        # Ajustar el umbral en las neuronas ocultas
        for g in range(self.n_ocultas):
            self.uoc[g, :] = self.uoc[g, :] +
             (self.fac_ap * self.docu[g, :])


    def Error(self):
        # Error cuadratico medio
        self.Ew = ((1/len(d)) * (sum(self.Error_actual)))
        self.error_red = (self.Ew - self.Error_prev)


# Funcion para la tanh
def tanh(x):
    return np.tanh(x)


# Funcion para la derivada de la tanh
def dtanh(x):
    return 1.0 - np.tanh(x)**2


# Funcion Sigmoide
def sigmoide(x):
    return 1/(1+np.exp(-x))


# Funcion para obtener la derivada de la funcion
def dsigmoide(x):
    s = 1/(1+np.exp(-x))
    return s * (1-s)


def Datos_entrenamiento(matriz, x1, xn):
    xin = matriz[:,x1:xn+1]
    return xin


def Datos_validacion(matriz, xji, xjn):
    xjn = matriz[:, xji:xjn+1]
    return xjn
```

```
# Programa principal
if "__main__" == __name__:
    xls = pd.ExcelFile('XOR.xlsx')
    datos = xls.parse('Hoja1')
    matrix_data = np.array(datos)
    # Datos entrada
    x_inicio = 0
    x_n = 1
    # Datos de validacion
    xj_inicio = 3
    xj_n = 4
    # Crear vector de entrada xi
    xi = (Datos_entrenamiento(matrix_data
    , x_inicio, x_n))
    d = matrix_data[:, x_n+1]
    # Crear vector de validacion
    xj = (Datos_validacion(matrix_data,
     xj_inicio, xj_n))
    # Parametros de la red
    f, c = xi.shape
    fac_ap = 0.2
    precision = 0.000001
    epocas = 10000
    epochs = 0
    # Arquitectura de la red
    n_entradas = c # Numero de entradas
    cap_ocultas = 1 # 1 Capa oculta
    n_ocultas = 5 # Numero de neuronas
     en la capa oculta
    n_salida = 1 # Numero de neuronas
     en la capa de salida
    # Valor de umbral o bias
    us = 0.1 # Bias neurona salida
    uoc = np.ones((n_ocultas, 1), float)
    # Matriz de pesos sinapticos
    random.seed(0)
    w_1 = random.rand(n_ocultas, n_entradas)
    w_2 = random.rand(n_salida, n_ocultas)

    # Inicializamos la red PMC
    red = MLP(xi, d, w_1, w_2, us, uoc,
     precision, epocas, fac_ap, n_ocultas,
      n_entradas, n_salida)
    epochs, w1_a, w2_a, us_a, uoc_a, E =
     red.Aprendizaje()
```

```
    # Graficar el error
    plt.grid()
    plt.ylabel("Error de la red")
    plt.xlabel("Epocas")
    plt.title("Perceptron Multicapa")
    x = np.arange(epochs)
    plt.plot(x, E, 'b', label = "Error Global")
    plt.legend(loc = 'upper right')
    plt.show()

    # Validacion
    red = MLP(xj, d, w1_a, w2_a, us_a, uoc_a,
     precision, epocas, fac_ap, n_ocultas, n_entradas,
      n_salida)
    salidas = red.Operacion()
    print("Salidas: ",salidas)
```

Código Ejercicio Perceptrón Multicapa:Funciones 2

```
clear all

x=0:200;
xx=201:300;

%datos originales
dataTrain=sin(x/10);
dataTest=sin(xx/10);

%media
mu = mean(dataTrain);
sig = std(dataTrain);

%datos estandarizados
dataTrainStandardized = (dataTrain - mu) / sig;
dataTestStandardized = (dataTest - mu) / sig;
XTest = dataTestStandardized(1:end-1);

%preparar los datos para la prediccion
XTrain = dataTrainStandardized(1:end-1);
YTrain = dataTrainStandardized(2:end);

%definimos y ponemos a entrenar la red
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%PARAMETROS DE ENTRENAMIENTO
```

```
numFeatures = 1;
numResponses = 1;
numHiddenUnits = 200;
layers = [
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits)
    fullyConnectedLayer(numResponses)
    regressionLayer];
options = trainingOptions('adam', ...
    'MaxEpochs',250, ...
    'GradientThreshold',1, ...
    'InitialLearnRate',0.005, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',125, ...
    'LearnRateDropFactor',0.2, ...
    'Verbose',0, ...
    'Plots','training-progress');


%entrenamiento de la red
net = trainNetwork(XTrain,YTrain,layers,options);

analyzeNetwork(net)

%predicciones
net = predictAndUpdateState(net,XTrain);
[net,YPred] = predictAndUpdateState(net,YTrain(end));
numTimeStepsTest = numel(XTest);
for i = 2:numTimeStepsTest
    [net,YPred(:,i)] = predictAndUpdateState(net,
    YPred(:,i-1),'ExecutionEnvironment','cpu');
end

%desestandarizamos
YPred = sig*YPred + mu;

%errores
YTest = dataTest(2:end);
rmse = sqrt(mean((YPred-YTest).^2))

figure
subplot(2,1,1)
plot(YTest)
hold on
plot(YPred,'.-')
hold off
```

```
legend(["Observed" "Forecast"])
ylabel("Cases")
title("Forecast")

subplot(2,1,2)
stem(YPred - YTest)
xlabel("Month")
ylabel("Error")
title("RMSE = " + rmse)
```

Código en Matlab GoogleNet

```
%Cargamos la red
net = googlenet;

%Cargamos la Imagen
im = imread('D:\Cosas Juan\Apuntes4\TFG\NEURALNETWORK\
Versiones\Perro.jpg');
imshow(im)
%Altura y Anchura de la imagen
imgSize = size(im);
imgSize = imgSize(1:2);

%Analizamos al red
analyzeNetwork(net)

%Primera convolucion
act1 = activations(net,im,'conv1-7x7_s2');

%Mostramos los resultados de 64 filtros
sz = size(act1);
act1 = reshape(act1,[sz(1) sz(2) 1 sz(3)]);
I = imtile(mat2gray(act1),'GridSize',[8 8]);
imshow(I)
%Vemos el filtro 17
act1ch17 = act1(:,:,:,17);
act1ch17 = mat2gray(act1ch17);
act1ch17 = imresize(act1ch17,imgSize);
I = imtile({im,act1ch17});
imshow(I)

%Encontramos el filtro cuya salida es mayor
[maxValue,maxValueIndex] = max(max(max(act1)));
act1chMax = act1(:,:,:,maxValueIndex);
act1chMax = mat2gray(act1chMax);
```

```
act1chMax = imresize(act1chMax,imgSize);
I = imtile({im,act1chMax});
imshow(I)
%Vemos una convolucion posterior
act6 = activations(net,im,'inception_3b-3x3');
sz = size(act6);
act6 = reshape(act6,[sz(1) sz(2) 1 sz(3)]);
I = imtile(imresize(mat2gray(act6),[64 64]),
'GridSize',[6 8]);
imshow(I)
%Mostramos la mas potente
[maxValue6,maxValueIndex6] = max(max(max(act6)));
act6chMax = act6(:,:,:,maxValueIndex6);
imshow(imresize(mat2gray(act6chMax),imgSize))
%Vemos una convolucion posterior
act7 = activations(net,im,'inception_5b-1x1');
sz = size(act7);
act7 = reshape(act7,[sz(1) sz(2) 1 sz(3)]);
I = imtile(imresize(mat2gray(act7),[64 64]),
'GridSize',[6 8]);
imshow(I)
%Mostramos la mas potente
[maxValue6,maxValueIndex6] = max(max(max(act7)));
act6chMax = act7(:,:,:,maxValueIndex6);
imshow(imresize(mat2gray(act6chMax),imgSize))
```