Universidad Miguel Hernández De Elche

Facultad de Ciencias Sociales y Jurídicas de Elche

Grado en Estadística Empresarial

NEURAL NETWORKS FROM SCRATCH

Trabajo fin de Grado

(Junio 2021)

Autor: Gurwinder Singh

Tutor: Jose Luis Sainz-Pardo Auñon

ÍNDICE

1.- Resumen

Este trabajo es realizado sobre las redes neuronales que son un sub campo de *Machine Learning*. Las redes neuronales utilizan *supervised learning* para aprender, las redes neuronales tienen su inspiración en las neuronas biológicas. En este trabajo se demuestra la estructura sobre la cual se asientan las redes neuronales.

En el paso de *forward propagation* vemos como las redes neuronales pasan la información de una capa a otra. Utilizando el producto matricial agilizamos este proceso de pasar la información hacía adelante para predecir. Una vez hecho la predicción necesitamos alguna manera para medir el error que se comete por lo tanto se explica lo que es *loss function* y cómo mide el error.

Los *weights* (pesos) que son los parámetros de nuestra red neuronal, son optimizados haciendo uso del cálculo y su regla de cadena. Podemos optimizar, entrenar los pesos para que el modelo cometa menos error. El paso para entrenar los pesos es conocido como *Backpropagation*, en este paso vemos como se calcula el gradiente y los trucos que se utilizan para calcular el gradiente para capas iniciales.

En todos los modelos de *Machine learning* puede haber problemas de sobreajuste (*Overfitting*), en las redes neuronales vemos algunas de las técnicas utilizadas para prevenir este sobre ajuste y que los modelos puedan generalizarse mejor. Una vez aprendida la estructura básica observamos como una imagen es tratada por un modelo *convolutional* y los distintos *frameworks* que existen para facilitar la programación de las redes neuronales.

Para demostrar todos los conceptos teóricos se aplican las redes neuronales a distintos tipos de problemas como: regresión, clasificación, clasificación de dígitos y observamos la precisión que alcanzan estas en dichas tareas.

2.- Introducción

La temática de este trabajo fin de grado se basa en estudiar las nociones básicas de las redes neuronales, como predicen y como se entrenan. Primero se definen los conceptos teóricos para familiarizarse con las redes neuronales y después se demuestra su funcionamiento con algunos ejemplos.

En los conceptos teóricos se comienza primero explicando lo que es *Machine Learning* y sus distintas ramas. Una vez comprendido *Machine Learning* se explica la historia de las redes neuronales. Después de comprender su surgimiento, mediante formulación matemática se verá el *Forward propagation* que es la manera en la que una red neuronal predice una vez que introducimos los inputs. Después de entender como una red neuronal hace sus operaciones para predecir se estudiará el algoritmo utilizado para entrenar los parámetros de una red neuronal: *Backpropagation*.

En la parte de entrenamiento se explicará el algoritmo *Gradient-Descent* y distintas funciones de activación que se utilizan para introducir la no-linealidad en las redes neuronales y para finalizar la parte teórica se comentarán distintas arquitecturas de las redes neuronales y sus aplicaciones, también se comentarán distintos *frameworks* existentes para programar las redes.

En la parte práctica utilizando el lenguaje de programación Python se verán distintos ejemplos programados mediante las redes neuronales: regresión, clasificación.

3.- Marco Teórico

3.1 Fundamentos de Machine Learning

A lo largo de la historia, los humanos han sentido la curiosidad para saber cómo funciona la inteligencia humana e intentar imitar dicha inteligencia de alguna manera como por ejemplo creando robots.

A esta rama que busca automatizar los trabajos inteligentes realizados por los humanos se le conoce como *Inteligencia Artificial(AI).* Esta rama tiene distintos subcampos y *Machine learning* es uno de los subcampos de esta rama. *Machine learning* consiste en que un modelo aprenda patrones de una dataset sin tener que programarlo es decir sin tener que decir las reglas. Por ejemplo, en la tarea de clasificar flores, proporcionado suficientes datos a un modelo, este aprende a distinguir distintas clases de flores sin tener que programar explícitamente que si su color es tal y tiene tantos pétalos entonces es de esta clase sino de otra.

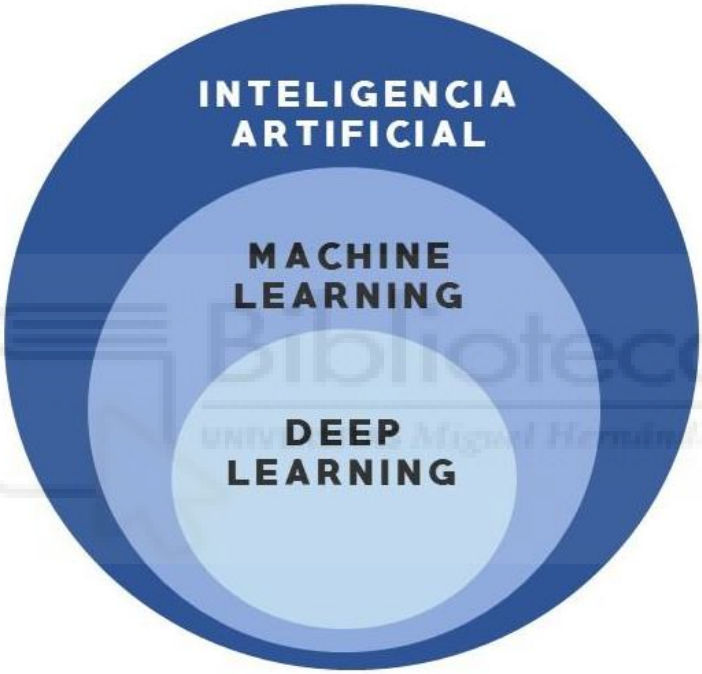


Figura 1.Distintos subcampos de la Inteligencia Artificial

Supervised Learning :

Machine Learning tiene distintos enfoques y *supervised learning* es uno de ellos, en este tipo de aprendizaje al modelo se le proporcionan inputs(ejemplos) y outputs (salidas) y mediante entrenamiento se busca una función que represente mejor la relación entre input y output. Una vez que el modelo es entrenado se utiliza la función para predecir nuevos datos.

Como se ilustra en la figura 2 donde input es la edad de una persona y output es su altura y la relación entre ellos es representada mediante una línea, este tipo de modelo *supervised* es conocido como la regresión lineal.

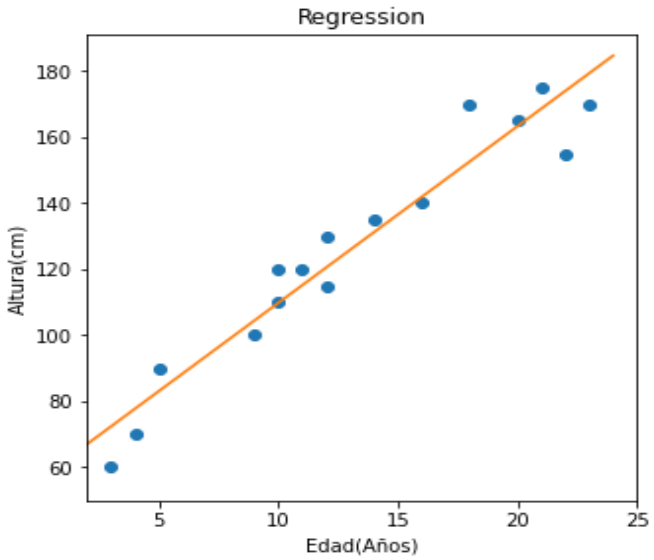


Figura 2.Supervised Linear Regression

Después de obtener el mejor modelo (la mejor recta), se utiliza dicha función para predecir nuevo input, un input de 10 años en este modelo tendría un output de 109 cm como se muestra en la figura 3.

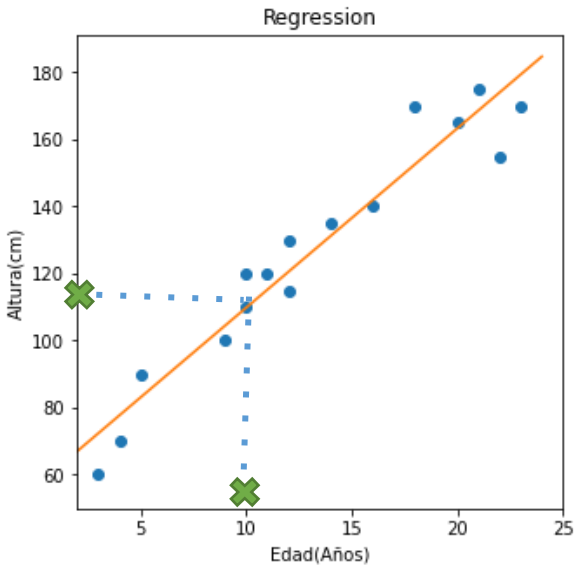


Figura 3.Predicción con el modelo [2]

Unsupervised Learning :

En *Supervised Learning* al modelo se le proporcionan inputs y outputs para que la relación entre dos sea representada por una función, pero en *Unsupervised learning* no tenemos los outputs solo conocemos los inputs por lo tanto en este aprendizaje el modelo busca los patrones escondidos (similitudes) en los inputs.

Un ejemplo de este tipo de aprendizaje es *clustering* (figura 4) donde el modelo agrupa en grupos los inputs, los elementos en un grupo son similares y los grupos entre sí son distintos.

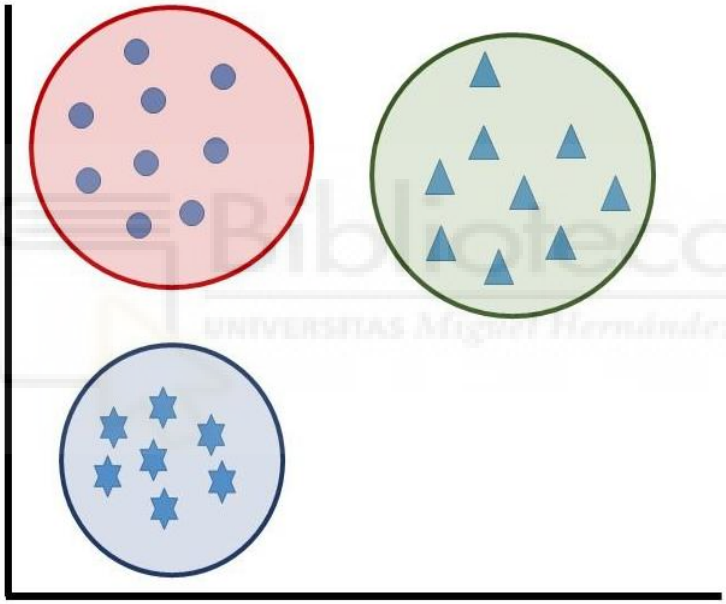


Figura 4.Unsupervised Learning Clustering

Reinforcement Learning :

En este tipo de aprendizaje, el modelo(agente) aprende a base de recompensa. Es decir, al modelo se le presenta una situación(entorno) y entre varias decisiones que puede tomar si este toma la decisión adecuada se le recompensa para que siga aprendiendo por ese camino. Un ejemplo de este tipo de aprendizaje se ilustra en la figura 5:

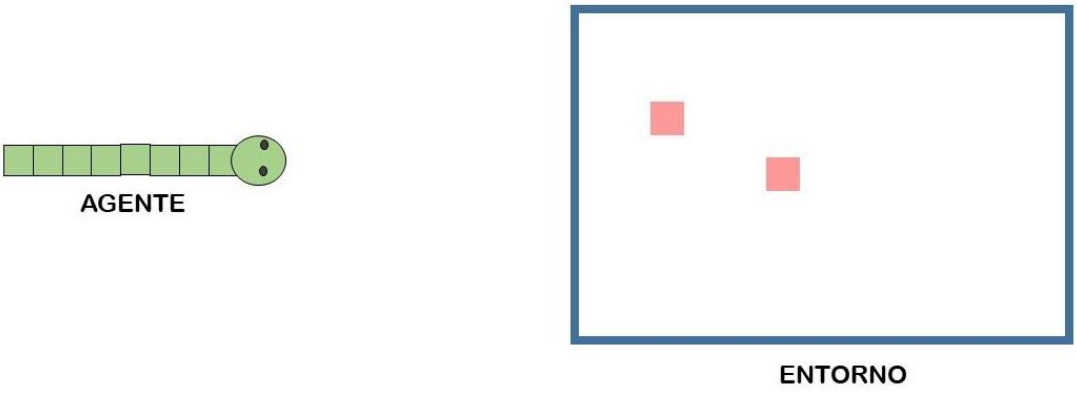


Figura 5. Ejemplo Reinforcement Learning

Todos hemos jugado el juego de serpiente y en el *reinforcement learning* se puede enseñar al agente (serpiente) por sí sola a jugar el juego. En este aprendizaje al agente se le pide tomar decisiones observando su entorno, en este caso mover arriba, abajo, derecha o izquierda si el agente toma la decisión correcta y mueve sobre los cuadros rosados se le recompensa con una puntuación positiva ya que al comer esos cuadros su tamaño aumenta y si el agente toma la decisión de mover hacía el cuadro azul entonces se le penaliza para que en la próxima iteración no cometa ese error.

3.2 Introducción a las Redes Neuronales

En el apartado anterior hemos comprendido los fundamentos básicos de *Machine Learning* y sus distintas ramas, las redes neuronales que hoy en día están en auge pertenecen al campo de *Deep learning* que a la vez es un subcampo de *Machine learning*. La arquitectura de las redes neuronales que vamos a estudiar es conocida como *Multilayer Perceptron* (MLP, *Fully Connected*) que utiliza *Supervised learning* para entrenar su modelo.

Las redes neuronales de las que se habla mucho hoy en día fueron programadas por primera vez en los años 50s por Frank Rosenblatt [1], que se inspiró de trabajos realizados sobre el funcionamiento del cerebro por personas como Santiago Ramón y Cajal para crear su primera red neuronal llamada *Perceptron*.

En la figura 6 se muestra el dibujo creado por Santiago Ramón y Cajal del cerebro de una paloma donde se observan varias neuronas biológicas.

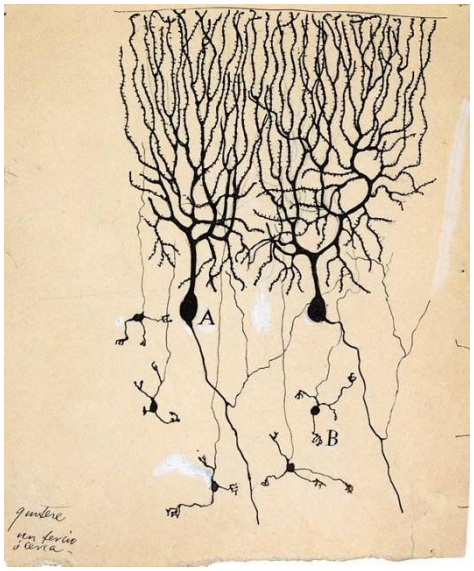


Figura 6.Dibujo del cerebro de una paloma por Santiago Ramón y Cajal (Fuente: NAUKAS.com)

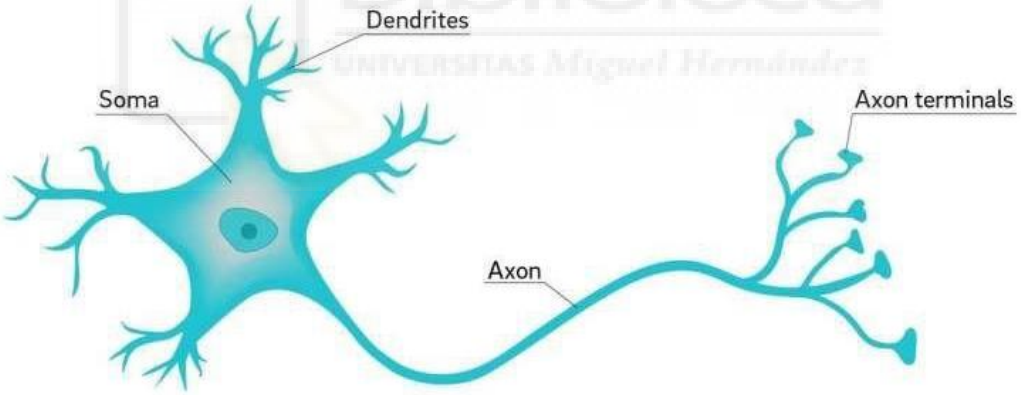


Figura 7.Una Neurona Biológica (Fuente: medium.com)

En la figura 7 observamos una neurona biológica donde la información que nuestros ojos, orejas perciben es llevada hasta la Soma mediante Dendritas. Soma se encarga de procesar la información recibida y si dicha información es suficiente para activar la Soma entonces esa dispara impulsos eléctricos que luego mediante Axon viajan hacia otras neuronas. Nuestro cerebro es formado

por 86 mil millones de unidades como la que se muestran arriba en el gráfico y todas repiten el mismo proceso hasta tomar alguna decisión (como andar, hablar etc)

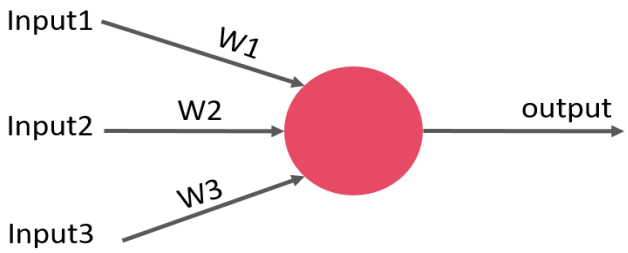


Figura 8. Perceptron

Inspirándose de una neurona biológica (figura 7) se creó el *Perceptron* (figura 8) donde la Neurona Artificial (color rosado) se puede ver como Soma y los W que son los pesos (parámetros) del modelo se pueden ver como Dendritas que llevan la información observada (Inputs) hasta la Soma. Output(Axon) es lo que sale de la Soma y que es transferido hacia otras neuronas o utilizada para tomar decisiones.

Con la formulación del *Perceptron* se genera un gran interés en esta rama pero en 1969 publicación de Minsky y Papert demuestran que el *Perceptron* es solo una función lineal $(f(x): x * w + b)$ y esa publicación hace estancar la investigación en las redes neuronales.

Perceptron : Una función lineal

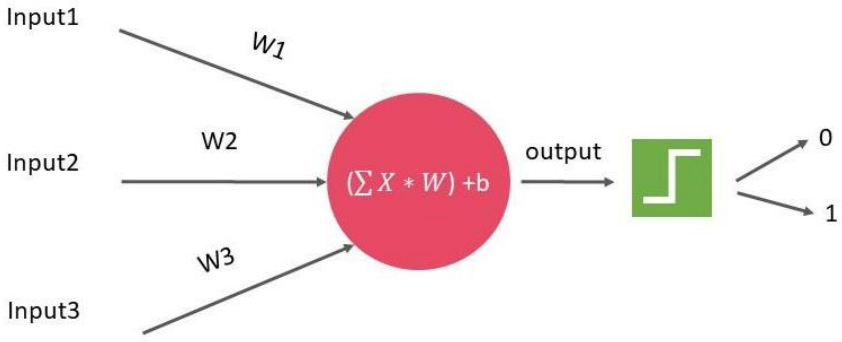


Figura 9. Perceptron para clasificación

En las tareas de clasificar *perceptron*, después de realizar la suma ponderada de inputs(x) con los pesos(w) y sumar *bias* $x * w + b$, pasa la salida (output) por una función de paso donde los outputs son redondeados a 0 o a 1. Como observamos que $x * w + b$ es una ecuación lineal y solo puede separar las clases mediante una línea, finalmente el *perceptron* puede clasificar bien el *dataset* que se observa en la figura 10. Para clasificar el *dataset* de la figura 11 necesitamos algo más porque ese *dataset* no se puede separar usando una línea.

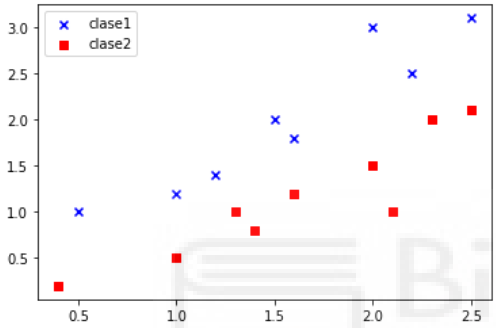


Figura 10.Dataset Clasificable

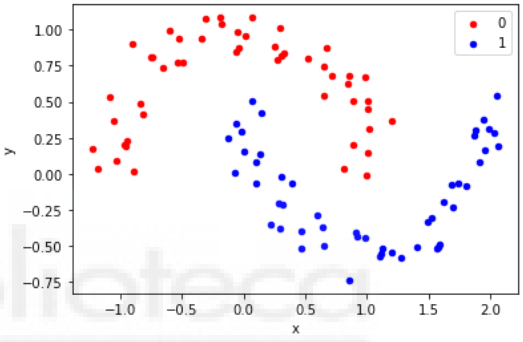


Figura 11.Dataset no clasificable

Minsky y Papert demostraron que el *perceptron* es solo una función lineal por lo tanto se estancó la investigación sobre las redes neuronales. Pero afirmaron que los *perceptrons* organizados en multicapas podrían solucionar el problema de la no linealidad, también enunciaron que estos últimos eran imposibles de entrenar.

Multilayer Perceptrons (MLP)

Multilayer Perceptrons o *Fully Connected layers* consiste en organizar varios *perceptrons* en capas que son interconectadas entre sí como se muestra en la figura 12. La primera capa que recibe los datos observados se llama Input layer y la capa final que predice se llama Output layer, las capas intermedias se llaman Hidden Layers. Se llaman *Fully Connected* también porque cada neurona de la capa anterior tiene conexiones con todas las neuronas de la capa siguiente.

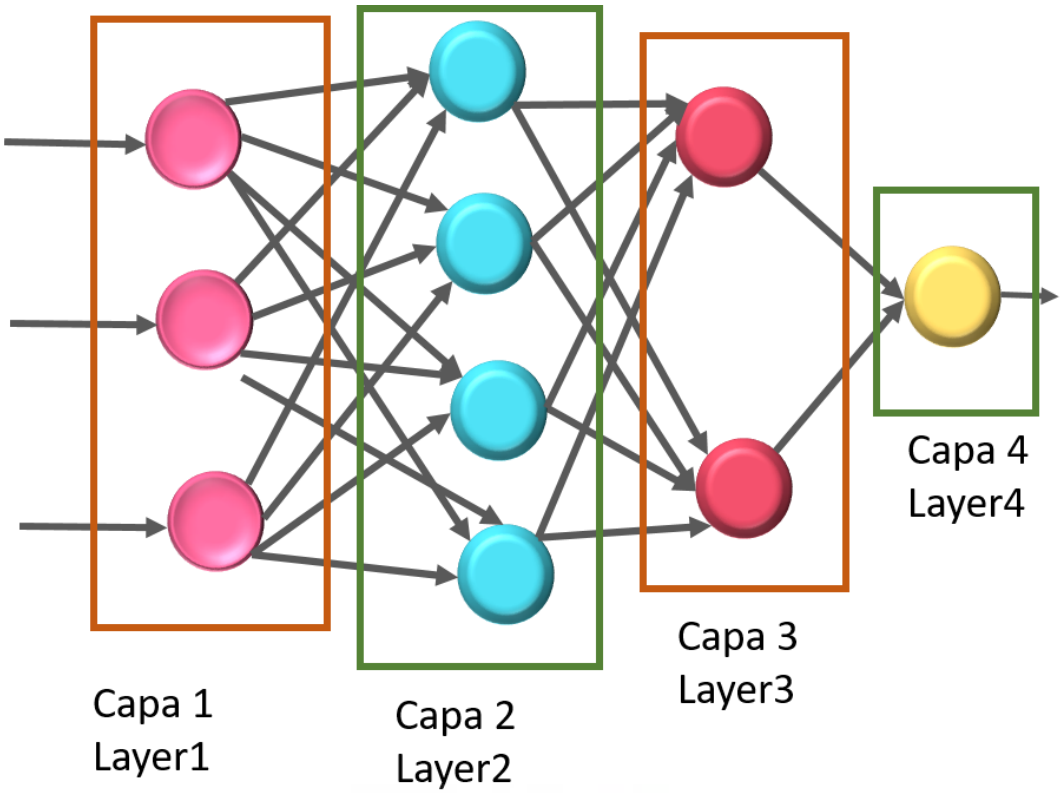


Figura 12.Multilayer Perceptrons (MLP)

Esta es la arquitectura esencial de las redes neuronales y en los próximos apartados vamos a ver como los datos son propagados hacia adelante (*forward propagation*) para predecir y como los parámetros son entrenados de esta arquitectura mediante *Backpropagation*.

3.3 Forward Propagation

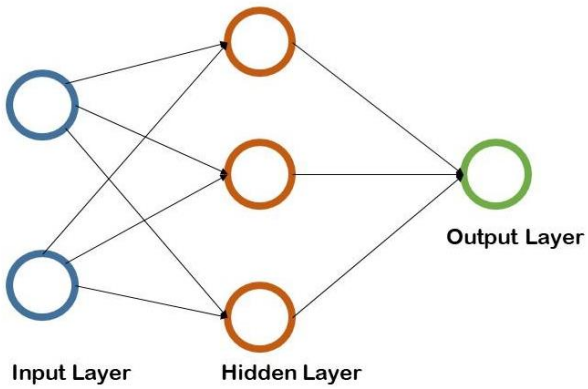


Figura 13. Red Neuronal

Para comprender cómo los datos son pasados por distintas capas de una red neuronal hasta la predicción vamos a utilizar una red neuronal que recibe dos inputs por lo tanto tiene dos neuronas(*Perceptrons*) en la capa inicial y luego en la capa hidden tiene 3 neuronas y la salida es un número por lo tanto una neurona en la capa output como se observa en la figura 13.

Por ejemplo si tenemos el siguiente *dataset* : $(x1\ x2\ )$ estos dos números serán inputs para nuestra capa inicial como se observa en la figura 14 y cada neurona en la capa hidden es conectada con las neuronas de la capa inicial mediante pesos como observamos en la figura 15 que la neurona H1 de la capa hidden se conecta con la capa inicial mediante pesos $(w1\ w2\ )$.

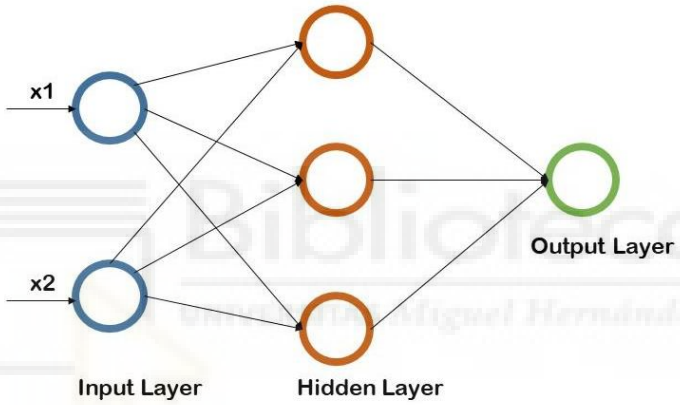


Figura 14.Input layer

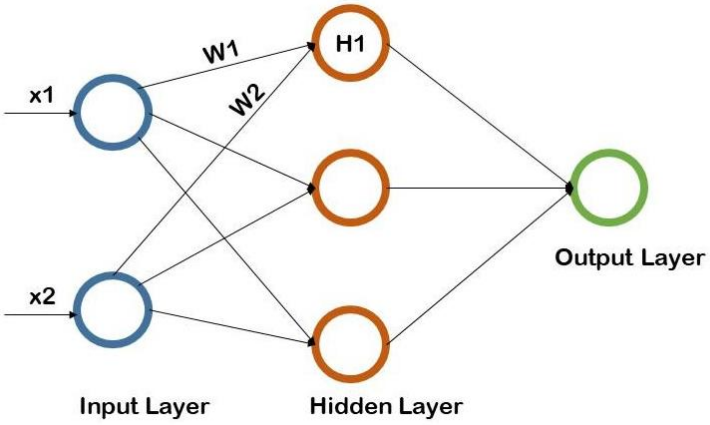


Figura 15.H1 conectada con la capa input

Cada neurona de la capa oculta se conecta con la capa inicial mediante pesos como observamos en la figura 16.

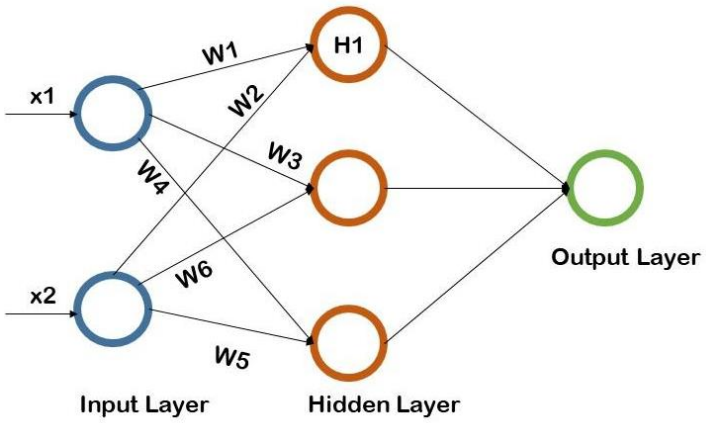


Figura 16. Capa Hidden conectada con Capa Input

Para obtener un valor para cada neurona de la capa hidden se hace una suma ponderada entre los inputs y pesos (weights) de cada capa Hidden.

$$H1 = x1 * w1 + x2 * w2$$

$$H2 = x1 * w3 + x2 * w4$$

$$H3 = x1 * w5 + x2 * w6$$

Esta suma ponderada en las redes neuronales se realiza a través de matrices donde X= es la matriz de inputs $(x1\ x2\ )$ y la W recoge los pesos que conectan la capa Inicial con la Hidden.

$$W = \begin{pmatrix} w1 & w3 & w5 \\ w2 & w4 & w6 \end{pmatrix}$$

La matrix X tiene dimensión (1,2) y la matriz W (2,3). Si realizamos la multiplicación matricial obtenemos una matriz resultante que tiene las siguientes dimensiones (1,3), es decir, un valor para cada neurona de la capa hidden. Las neuronas de la capa Hidden se conectan con la capa Output mediante pesos como observamos en la figura 17.

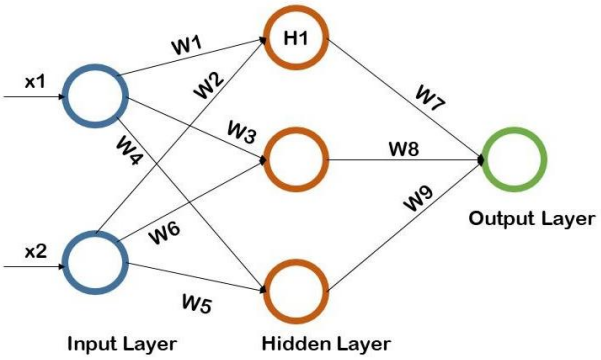


Figura 17.Capa Oculta conectada con la Capa Hidden

Para pasar los valores calculados de la capa hidden a la capa oculta se realiza una suma ponderada entre capa hidden y los pesos que la conectan con la capa Output.

$$Output = H1 * w7 + H2 * w8 + H3 * w9$$

Esta suma también se realiza de forma matricial, habíamos calculado los valores para la capa hidden que tienen la dimensión (1,3) y si multiplicamos con la matriz de pesos W2 que recoge los pesos que conectan la capa output con la Hidden y tiene dimensión (3,1) obtenemos el valor para la capa output.

$$W2 = \begin{pmatrix} w7 \\ w8 \\ w9 \end{pmatrix}$$

Para pasar los inputs desde la capa inicial hasta la output hemos realizado las siguientes multiplicaciones matriciales [3] :

(1) $$(x1 \quad x2) * \begin{pmatrix} w1 & w3 & w5 \\ w2 & w4 & w6 \end{pmatrix} = (H1 \quad H2 \quad H3)$$

(2) $$(H1 \quad H2 \quad H3) * \begin{pmatrix} w7 \\ w8 \\ w9 \end{pmatrix} = (Output)$$

(1) $X * W$
(2) $H * W2$

La multiplicación matricial para obtener output sólo transforma el espacio de manera lineal (figura 18) pero para resolver tareas complejas necesitamos la no linealidad.

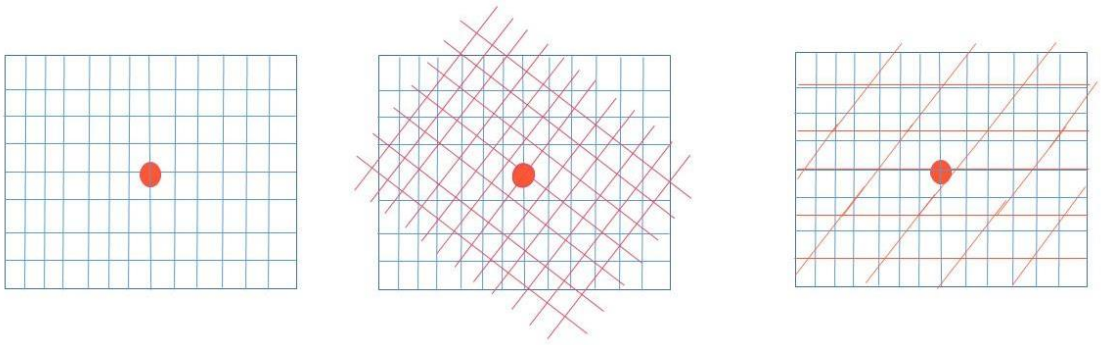
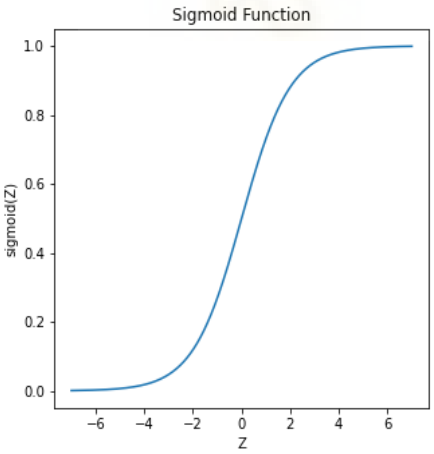


Figura 18.Transformaciones Lineales

Las transformaciones lineales solo rotan, escalan al espacio, manteniendo su estructura lineal manteniendo el origen fijo (punto naranja). Para solucionar el problema de la linealidad se introdujeron funciones de activación para las neuronas.

Las funciones de activación son funciones que introducen la no linealidad en las redes. La primera función de activación que obtuvo fama fue *sigmoid* o la función logística.



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

Figura 19. Función Sigmoid

Las funciones de activación se pueden aplicar en cada capa hidden o capa output, nunca en la Input. Por ejemplo, en nuestra red neuronal si queremos introducir la no linealidad podemos hacer que en cada neurona después de realizar la suma ponderada dicha suma sea pasada por una función de activación antes de ser transferida a la capa siguiente.

$$H1 = x1 * w1 + x2 * w2$$
$$H2 = x1 * w3 + x2 * w4$$
$$H3 = x1 * w5 + x2 * w6$$

→ Se le aplica *Sigmoid* →

$$Sigmoid(H1)$$
$$Sigmoid(H2)$$
$$Sigmoid(H3)$$

Si decidimos aplicar *sigmoid* en la capa Hidden o capa Output de nuestra red entonces las neuronas de cada capa como entrada reciben la suma ponderada, la cual representamos con $Z^{(i)}$ y como salida ofrecen $sigmoid(Z^{(i)})$ al cual le representamos con $a^{(i)}$(Figura 20).

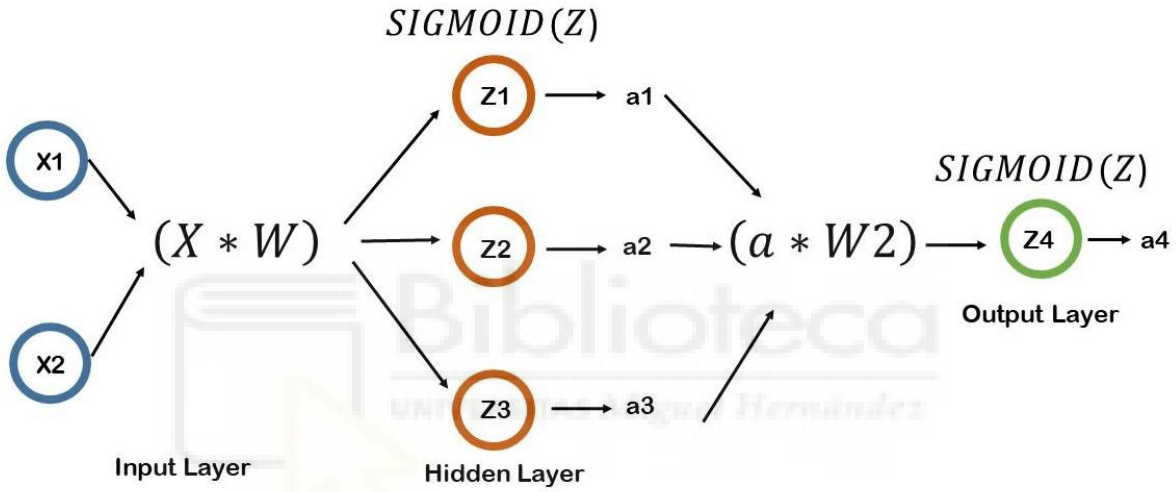


Figura 20.Forward Propagation con Sigmoid

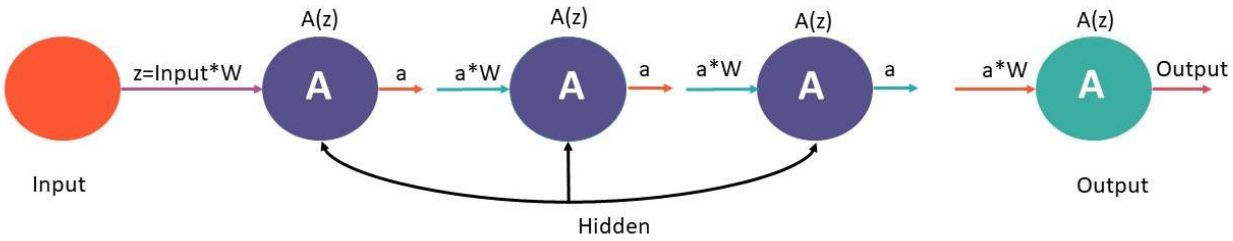


Figura 21. Forward Propagation es una función compuesta

Como observamos en la figura 21, el paso forward propagation consiste en pasar la información desde la capa inicial hasta final. La red neuronal no es nada más que una función gigante compuesta por otras funciones en la cual cada función

recibe un input desde la otra función y esa función luego convierte el input recibido en una salida para que pueda ser utilizada por la otra función. Podemos agregar las capas ocultas que queramos, el funcionamiento para pasar adelante los inputs va a ser siempre de la manera secuencial como se observa en la figura 21.

Nuestra red neuronal con la incorporación de una función de activación tendrá los siguientes pasos:

(1) $$(x1 \quad x2) * \begin{pmatrix} w1 & w3 & w5 \\ w2 & w4 & w6 \end{pmatrix} = (Z1 \quad Z2 \quad Z3)$$

(2) $$a = Sigmoid\ (Z1 \quad Z2 \quad Z3)$$ (función de activación se aplica elemento a elemento)

(3) $$(a1 \quad a2 \quad a3) * \begin{pmatrix} w7 \\ w8 \\ w9 \end{pmatrix} = (Z4)$$

(4) $$a = Sigmoid(Z4) \rightarrow Output$$

Con las funciones de activación ya podremos clasificar problemas no lineales (figura 22)

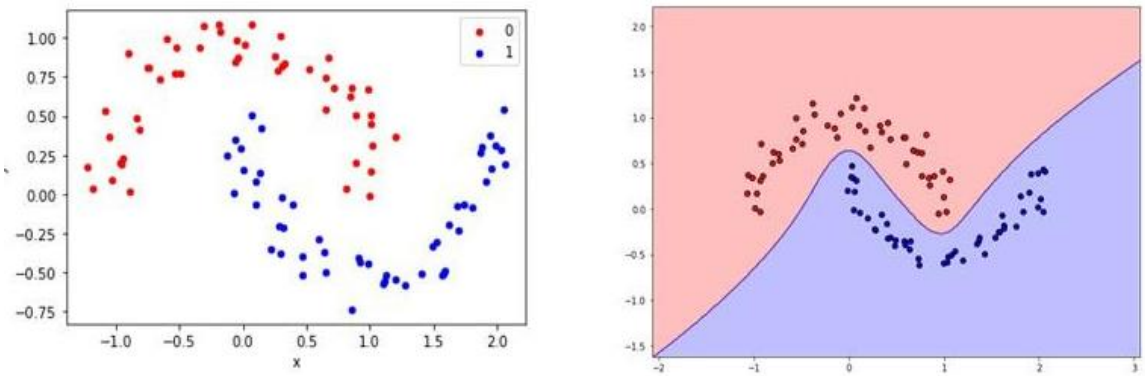


Figura 22. Non Linear data Clasificada

Forward Propagation con una red neuronal de 3 inputs, 2 capas ocultas,2 outputs y función de activación (Figura 23)

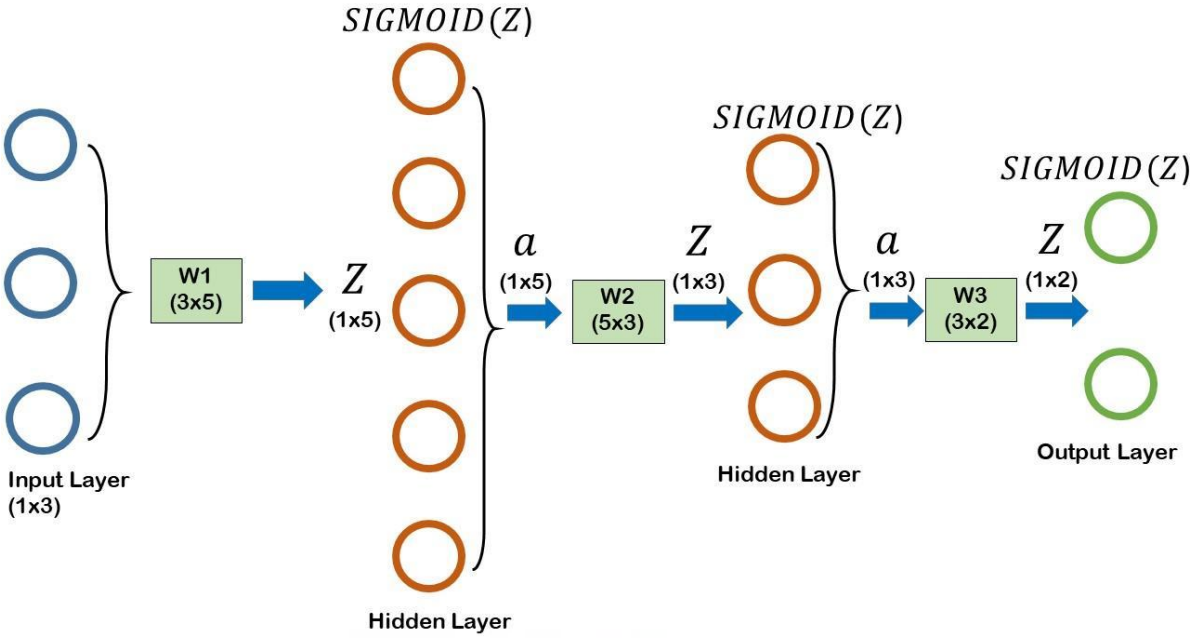


Figura 23.Forward Propagation en una red con 2 capas ocultas

Vemos que los pasos en *forward propgation* siempre se aplican de forma secuencial da igual el número de capas o las neuronas en cada capa.

(1) $X_{(1,3)} * W_{(3,5)} = Z_{(1,5)}$

(2) $a = Sigmoid\left(Z_{(1,5)}\right)$

(3) $a_{(1,5)} * W_{(5,3)} = Z_{(1,3)}$

(4) $a = Sigmoid\left(Z_{(1,3)}\right)$

(5) $a_{(1,3)} * W_{(3,2)} = Z_{(1,2)}$

(6) $a = Sigmoid\left(Z_{(1,2)}\right) \rightarrow Output$

Los W son las matrices de pesos y se inician con números aleatorios.

$$X_{(1,3)} * W_{(3,5)} \ * \rightarrow multiplicación\ matricial.$$

3.4 Loss Function y Gradient Descent

En una red neuronal los pesos (*Weights*) son iniciados aleatoriamente por lo tanto las predicciones del modelo van a ser aleatorias. Entonces necesitamos ajustar dichos pesos y debemos encontrar alguna medida que nos indique si aumentar o disminuir el valor de los pesos.

En las redes neuronales esa medida es conocida como *Loss function* o la función objetivo, es la función que nos indica qué error estamos cometiendo, porque como hemos dicho anteriormente una red neuronal utiliza un tipo de aprendizaje *supervised* entonces en *loss function* se compara el verdadero output con uno predicho por el modelo.

Loss function nos indica qué error estamos cometiendo entonces para minimizar ese error debemos minimizar el *Loss function*. Existen distintos tipos de *loss function* algunos de ellos son:

MSE = Mean Squared Error (error cuadrático medio)
MAE= Mean Absolute Error (error medio absoluto)
Cross-entropy = Entropía Cruzada

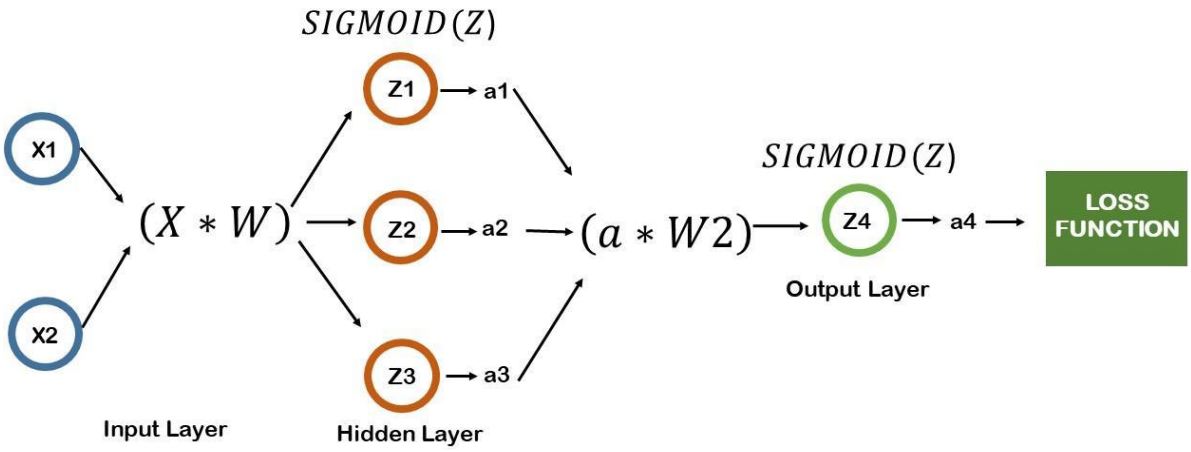


Figura 24.Red Neuronal con Loss function

Para minimizar el *loss function* tanto en otros algoritmos de *Machine learning* como en las redes neuronales se utiliza la técnica de optimización conocida como el *Gradient Descent* (Descenso del Gradiente)

Antes de aplicarlo a las redes neuronales vamos a ver en qué consiste esta técnica de optimización. *Gradient Descent* consiste en calcular las derivadas parciales. Las Derivadas nos indican una relación entre dos variables y su tasa de cambio, es decir, si varía una variable ese cambio cómo afecta a la otra variable. Por ejemplo, en la figura 25 vemos que en una función entra un input y sale un output. Si el input es 1 la función nos devuelve 2. Ahora, si aumentamos nuestro input un poco queremos saber eso cómo cambia al output, esa tasa de cambio nos la da la derivada $\frac{dy}{dx} \rightarrow \frac{d\ output}{d\ input}$.

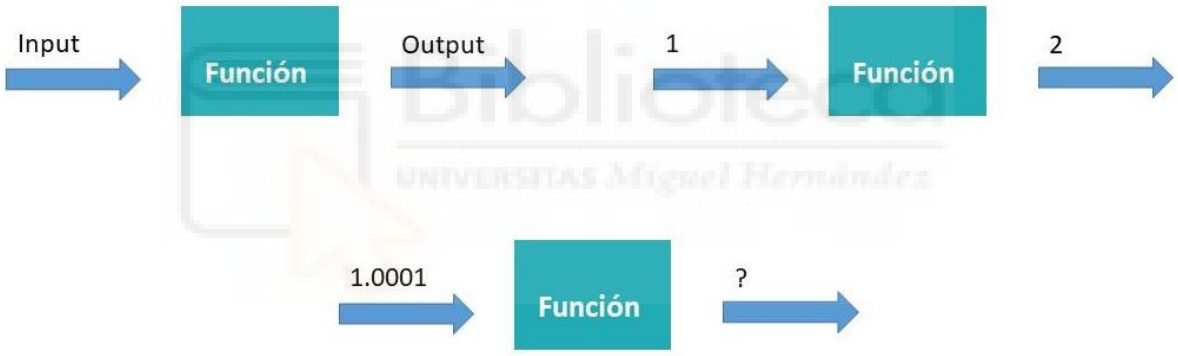


Figura 25.Derivadas muestran tasas de cambio

En nuestro modelo de redes neuronales tenemos varios parámetros (pesos). Para saber cómo cambia el error si se cambia un parámetro manteniendo los otros constantes, debemos calcular las derivadas parciales. El gradiente recoge las derivadas parciales respecto cada parámetro de la función.

$$\begin{bmatrix} \frac{\partial loss}{\partial parámetro\ 1} \\ \vdots \\ \frac{\partial loss}{\partial parámetro\ n} \end{bmatrix} \rightarrow Gradiente$$

El gradiente apunta hacía la dirección de mayor aumento, es decir, apunta hacia la dirección donde el cambio que se produce en la función es máximo. Imaginemos que la montaña (figura 26) es una función matemática y el punto azul es nuestra posición actual, queremos subir a la montaña y desde nuestra posición podemos ir en varias direcciones. Siguiendo algunas tardaremos más en llegar al siguiente nivel 200 y en otros menos. Pero antes de subir queremos saber cuál es la dirección más rápida, donde el cambio es mayor y dicho cambio es dado por el gradiente de la montaña.

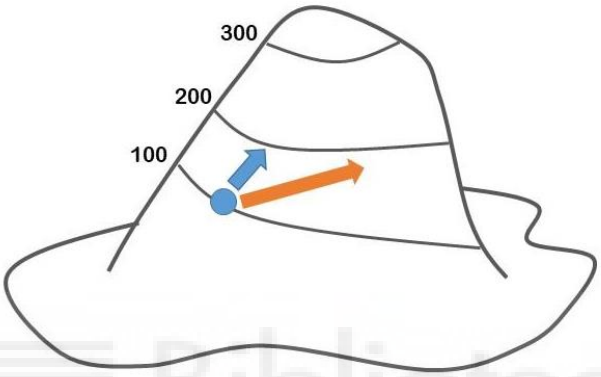


Figura 26. Gradiente apunta a la dirección de mayor aumento

En la figura 26 observamos que si cogemos la dirección azul llegamos al siguiente nivel más rápido y esa dirección es la del gradiente. Entonces si el gradiente indica la dirección de mayor aumento su negativo indicará la dirección de descenso más rápido: *Gradient Descent*.

¿Por qué la dirección de mayor aumento es en la dirección del gradiente y no de otro vector?

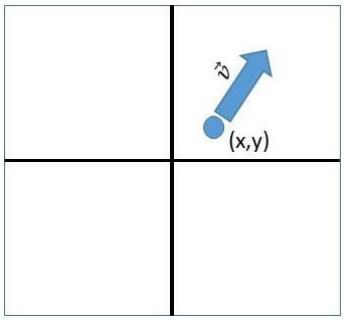


Figura 27. Derivada direccionales

Para responder la pregunta planteada debemos calcular las derivadas direccionales, derivadas en la dirección de un vector.

El punto azul (figura 27) representa nuestra posición (x,y) y el vector $\vec{v}$ representa la dirección y podemos calcular las derivadas direccionales en la dirección de $\vec{v}$ con la fórmula $D_V f(x,y) = \nabla f(x,y) \cdot \vec{v}$ donde $\nabla f(x,y)$ es el gradiente en el punto (x,y) y el $\vec{v}$ la dirección , producto escalar $\nabla f(x,y) \cdot \vec{v}$ se puede escribirse de la siguiente forma : $\|\nabla f\| \cdot \|\vec{v}\| \cdot cos(\alpha)$ donde $\|\nabla f\|, \|\vec{v}\|$ son vectores unidad, su longitud es 1.

$\|\nabla f\| \cdot \|\vec{v}\| \cdot cos(\alpha)$,esta expresión será máxima cuando el coseno sea 0 porque cos(0) = 1 , α = 0 significa que el ángulo entre el gradiente y el vector dirección es 0 lo que significa que van en la misma dirección. Por lo tanto, el mayor aumento siempre será en la dirección del gradiente y el mayor descenso en la dirección contraria.

3.5 Backpropagation

Loss function indica el error que cometemos al predecir y con el algoritmo de *Gradient descent* podemos minimizar dicho error. Ahora vamos a ver cómo se actualizan los parámetros de una red neuronal con la técnica comentada previamente.

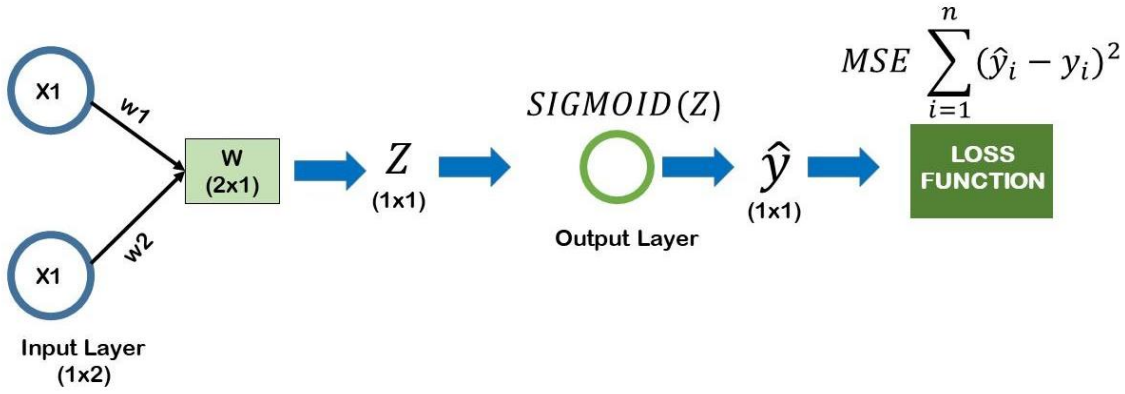


Figura 28. Red neuronal con Loss function

Vamos a optimizar los pesos w1, w2 de esta red neuronal (figura 28) que tiene en la capa final una función de activación *sigmoid* y como *loss function MSE* (error cuadrático medio).

$$\hat{y}: Valor\ predicho\ ,\qquad y: Valor\ real\ Observado$$

Esta red neuronal en paso *forward propagation* realiza las siguientes operaciones:

(1) $$X_{(1,2)} * W_{(2,1)} = Z_{(1,1)} \quad \rightarrow \quad X_{(1,2)} * W_{(2,1)} = x1 * w1 + x2 * w2$$

(2) $$\hat{y} = Sigmoid\left(Z_{(1,1)}\right)$$

(3) $$loss = MSE = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Vemos que los pesos w1, w2 no influyen directamente en el error. En *MSE* no aparecen los términos w1 y w2. Entonces, ¿cómo podemos calcular las derivadas del *loss function MSE* respecto a los pesos (parámetros) $\frac{\partial loss}{\partial W\ 1}$, $\frac{\partial loss}{\partial W\ 2}$?.

Necesitamos las derivadas para saber cómo cada peso cambia al *loss function*. Sabemos que los pesos w1,w2 no tienen influencia directa en el error, pero sí indirecta(figura 29) porque $\hat{y}$ es $Z_{(1,1)}$ después de aplicarle función sigmoid y $Z_{(1,1)}$ se obtiene al hacer producto entre $Z_{(1,1)} = x1 * w1 + x2 * w2$. Debemos utilizar esta relación para calcular las derivadas $\frac{\partial loss}{\partial W\ 1}$, $\frac{\partial loss}{\partial W\ 2}$.

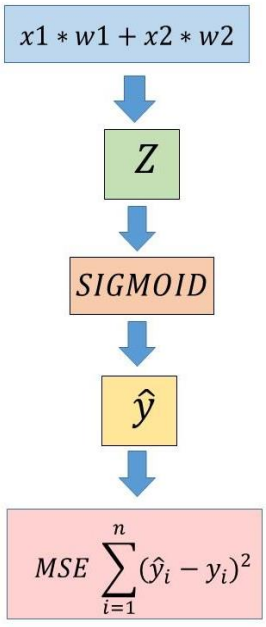


Figura 29. W1, W2 influyen de forma indirecta en loss function

Cuando una variable es relacionada con otra mediante una variable intermedia hay que aplicar la regla de cadena (*Chain Rule*) para calcular las derivadas.

$$\frac{\partial\ Agua\ consumida}{\partial\ Temperatura} = \frac{\partial\ Agua\ Consumida}{\partial\ sed} \quad x \quad \frac{\partial\ sed}{\partial\ Temperatura}$$

Figura 30. Chain rule example

Vamos a entender el funcionamiento del *chain rule* con el ejemplo de la figura 30, Vemos en un gráfico que a medida que aumenta la temperatura, sed aumenta también y en el otro gráfico a medida que aumenta sed, agua consumida también aumenta. Entonces, si queremos ver cómo cambia la consumición de agua si aumentamos la temperatura y ese cambio es dado por la derivada de $\frac{\partial\ Agua\ consumida}{\partial\ Temperatura}$. Como vemos en el gráfico consumición de agua no se relaciona directamente con la temperatura sino mediante la variable sed, por lo tanto tenemos que aplicar la regla de cadena: primero calculamos cómo la consumición de agua cambia al variar la variable sed y esa derivada $\frac{\partial\ Agua\ Consumida}{\partial\ sed}$ la multiplicamos por la derivada que nos indica cómo sed cambia al variar la variable temperatura $\frac{\partial\ sed}{\partial\ Temperatura}$. Finalmente, al multiplicar, obtenemos como la consumición de agua cambia al variar la variable temperatura (figura 30).

¿Por qué se multiplican? Por ejemplo, si al aumentar la temperatura en 1 unidad sed aumenta en 0.5 unidades y al aumentar la sed en 1 unidad consumición

de agua aumenta en 2 unidades. Si queremos saber cómo cambia la consumición de agua al aumentar 1 unidad de temperatura entonces primero vemos que si cambiamos una unidad de la temperatura sed aumenta en 0.5 unidades. Sabemos que un aumento de 1 unidad de sed aumenta la consumición en 2 unidades. Entonces, un aumento en 0.5 unidades de sed harán aumentar la consumición en 1 unidad y así calculamos cómo la consumición de agua varía al hacer cambios en la temperatura: primero vemos cómo el cambio en temperatura hace variar la variable sed y después cómo esa variación en sed cambia la consumición utilizando la regla de cadena:

$$\frac{\partial\ Agua\ consumida}{\partial\ Temperatura} = \frac{\partial\ Agua\ Consumida}{\partial\ sed} \quad x \quad \frac{\partial\ sed}{\partial\ Temperatura}$$

$$\frac{\partial\ Agua\ consumida}{\partial\ Temperatura} = 0.5 \quad x \quad 2 = 1$$

Volviendo a nuestro ejemplo (figura 29), ahora si queremos saber cómo un parámetro por ejemplo el peso w1 hace variar el error (*Loss function*) debemos aplicar la regla de cadena porque se observa una relación indirecta entre peso w1 y el error.

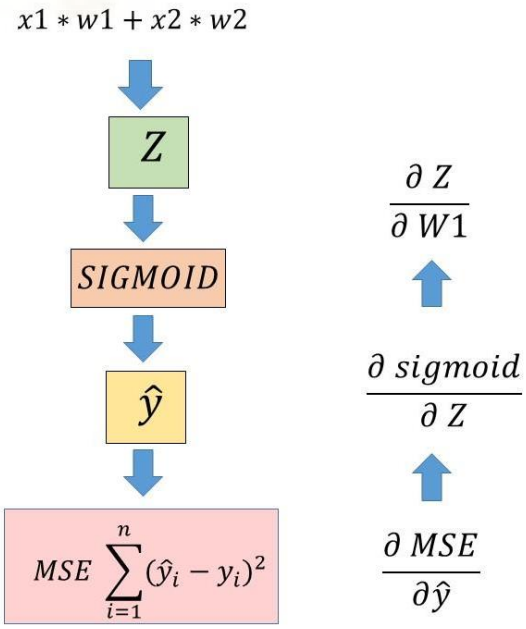


Figura 31. Backpropagation: calculando derivadas

(figura 31) Para ver cómo el peso w1 cambia al *loss function* (*MSE*) debemos aplicar la regla de cadena, porque primero debemos calcular cómo la predicción $\hat{y}$ cambia al *loss function MSE* y después cómo vemos que la predicción $\hat{y}$ se obtiene al aplicar *Sigmoid* a Z. A continuación, debemos calcular cómo *Sigmoid* cambia si variamos Z que es su entrada. En el paso final, vemos que Z se obtiene al hacer $x1 * w1 + x2 * w2$. Por tanto calculamos como Z cambia si variamos w1. Para obtener el cambio en MSE respecto al peso w1 se aplica *chain rule*:

$$\frac{\partial\, MSE}{\partial w1} = \frac{\partial\, MSE}{\partial \hat{y}} \times \frac{\partial\, sigmoid}{\partial\, Z} \times \frac{\partial\, Z}{\partial\, w1}$$

Se realiza el mismo procedimiento para calcular el cambio en *MSE* respecto al peso w2.

$$\frac{\partial\, MSE}{\partial w2} = \frac{\partial\, MSE}{\partial \hat{y}} \times \frac{\partial\, sigmoid}{\partial\, Z} \times \frac{\partial\, Z}{\partial\, w2}$$

El vector que recoge ambas derivadas se llama el gradiente de *Loss function* respecto a los parámetros w1, w2. $\begin{bmatrix} \frac{\partial\, MSE}{\partial w1} \\ \frac{\partial\, MSE}{\partial w2} \end{bmatrix}$

Para optimizar los parámetros (pesos) se resta el gradiente a los pesos actuales. Se resta porque el gradiente apunta hacía la dirección de mayor aumento, pero nosotros en este caso queremos minimizar. Por lo tanto, debemos ir en la dirección contraria. Los pesos w1, w2 se inicializan con valores aleatorios y por lo tanto al inicio producirán mayor error lo que equivale en mayor valor de *Loss function*. Pero en cada iteración los pesos se van actualizando, restándoles el gradiente.

$$Nuevo\ w1 = w1 - \frac{\partial\, MSE}{\partial w1}$$

$$Nuevo\ w2 = w2 - \frac{\partial\ MSE}{\partial w1}$$

Pero hay un parámetro que se añade más en la actualización de pesos y es conocido como el *learning rate*.

Si sólo cogemos el gradiente para actualizar los parámetros el problema que surge es que la magnitud del gradiente puede ser tan elevada que el problema nunca converge. Converger es encontrar la solución óptima. Por lo tanto, de alguna manera debemos movernos en la dirección opuesta del gradiente pero con garantía de que vamos a llegar al óptimo.

El problema de magnitud del gradiente sucede si por ejemplo, en nuestra red neuronal tenemos inputs y pesos que son números grandes. Entonces al calcular las derivadas el gradiente también tendrá números grandes y si los pesos requieren actualizaciones pequeñas, los gradientes con números grandes harán cambios grandes que en vez de mejorar el resultado lo van a empeorar más. Por lo tanto, en estos casos debemos introducir otro parámetro que controle el tamaño de los pasos que debemos dar en la dirección opuesta del gradiente. A ese nuevo parámetro le llamaremos: *Learning rate*.

Figura 32. Learning rate controla los pasos

Como observamos en la figura 32, el círculo rojo es el óptimo de nuestra función, donde el error es mínimo; y el círculo azul nuestra posición inicial. Para minimizar el error en cada iteración, calculamos el gradiente y restamos dicho gradiente a los parámetros(pesos) para actualizarlos. Para controlar la magnitud que restamos a los parámetros se usa *learning rate*.

Debemos elegir *Learning rate* de tal manera que ni sea muy pequeño y que ni sea muy grande: porque pasos pequeños pueden tardar mucho para converger (figura32.dibujo 1) mientras que los pasos grandes pueden hacer que estemos rebotando acerca del óptimo (figura32.dibujo2).

Gradient Descent con Learning rate :

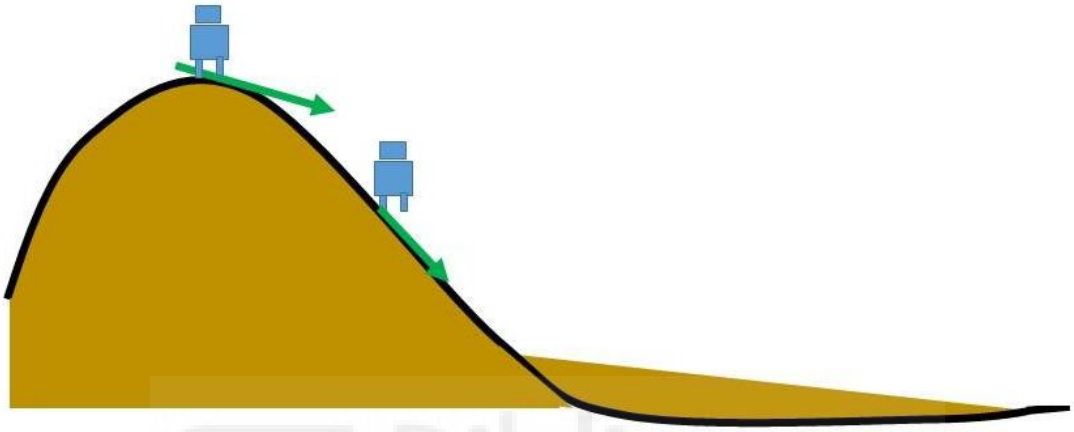


Figura 33.Gradient descent con Learning Rate

El algoritmo de optimización *Gradient descent* junto con *learning rate* se puede entender como que tenemos un robot que está situado sobre una montaña (figura 33) y lleva una cámara incorporada que solo puede observar el entorno que rodea al robot, pero solo hasta 100 metros. Nosotros tenemos el mando que controla al robot. Queremos hacer descender al robot, ¿cómo lo haríamos?

Primero observaríamos nuestro entorno y elegiríamos la dirección más inclinada (opuesto del Gradiente) luego daríamos unos pasos hacía dicha dirección (*learning rate*) y volveríamos evaluar de nuevo la situación (repetir el algoritmo) hasta llegar al punto óptimo en nuestro caso es descender la montaña.

Entonces los pesos con *learning rate* (α) se actualizan de la siguiente manera:

$$Nuevo\ w1 = w1 - \left(\alpha \times \frac{\partial\ MSE}{\partial w1}\right)$$

$$Nuevo\ w2 = w2 - \left(\alpha \times \frac{\partial\ MSE}{\partial w2}\right)$$

En las redes neuronales hacemos *forward propagation*, que es hacer la predicción utilizando multiplicación matricial. Para el paso de *backpropagation* que consiste en calcular gradientes para entrenar las redes neuronales en la forma matricial sería:

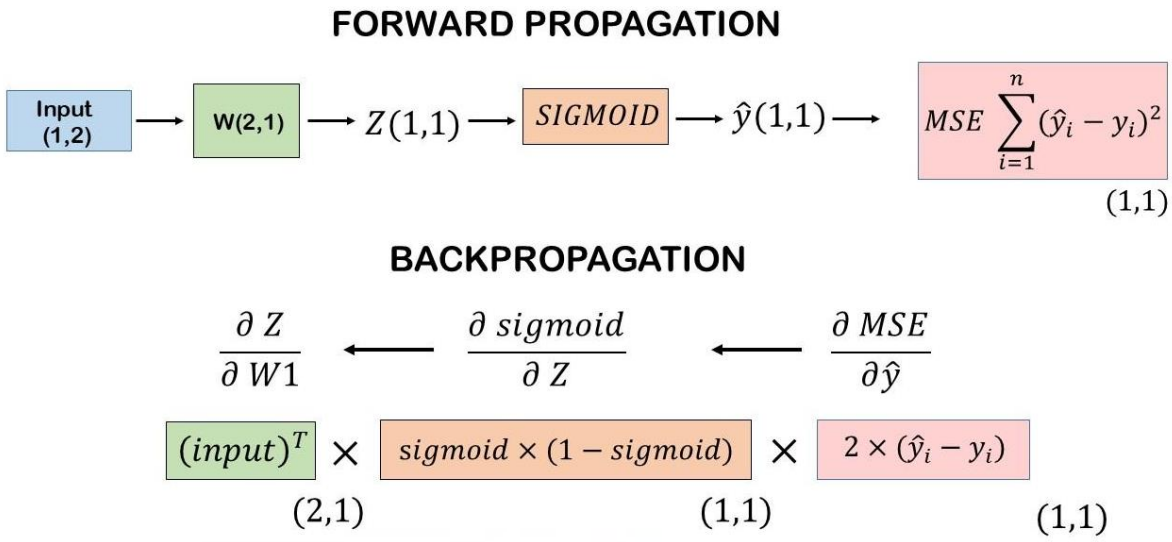


Figura 34. Forward + Backpropagation

En el paso forward los pesos son una matriz de dimensión 2x1. Para actualizar esa matriz necesitamos restarle otra matriz 2x1 formada por los gradientes, podemos calcular por separado cada componente del gradiente :

$$\frac{\partial\, MSE}{\partial w1} = \frac{\partial\, MSE}{\partial \hat{y}} \times \frac{\partial\, sigmoid}{\partial\, Z} \times \frac{\partial\, Z}{\partial\, w1}$$

o utilizar el algoritmo de *backpropagation* junto con las matrices. Por ejemplo, en la figura 34 en el paso *backpropagation* la derivada de *MSE* respecto al $\hat{y}$ nos devuelve una matriz de (1x1) que es el error. Ese error luego se multiplica por la derivada de *sigmoid*. Como solo había un Z, la derivada de *sigmoid* también es (1x1) y al multiplicar elemento a elemento $\frac{\partial\, MSE}{\partial \hat{y}} \times \frac{\partial\, sigmoid}{\partial\, Z}$ obtenemos una matriz (1x1). Estos dos pasos son idénticos en $\frac{\partial\, MSE}{\partial w1}$, $\frac{\partial\, MSE}{\partial w2}$. El paso que marca la diferencia en la regla de cadena es $\frac{\partial\, Z}{\partial\, w1}$. Hay que derivar Z respecto distintos

parámetros y sabemos que Z es $x1 * w1 + x2 * w2$. Por lo tanto, $\frac{\partial Z}{\partial w1}$ es x1 y $\frac{\partial Z}{\partial w2}$ es x2, o sea, los inputs.

Pero los inputs como vemos en la figura 34 son una matriz (1x2) y no podemos multiplicar dicha matriz con la matriz que nos sale al multiplicar los pasos comunes, $\frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial sigmoid}{\partial Z}$ y cuya dimensión es (1x1). Pero si transponemos la matriz de inputs obtenemos una dimensión (2x1) y esa multiplicada con la de pasos comunes (1x1) nos devuelve una matriz (2x1) y esa es la matriz gradiente que necesitamos restar a la matriz de pesos para actualizar sus valores. Se puede comprobar que al transponer la matriz y hacer las multiplicaciones matriciales obtenemos la expresión que nos da la regla de cadena:

$$\begin{pmatrix} \frac{\partial Z}{\partial w1} \\ \frac{\partial Z}{\partial w2} \end{pmatrix} \times \left(\frac{\partial sigmoid}{\partial Z}\right) \times \left(\frac{\partial MSE}{\partial \hat{y}}\right) = \begin{pmatrix} \frac{\partial Z}{\partial w1} \times \frac{\partial sigmoid}{\partial Z} \times \frac{\partial MSE}{\partial \hat{y}} \\ \frac{\partial Z}{\partial w2} \times \frac{\partial sigmoid}{\partial Z} \times \frac{\partial MSE}{\partial \hat{y}} \end{pmatrix}$$

Con la multiplicación matricial los pesos se actualizan de la siguiente manera:

$$\begin{pmatrix} w1 \\ w2 \end{pmatrix} = \begin{pmatrix} w1 \\ w2 \end{pmatrix} - \alpha * \begin{pmatrix} \frac{\partial Z}{\partial w1} \times \frac{\partial sigmoid}{\partial Z} \times \frac{\partial MSE}{\partial \hat{y}} \\ \frac{\partial Z}{\partial w2} \times \frac{\partial sigmoid}{\partial Z} \times \frac{\partial MSE}{\partial \hat{y}} \end{pmatrix}$$

Backpropagation con varias capas hidden (ocultas):

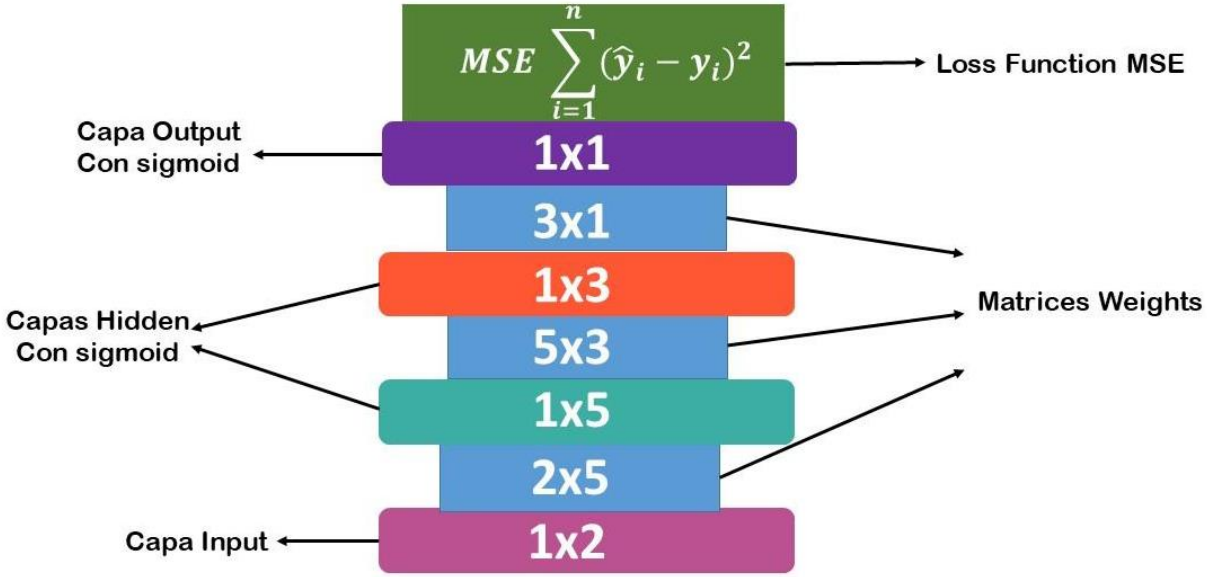


Figura 35. Red neuronal con dos capas Hidden

En la figura 35 podemos representar una red neuronal como bloques de lego que van encajando según las dimensiones, por ejemplo, los bloques azules son las matrices de pesos que conectan las capas anteriores con las siguientes. Las capas hidden y output tienen función de activación *sigmoid* y el bloque final es el de *Loss function*.

Cada bloque es una matriz, por lo tanto, los números inscritos encima de él son sus dimensiones la capa output tiene una dimensión 1x1 es decir output es un solo número y si hacemos las multiplicaciones matriciales desde abajo hasta arriba de forma secuencial (*forward propagation*) obtendremos como resultado una matriz 1x1.

Tiene dos capas ocultas donde una capa tiene 5 neuronas y otra 3, por eso en la figura 35 la capa hidden es representada mediante una matriz 1x5 y 1x3

porque en cada neurona entra un valor. En la figura 36 se muestra la misma red neuronal con la representación que vemos usualmente.

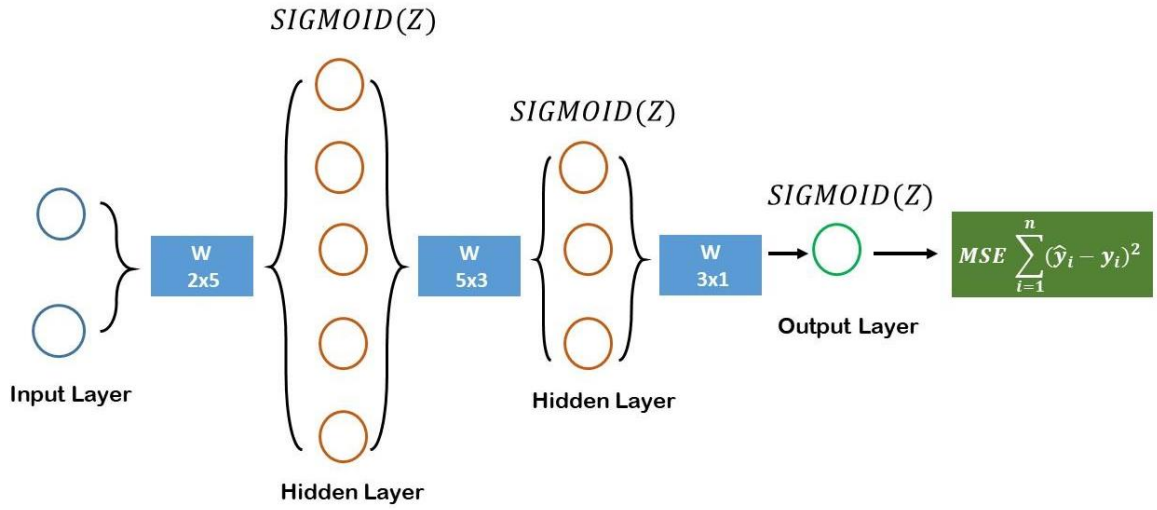


Figura 36. Red neuronal con 2 capas Hidden

En el paso de predicción, *forward propagation* consiste en multiplicar las matrices de forma secuencial mostradas en la figura 35 aplicando en la matriz función de activación si una capa la tiene.

Pasos:

(1) $\begin{pmatrix} x1 \\ x2 \end{pmatrix} \times (Wih1)_{2\times5} = (Zh1)_{1\times5}$

(2) $ah1 = Sigmoid\ (Zh1)_{1\times5}$

(3) $(ah1)_{1\times5} \times (Wh1h2)_{5\times3} = (Zh2)_{1\times3}$

(4) $ah2 = Sigmoid\ (Zh2)_{1\times3}$

(5) $(ah2)_{1\times3} \times (Wh2O)_{3\times1} = (ZO)_{1\times1}$

(6) $\hat{y} = Sigmoid\ (ZO)_{1\times1}$

(7) $MSE\ \sum_{i=1}^{n}(\hat{y}_i - y_i)^2$

La función de activación se aplica a cada elemento de Z.

$(Wih1)_{1\times5}$ → Matriz de pesos entre la capa input y Primera hidden

$ah1$ → Valores de hidden 1 después de activarles *sigmoid*

$(Wh1h2)_{5\times3}$ → Matriz de pesos entre la capa Primera hidden y Segunda hidden

$ah2$ → Valores de hidden 2 después de activarles *sigmoid*

$(Wh2O)_{3\times1}$ → Matriz de pesos entre la capa Segunda Hidden y Output

$\hat{y}$ → Valor predicho (output)

Arriba vemos los pasos desde input hasta output y vemos que tenemos 3 matrices de pesos (*weights*) que al inicio se inicializan de forma aleatoria y que con *backpropagation* debemos optimizar sus valores.

Aquí observamos como en el caso anterior que dichos *weights* no tienen influencia directa sobre el error calculado en el *Loss function*. Por lo tanto, debemos usar la regla de cadena para calcular los gradientes para cada matriz de parámetros (pesos) presente en la red neuronal.

Cuando hay varias capas la estrategia que se utiliza en *backpropagation* es calcular un error para cada capa, sin incluir la capa inicial que es de inputs. A ese error que se calcula para cada capa se le llama delta δ. Se utilizan deltas porque aplicar regla de cadena en las redes con muchas capas es hacer muchas multiplicaciones de derivadas, en las que unas partes son comunes.

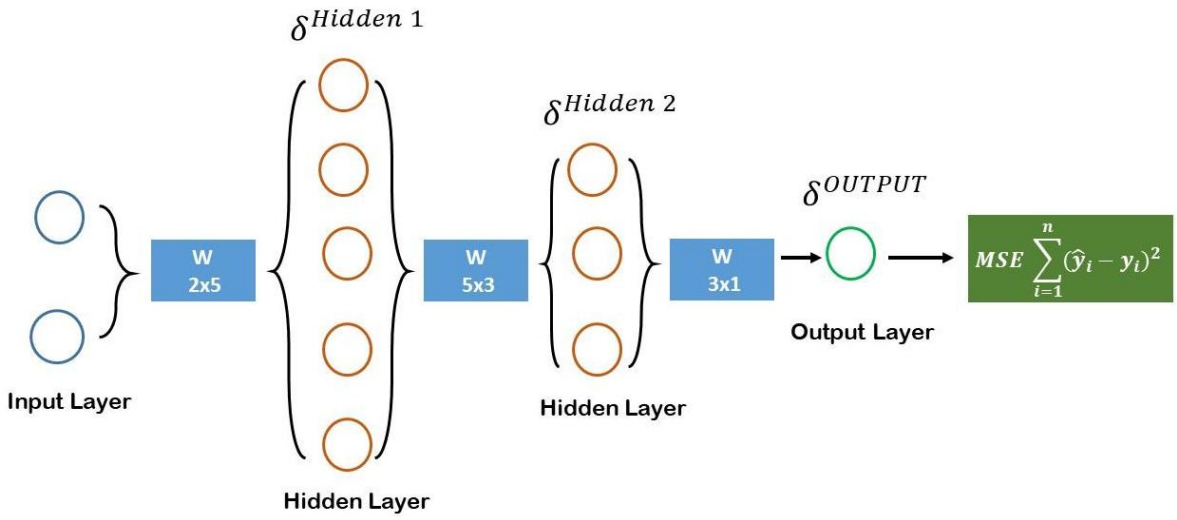


Figura 37. Backpropagation: utilizando deltas como error de cada capa

En la figura 37 vemos que tenemos 3 deltas y estas deltas se utilizarán para actualizar los pesos. Empezando por δ^{OUTPUT} esta capa es la de output y esta capa como entrada tiene $(ZO)_{1\times1}$ y como salida $\hat{y}$. Entonces, para ver cómo esta capa contribuye en el error debemos ver cómo su salida cambia al error y su salida depende de la entrada $(ZO)_{1\times1}$.

$$\delta^{OUTPUT} = \frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial ZO}$$

δ^{OUTPUT} se obtiene al ver cómo *MSE* cambia al variar $\hat{y}$, y cómo $\hat{y}$ cambia al variar ZO. Ahora para la siguiente capa Hidden 2 vemos que como entrada tiene $(Zh2)_{1\times3}$ y como salida $ah2$. Para ver cómo su salida $ah2$ influye en el error debemos utilizar la regla de cadena porque $ah2$ no está relacionada directamente con el *MSE*.

$$\frac{\partial MSE}{\partial ah2} = \frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial ZO} \times \frac{\partial ZO}{\partial ah2}$$

Observamos que $ah2$ influye en el error al cambiar ZO que es la entrada de la capa output porque $(ZO)_{1\times1} = (ah2)_{1\times3} \times (Wh2O)_{3\times1}$. Entonces $ah2$ puede aumentar o disminuir ZO y después ese ZO es utilizado para calcular la predicción final $\hat{y}$. Tenemos cómo la salida de capa hidden2 influye en el error. Pero la salida depende de la entrada que es $(Zh2)_{1\times3}$. Entonces como la capa hidden2 influye en el error, tenemos que calcular:

$$\delta^{Hidden2} = \frac{\partial MSE}{\partial Zh2} = \frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial ZO} \times \frac{\partial ZO}{\partial ah2} \times \frac{\partial ah2}{\partial Zh2}$$

Pero vemos que hay algunas partes en $\delta^{Hidden2}$ que son comunes con δ^{OUTPUT} : $\frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial ZO}$ por lo tanto aquí es la magia de *backpropagation* al haber muchas operaciones comunes en la regla de cadena, no tenemos que calcular dichas operaciones cada vez por eso hemos creado estas variables intermedias llamada deltas que se pueden utilizar en las partes comunes. Por ejemplo $\delta^{Hidden2}$ se puede calcular como :

$$\delta^{Hidden2} = \delta^{OUTPUT} \times \frac{\partial ZO}{\partial ah2} \times \frac{\partial ah2}{\partial Zh2}$$

Ahora para calcular como la salida de la capa hidden1 afecta al error debemos calcular:

$$\frac{\partial MSE}{\partial ah1} = \frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial ZO} \times \frac{\partial ZO}{\partial ah2} \times \frac{\partial ah2}{\partial Zh2} \times \frac{\partial Zh2}{\partial ah1}$$

La salida de hidden1, $ah1$ tiene influencia sobre el error mediante la entrada de la capa hidden2 porque $(Zh2)_{1\times3} = (ah1)_{1\times5} \times (Wh1h2)_{5\times3}$ y como la salida de la capa hidden1 depende de la entrada al hidden1 para ver cómo la capa hidden1 tiene influencia sobre el error debemos calcular:

$$\delta^{Hidden1} = \frac{\partial MSE}{\partial Zh1} = \frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial ZO} \times \frac{\partial ZO}{\partial ah2} \times \frac{\partial ah2}{\partial Zh2} \times \frac{\partial Zh2}{\partial ah1} \times \frac{\partial ah1}{\partial Zh1}$$

Como podemos observar que también tiene muchas partes comunes con $\delta^{Hidden2}$ entonces podemos escribir $\delta^{Hidden1}$ como :

$$\delta^{Hidden1} = \delta^{Hidden2} \times \frac{\partial Zh2}{\partial ah1} \times \frac{\partial ah1}{\partial Zh1}$$

Ahora, para cada capa tenemos su error en forma de deltas, es decir, cómo esas capas influyen en *MSE*. Ahora vamos a utilizar estas deltas para optimizar los pesos que hay entre esas capas.

Para optimizar $(Wh2O)_{3\times1}$ que conecta la capa hidden2 con la output debemos calcular:

$$\frac{\partial MSE}{\partial Wh2O} = \frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial ZO} \times \frac{\partial ZO}{\partial Wh2O}$$

Porque los pesos $Wh2O$ pueden variar la entrada ZO de la capa output debido a que la entrada ZO es $(ah2)_{1\times3} \times (Wh2O)_{3\times1}$. Pero vemos que en la derivada de *MSE* respecto los pesos $Wh2O$ hay pasos comunes que anteriormente los hemos nombrado como deltas entonces la derivada se puede escribir como:

$$\frac{\partial MSE}{\partial Wh2O} = \delta^{Output} \times \frac{\partial ZO}{\partial Wh2O}$$

El mismo procedimiento para los pesos $Wh1h2$, $Wih1$.

$$\frac{\partial MSE}{\partial Wh1h2} = \boxed{\frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial ZO} \times \frac{\partial ZO}{\partial ah2} \times \frac{\partial ah2}{\partial Zh2}} \times \frac{\partial Zh2}{\partial Wh1h2}$$

$$\frac{\partial MSE}{\partial Wh1h2} = \delta^{Hidden2} \times \frac{\partial Zh2}{\partial Wh1h2}$$

$$\frac{\partial MSE}{\partial Wih1} = \boxed{\frac{\partial MSE}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial ZO} \times \frac{\partial ZO}{\partial ah2} \times \frac{\partial ah2}{\partial Zh2} \times \frac{\partial Zh2}{\partial ah1} \times \frac{\partial ah1}{\partial Zh1}} \times \frac{\partial Zh1}{\partial Wih1}$$

$$\frac{\partial MSE}{\partial Wih1} = \delta^{Hidden1} \times \frac{\partial Zh1}{\partial Wih1}$$

Gradient Descent para optimizar las matrices weights:

Arriba ya hemos demostrado cómo en una red multicapas aplicar el algoritmo de *backpropagation* permite calcular las matrices de gradientes para cada matriz *weights* (parámetros). En este paso vamos a ver cómo en cada iteración optimizar los valores de las matrices pesos para que la red neuronal genere menos errores y prediga bien.

Cada una de las matrices *weights* $(Wih1)_{1\times5}$, $(Wh1h2)_{5\times3}$, $(Wh2O)_{3\times1}$ tiene una dimensión. Entonces, el gradiente que debemos restar a esas matrices debe tener la misma dimensión. Para optimizar la matriz $(Wh2O)_{3\times1}$ se calculan las siguientes multiplicaciones matriciales:

Función de activación se aplica a cada elemento de z por lo tanto su derivada $\frac{\partial \hat{y}}{\partial ZO}$ no es una multiplicación matricial sino una multiplicación elemento a elemento.

$$\delta^{OUTPUT} = \frac{\partial \hat{y}}{\partial ZO} \times \frac{\partial MSE}{\partial \hat{y}}$$

$$\underset{(1,1)}{\delta^{OUTPUT}} = \underbrace{\hat{y}_i \times (1-\hat{y}_i)}_{(1,1)} \times \underbrace{2 \times (\hat{y}_i - y_i)}_{(1,1)}$$

$$\text{GRADIENTE} \rightarrow \frac{\partial MSE}{\partial Wh2O} = \frac{\partial ZO}{\partial Wh2O} \times \delta^{Output}$$

$$\underset{(3,1)}{Gradiente\ Wh2O} = \underbrace{(ah2)^T}_{(3,1)} \times \underbrace{\delta^{OUTPUT}}_{(1,1)}$$

Figura 38. Gradiente para optimizar pesos Wh2O

Como observamos en la figura 38 si realizamos dichos productos matriciales obtenemos el gradiente para restar a nuestra matriz de pesos $Wh2O$. Los valores de la matriz se actualizan de la siguiente manera, incluyendo l*earning rate* para controlar los pasos en la dirección del gradiente:

$$(Wh2O)_{3\times1} = (Wh2O)_{3\times1} - \alpha \times (\nabla Gradiente\ Wh2O)_{3\times1}$$

Para actualizar la matriz $Wh1h2$ se realizan los siguientes pasos (figura 39) :

$$\delta^{Hidden2} = \delta^{OUTPUT} \times \frac{\partial ZO}{\partial ah2} \times \frac{\partial ah2}{\partial Zh2}$$

$$\underset{(1,3)}{\delta^{Hidden2}} = \left[\underset{(1,1)}{\delta^{OUTPUT}} \times \underset{(1,3)}{(Wh2O)^T} \right] \times \underset{(1,3)}{ah2 \times (1 - ah2)}$$

Multiplicación elemento a elemento

$$\text{GRADIENTE} \rightarrow \frac{\partial MSE}{\partial Wh1h2} = \frac{\partial Zh2}{\partial Wh1h2} \times \delta^{Hidden2}$$

$$\underset{(5,3)}{Gradiente\ Wh1h2} = \underset{(5,1)}{(ah1)^T} \times \underset{(1,3)}{\delta^{Hidden2}}$$

Figura 39. Gradiente para optimizar los pesos Wh1h2

$\frac{\partial ah2}{\partial Zh2}$ es la derivada de la función de activación y se multiplica elemento a elemento (figura 40) al resultado que se obtiene al multiplicar siguientes matrices:

$\delta^{OUTPUT} \times \frac{\partial ZO}{\partial ah2}$

$$\left[\underset{(1,1)}{\delta^{OUTPUT}} \times \underset{(1,3)}{(Wh2O)^T} \right] \times \underset{(1,3)}{ah2 \times (1 - ah2)}$$

$$(elemento1 \quad elemento2 \quad elemento3)_{1\times3} \times (a1 \quad a2 \quad a3)_{1\times3}$$

Figura 40. Multiplicación elemento a elemento

Los valores de la matriz $Wh1h2$ se actualizan con la siguiente operación:

$$(Wh1h2)_{5\times3} = (Wh1h2)_{5\times3} - \alpha \times (\nabla Gradiente\ Wh1h2)_{5\times3}$$

Para actualizar la matriz $Wih1$ se realizan los siguientes pasos (figura 41) :

$$\delta^{Hidden1} = \delta^{hidden2} \times \frac{\partial Zh2}{\partial ah1} \times \frac{\partial ah1}{\partial Zh1}$$

$$\underset{(1,5)}{\delta^{Hidden1}} = \left[\underset{(1,3)}{\delta^{hidden2}} \times \underset{(3,5)}{(Wh1h2)^T} \right] \times \underset{(1,5)}{ah1 \times (1 - ah1)}$$

Multiplicación elemento a elemento

$$\text{GRADIENTE} \rightarrow \frac{\partial MSE}{\partial Wih1} = \frac{\partial Zh1}{\partial Wih1} \times \delta^{Hidden1}$$

$$\underset{(2,5)}{\begin{matrix} Gradiente \\ Wih1 \end{matrix}} = \underset{(2,1)}{(input)^T} \times \underset{(1,5)}{\delta^{Hidden1}}$$

Figura 41. Gradiente para optimizar los pesos Wih1

Observamos que los pasos son casi iguales que se utilizaron para calcular el gradiente de la capa hidden2, eso es porque las redes tienen una estructura secuencial y gracias a las similitudes entre operaciones de cada capa podemos utilizar el algoritmo de *backpropagation* para entrenar parámetros(*weights*) de varias capas.

Los valores de la matriz $Wih1$ se actualizan con la siguiente operación :

$$(Wih1)_{2\times5} = (Wih1)_{2\times5} - \alpha \times (\nabla Gradiente\ ih1)_{2\times5}$$

Con esto hemos visto cómo entrenar una red con varias capas. Además, hemos visto en *forward propagation*, cómo utilizando el producto matricial, predecir. Luego, en *Backpropagation*, hemos visto cómo propagar el error hacía atrás para entrenar las distintas matrices *weights*.

3.6 Funciones de activación

En los ejemplos anteriores hemos visto que como función de activación para introducir la no-linealidad hemos utilizado *sigmoid*. Pero con el paso del tiempo

surgió la necesidad de buscar otras funciones de activación, en este apartado vamos a ver distintos tipos de funciones de activación que se utilizan hoy en día.

En las redes neuronales modernas que utilizan varias capas ocultas (10+), *sigmoid* en algunos casos presenta problemas de *vanishing gradient*, en el siguiente párrafo veremos en qué consiste este problema. En el paso *forward propagation*, si se utiliza *sigmoid* como función de activación. Entonces en el paso *backpropagation*, en la regla de cadena se utilizará la derivada de la función *sigmoid*. La derivada de *sigmoid* toma valores entre el rango (0 - 0.25), cómo se observa en la figura 42.

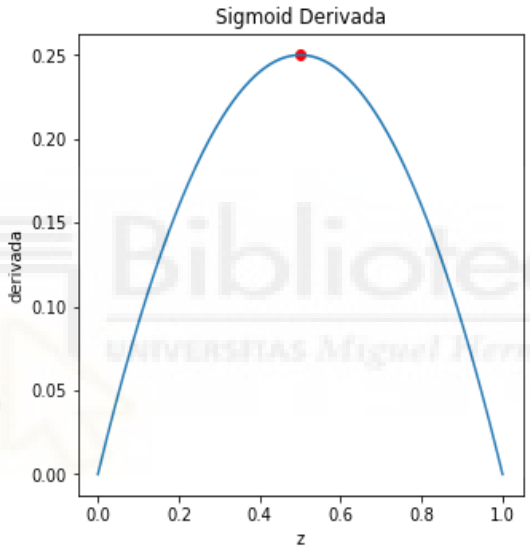


Figura 42. Derivadas de la función sigmoid

Entonces en el problema de *vanishing gradient* las capas que son cercanas al *loss function* sus matrices de pesos optimizan bien porque en el cálculo de su gradiente las derivadas de *sigmoid* solo se multiplican una o dos veces, pero el problema viene cuando tenemos que optimizar las matrices de pesos que son cercanas al capa input porque si entre capa input y output hay 15 capas hidden con la función de activación *sigmoid* entonces para optimizar las matrices de pesos en la capa hidden 1 o 2 debemos multiplicar las derivadas de *sigmoid* más de 10 veces. Las derivadas de *sigmoid* son números comprendidos entre 0 y 0.25 y al multiplicar 10 veces esos números tan pequeños, gradiente casi se desaparece (*vanish*) porque la multiplicación resulta casi 0.

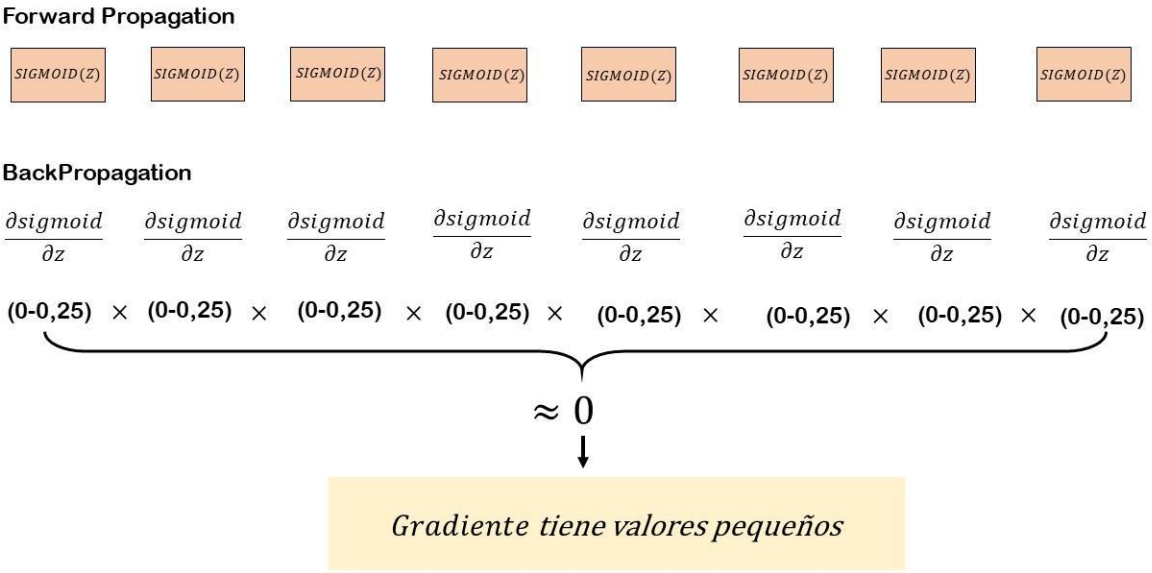


Figura 43. Vanishing Gradient en backpropagation

En la figura 43 se muestra que si tenemos varias capas ocultas(hidden) con *sigmoid* entonces en el paso de *backpropagation* cada capa en su gradiente va tener la derivada de *sigmoid* $\frac{\partial sigmoid = ah}{\partial z}$ y si tenemos que optimizar la matriz de pesos en 8 capas atrás entonces se multiplica la derivada de *sigmoid* 8 veces que resulta en un número que es aproximado a 0. Ahora en el paso de actualizar los pesos ese valor 0 va multiplicar otras derivadas por la regla de cadena y el gradiente resultante va tener valores muy pequeños que van a actualizar muy despacio a pesos en cada iteración o van a dejar de actualizar. Eso resulta en que el error no se disminuye porque los pesos no se están entrenando bien dado que el gradiente desaparece, lo cuál se conoce como problema de *vanishing gradient.*

Otro problema que se da en *sigmoid* es de *Dead neurons* (neuronas muertas). Ese problema ocurre cuando la entrada (Z) en el *sigmoid* es un número demasiado grande o demasiado pequeño. Si observamos las colas de *sigmoid* (figura 44) vemos que *sigmoid* para los números muy grandes y pequeños es casi plano y las derivadas en ese tramo son casi 0.

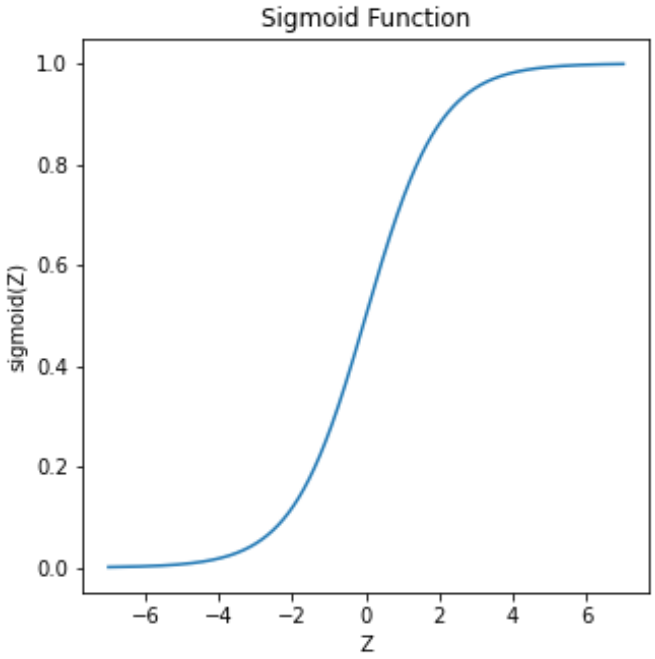
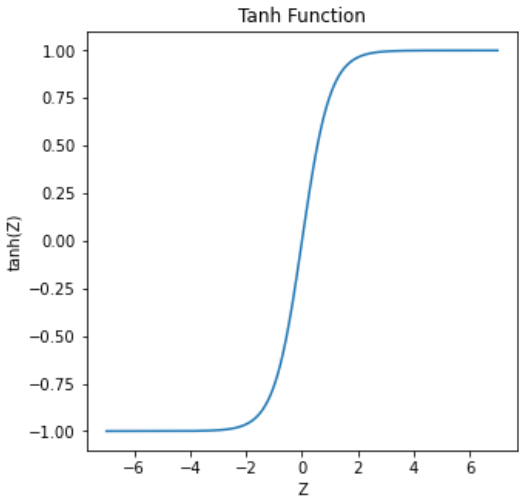


Figura 44. Sigmoid Function

Entonces puede ocurrir que algunas neuronas, tanto en capas iniciales como finales, que reciben como entrada números grandes y pequeños dejen de optimizar sus pesos debido a que la derivada del *sigmoid* para esos números es 0. A esa neurona la consideramos muerta porque sus pesos no cambian en la fase de entrenamiento porque el gradiente tiene valores muy pequeños. En contraste con *vanishing gradient* donde este problema afecta a toda la capa, en *dead neurons* la capa entera o algunas neuronas de esa capa dejan de optimizar, algunas porque la derivada de *sigmoid* se aplica elemento a elemento y solo será próximo a 0 para las capas con entrada grande y pequeña.

Hemos visto algunos de los problemas de usar la función de activación *Sigmoid.* Entonces para solucionar dichos problemas los investigadores de redes neuronales han creado nuevas funciones que ayudan en acelerar el entrenamiento y no presentan problemas citadas anteriormente.

Tanh function :

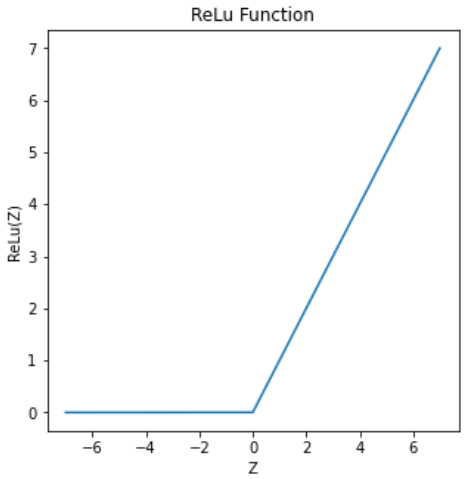


$$\frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Figura 45. Hyperbolic Tangent (Tanh) function

Hyperbolic Tangent puede dar números entre -1 y 1 en cambio *sigmoid* está contenido en el rango de 0 a 1. Si nuestra predicción necesita predecir números entre -1 y 1 entonces en la capa output se puede utilizar esta función de activación.

ReLu function :

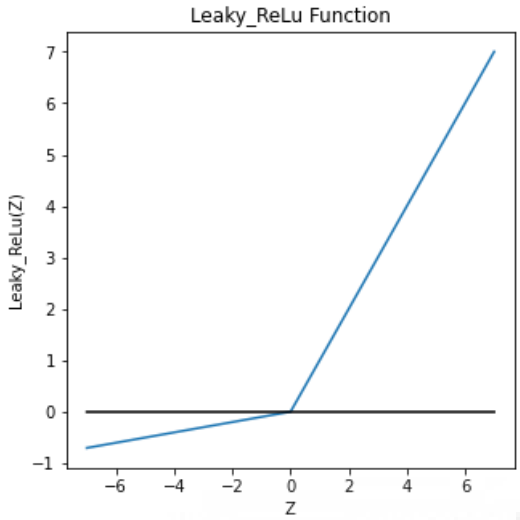


$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

Figura 46. Rectified Linear Unit (ReLu) function

Hoy en día *ReLu* es una de las funciones de activaciones que más se utilizan debido a que su derivada ayuda en entrenar los pesos más rápidos. *Dying ReLu*

(relu muerto): *ReLu* para los inputs negativos devuelve valor 0, y la función en ese tramo es una recta plana eso significa que esa neurona no se activa, su derivada es 0 y estará muerta y a veces surge problema que esa neurona nunca se activa. El problema de *Dying ReLu* es solucionado por una variación de *ReLu* conocida como *Leaky ReLu* (Figura 47)



$$f(x) = \begin{cases} \alpha x, & x < 0 \\ x, & x \geq 0 \end{cases}$$

Figura 47. Leaky ReLu function

Leaky ReLu resuelve el problema de *Dying ReLu*, introduciendo para los números negativos una pequeña pendiente cuya inclinación depende del factor α. Si α es grande, más inclinada es la pendiente, mayor es el valor que toman los números negativos y si α es pequeña, menos inclinada es la pendiente, menor es el valor que toman los números negativos.

Softmax function :

Esta función de activación junto con *ReLu* también es una de las más utilizadas. *Softmax* se utiliza en la capa output para dar predicciones. En las tareas de clasificación multiclase se utiliza esta función porque devuelve probabilidades para cada clase y la suma de probabilidades de todas las clases es 1. Por ejemplo, si queremos clasificar un ejemplo en 3 clases entonces *softmax* nos devuelve lo probable que es que ese ejemplo sea de clase 1, clase 2 o clase 3. (Figura 48).

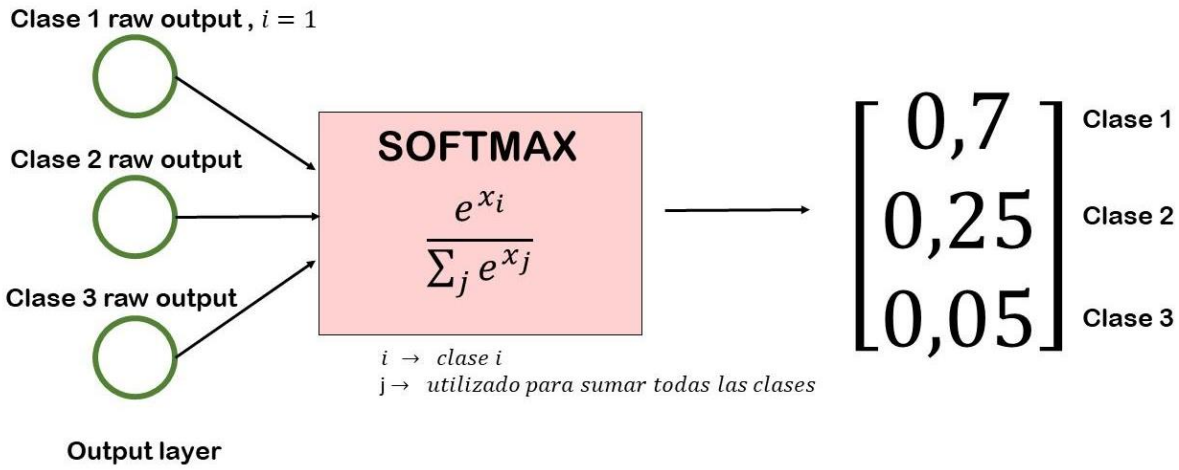


Figura 48. Softmax activation para multiclass Classification

Observamos en la figura 48 que la capa output primero da 3 valores sin aplicarles *softmax* por eso se llama *raw*, 3 porque es un problema de clasificar un objeto en 3 clases. En *softmax* cada clase i es elevada exponencialmente y dividida entre la suma de exponenciales de todas las clases y como resultado obtenemos una matriz de probabilidades que indica que es 70% probable que el objeto sea de clase 1, 25% clase 2 y 5% clase 3.

Podemos utilizar en una capa como función de activación *sigmoid* y en la siguiente *ReLu*, es decir, se pueden combinar distintas funciones de activación. Pero si el problema es de clasificación binaria (2 clases) en la capa output se utiliza *Sigmoid* porque puede dar valores entre 0 y 1 que pueden ser tomados como una probabilidad. Para la clasificación multiclase se debe utilizar *Softmax function* y para las tareas de regresión podemos no utilizar ninguna función o las variantes de *ReLu* en la capa output.

En el siguiente apartado vamos a ver algunas de las técnicas de algoritmo *gradient descent*.

3.7 Variantes de Gradient Descent

En las redes neuronales el algoritmo de entrenamiento consiste en los siguientes pasos:

(1) Inicializar las matrices de pesos (*weights*)

(2) Calcular el Error

(3) Utilizar el Error para calcular el gradiente para cada matriz

(4) Restar el gradiente multiplicado por *learning rate* a las matrices de pesos

(5) Repetir los pasos (2-4)

Ahora si tenemos un *dataset* que contiene varios ejemplos (figura 49) entonces podemos utilizar todos los ejemplos para calcular el gradiente y actualizar los pesos o iterar sobre todos los ejemplos y en cada iteración con un ejemplo calcular el gradiente y actualizar los pesos, a esta variante se le llama *Stochastic gradient descent* (*SGD*).

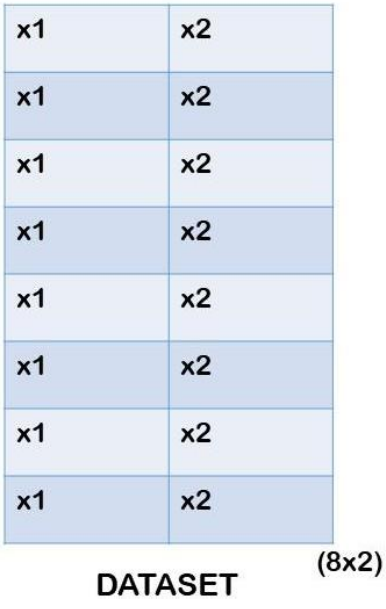


Figura 49. Dataset

Gradient Descent utiliza todos los ejemplos por lo tanto su error no oscila porque utilizamos toda la población para calcular el error en *loss function* y no se pierde la información. Sucede, por ejemplo, cuando estamos situados en una montaña y podemos ver todo lo que hay alrededor nuestro. Mientras que en *stochastic gradient descent* al utilizar solo un ejemplo para calcular el gradiente, el error suele oscilar porque en contraste con *Gradient descent* donde utilizamos la población aquí solo utilizamos una muestra que no nos da una imagen completa de *loss function*(montaña). Tanto el *Gradient Descent* como el *SGD* (*stochastic gradient descent*) minimizan el *loss function* (figura 50).

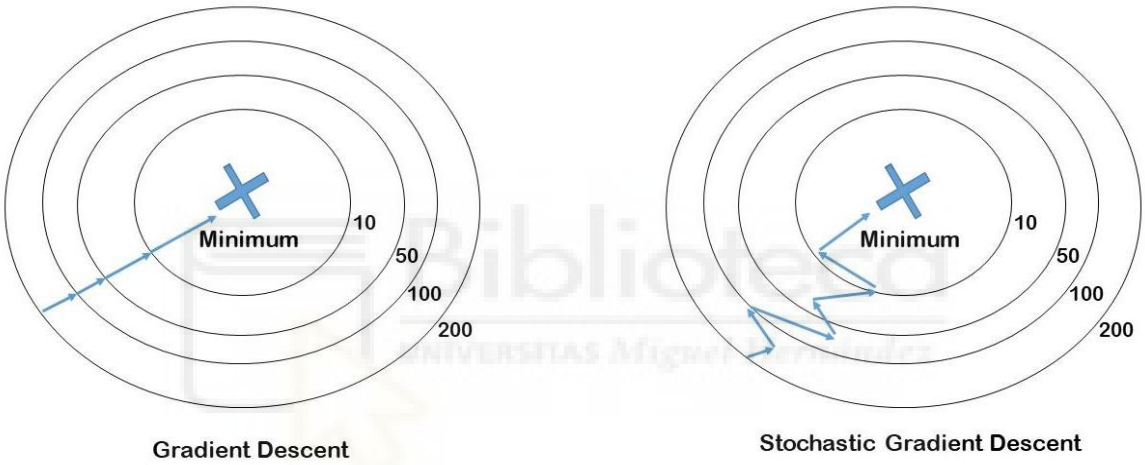


Figura 50. SGD vs Gradient descent: manera de encontrar el mínimo

Durante el entrenamiento realizamos varios *epochs*. Un *epoch* se considera completado cuando se ha visto todo el *dataset*, es decir, cuando se ha calculado el error sobre todo el *dataset*. Entonces, al utilizar el *gradient descent* para actualizar una vez las matrices de pesos, debemos iterar sobre todos los ejemplos de *dataset* y así en cada *epoch*. Si tenemos un *dataset* con 10000 o más ejemplos esto puede ralentizar el entrenamiento o que todos los ejemplos para calcular el error no puedan caber en la memoria. Para eso *SGD* al utilizar un solo ejemplo para calcular el gradiente en un *epoch* puede actualizar las matrices tantas veces como haya ejemplos en un *dataset*, es rápido y no tiene problemas de memoria.

SGD puede oscilar mucho por lo tanto hay otra variante de *Gradient descent* que está entre *SGD* y *Gradient descent*: *Mini-Batch Gradient Descent*. (Figura 51)

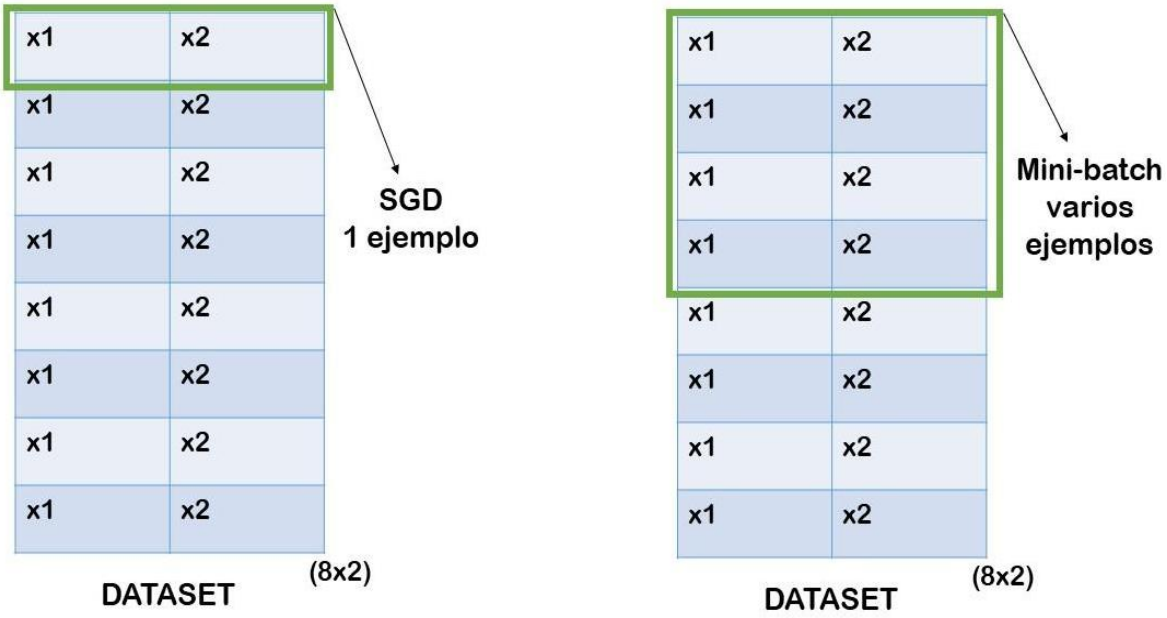


Figura 51. Mini-Batch vs Sgd

En *Mini-batch* en un *epoch* para calcular el gradiente se puede utilizar más de 1 ejemplo, pero nunca se escoge el *dataset* entero porque eso ya lo *hace Gradient descent*. En *mini-batch* si cogemos un número de ejemplos cercanos a los que tenemos en el *dataset* entonces *mini-batch* no produce muchas oscilaciones en el error porque cogemos una muestra representativa del *dataset*. En cambio, si cogemos un número de ejemplos pequeño entonces el error oscila. Por lo tanto, lo mejor es ir probando con diferentes tamaños de *mini-batch* para ver cuál es rápido y se ajusta bien a nuestros ejemplos, es decir, no ralentiza el entrenamiento.

Existen otras variantes de gradiente que utilizan diferentes técnicas para acelerar el entrenamiento de las redes como, por ejemplo: *Gradient Descent* con *Momentum* o *Nesterov accelerated gradient* que a la hora de calcular los gradientes tienen en cuenta el gradiente del paso anterior. Existen otras variantes de gradientes que son conocidas como *adaptive* gradientes. Como hemos visto, al gradiente se le multiplica por un *learning rate* que debemos elegir

nosotros. Pero en *adaptive methods* como *adagrad, adadelta, Adam, RmsProp* el *learning rate* es actualizado por el algoritmo y esas variantes son las que más se utilizan hoy en día.

3.8 Regularization

Las redes neuronales hoy en día están de moda por sus capacidades de crear modelos para resolver casos complejos. Las redes neuronales gracias a sus capas hidden pueden ajustar funciones complejas. Gracias a las capas ocultas las redes neuronales pueden ajustar casi cualquier función, como lo vemos en la figura 52.

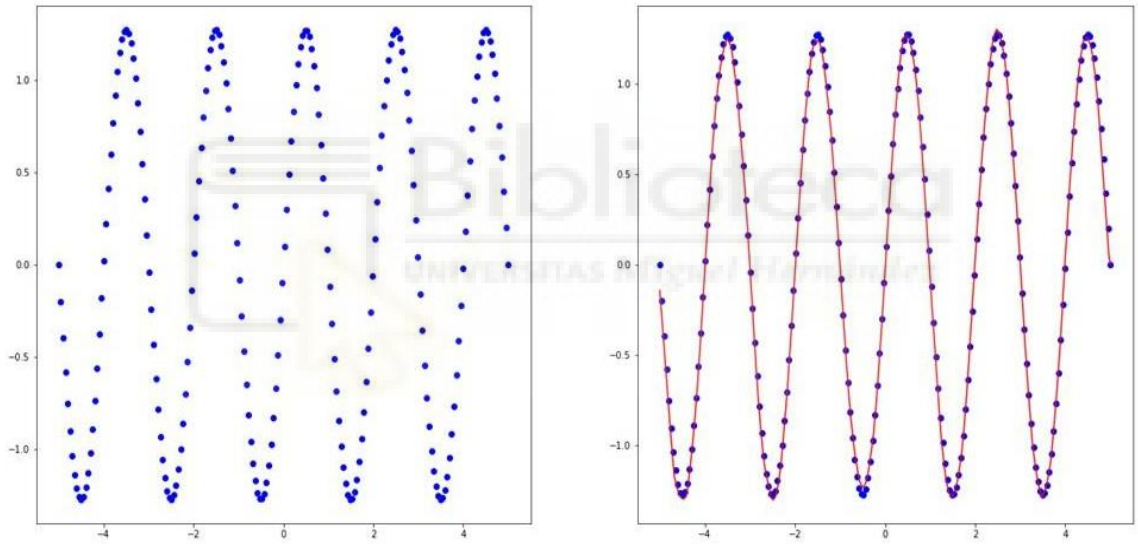


Figura 52. Función matemática ajustada por una red neuronal

Cuantas más capas hidden tenga nuestro modelo más poder tiene para aprender funciones complejas, pero introducir muchas capas puede traer problemas de *overfitting*. A continuación, vamos a explicar en qué consiste el *overfitting* o sobreajuste.

Como cualquier modelo de *machine learning*, las redes neuronales se entrenan luego para predecir datos que no han sido utilizados durante la fase de

entrenamiento, por lo tanto las redes neuronales deben generalizarse bien. Como cualquier modelo tenemos que encontrar equilibrio entre un modelo simple y complejo. Un modelo simple no tiene poder para dar buenas predicciones mientras que un modelo complejo aprende información innecesaria del *dataset* y no generaliza bien. Vamos a ver este proceso mediante un ejemplo.

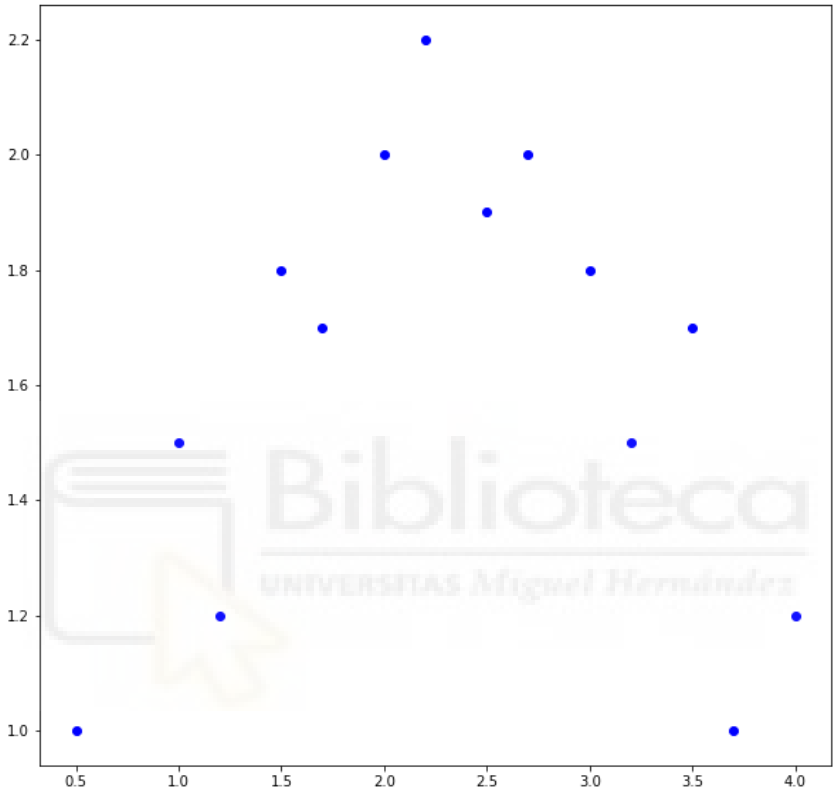


Figura 53. Dataset usado para entrenar

En la figura 53 tenemos un *dataset* que podemos entrenar con las redes neuronales para ajustar una función que tenga mínimo error y que además generalice bien para los ejemplos no vistos en el período entreno. Una buena aproximación mediante una función se muestra en la figura 54 donde vemos que la función ajustada es cercana a los puntos, menos error y además lo podemos utilizar para predecir ejemplos nuevos.

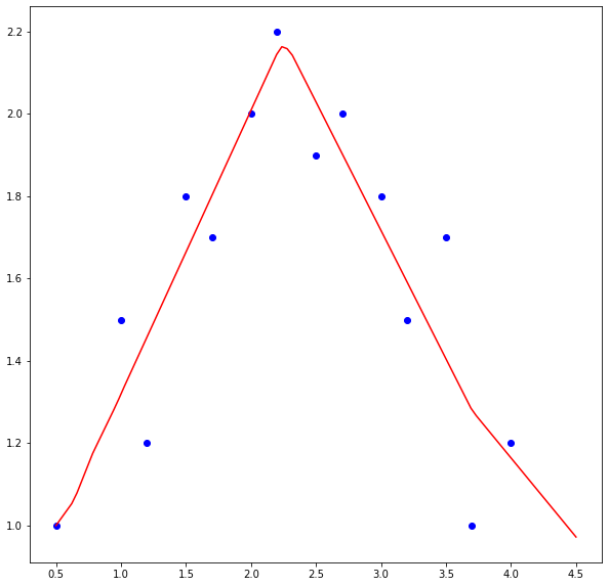


Figura 54. Una aproximación buena

En la figura 54 vemos una buena aproximación mientras que en la figura 55 vemos un *overfitting*, es decir, el modelo ajusta una función muy compleja que no tiene error, pero no es útil para predecir nuevos ejemplos porque ha aprendido información innecesaria y no el patrón adecuado.

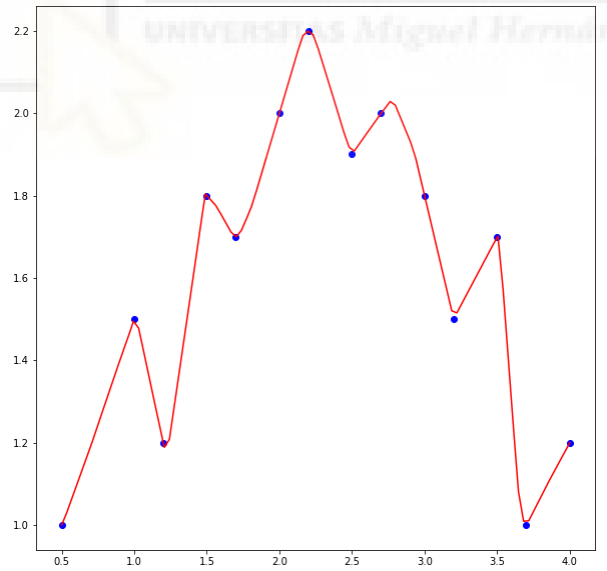


Figura 55. Overfitting de una función

Overfitting sobreajusta la función, pero también hay casos de *underfitting* como lo vemos en la figura 56 donde nuestro modelo lineal en este caso no es adecuado para este *dataset* porque con una línea no podemos aproximar los

puntos que tenemos, necesitamos más variables en nuestro modelo, o un modelo más complejo.

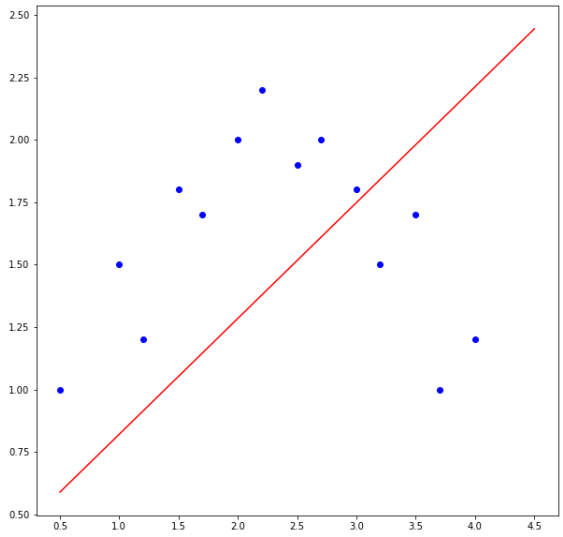


Figura 56. Underfitting

Para ver que un modelo no se está sobre ajustando hay varias técnicas para regularizar el modelo, la primera que es la que se utiliza en todos los modelos es separar el *dataset* en *train* y *test* (figura 57). Se ajusta el modelo sobre *train data* y una vez completado el entrenamiento se evalúa el rendimiento del modelo sobre *test data*.

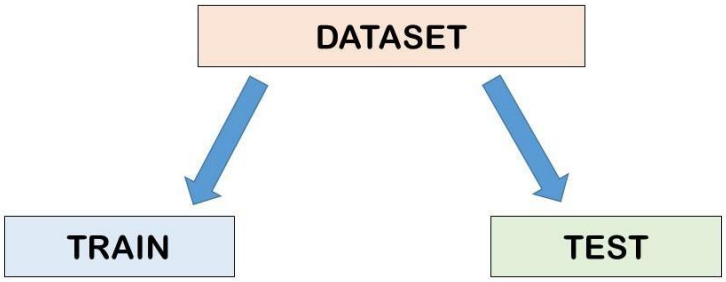


Figura 57. Dataset dividido en train and test

Otras técnicas consisten en reducir el número de capas hidden para no tener un modelo complejo o reducir el número de neuronas en cada capa hidden. *Early stopping* es una técnica basada en localizar el instante en el que el error sobre test data después de reducir, empieza aumentar mientras que en training data sigue decreciendo. Ello significa que el modelo ha empezado a aprender

información innecesaria de *train data* y no se está generalizando bien a *test data*. Por lo tanto, cuando el error empieza a aumentar en test data debemos de parar el entrenamiento. Podemos ver en la figura 58 cómo el error sobre *test* empieza a subir después de decrecer.

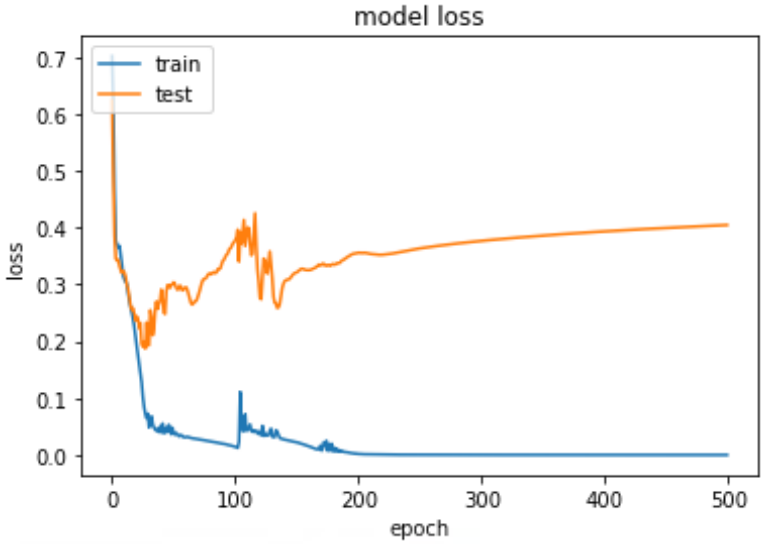


Figura 58. Overfitting empieza cuando test error sube

Otros métodos utilizados para regularizar el modelo son: *dropout*, *L1* Y *L2 regularization*. En *Dropout* durante el entrenamiento en cada *epoch* un x número de neuronas aleatoriamente son desactivadas en cada capa oculta para que un modelo no memorice caminos. Por ejemplo, en sobreajuste un modelo puede memorizar que si los inputs son en tal rango debo predecir esto porque ya tiene memorizado un camino. Desactivando en cada *epoch* un porcentaje de neuronas en cada capa oculta obligamos al modelo a no memorizar training data.

L1 y *L2 regularization* añaden al *Loss function* un término más. Por ejemplo, en L1 *regularization* al error se le añade la suma de todos los pesos en valor absoluto multiplicado por un factor landa:

$$L1 = LOSS + \lambda * \sum |w|$$

Por lo tanto, si añadimos en *loss function* la suma de los pesos en valor absoluto, entonces para minimizar el error el modelo mantiene los valores de pesos lo mínimo posible. L2, en vez de valor absoluto, añade la suma de los pesos al cuadrado:

$$L2 = LOSS + \lambda * \sum w^2$$

Al minimizar los pesos algunos pueden llegar a tener valores cercanos a 0. Entonces las variables de entrada que multipliquen a esos pesos no tendrán influencia en el modelo y el modelo automáticamente seleccionará unas variables antes que otras. Además, sólo seleccionará aquellos parámetros que en gran medida ayudan a reducir el error, disminuyendo así probabilidades de aprender información innecesaria y sobreajuste.

Tanto landa en *L1* y *L2 regularization* como *learning rate* son parámetros que se van probando hasta encontrar el mejor para el modelo. Los *weights* se inician aleatoriamente, pero hay algoritmos para inicializar los pesos también como: *He initialization*, *Xavier initialization* o *Xavier-Glorot Initialization*.

Se recomienda estandarizar las variables de entrada y que tengan valores pequeños para luego en alguna función como *sigmoid* no tener problemas de gradiente. Cuanto más grandes sean en magnitud las variables input, mayor será la magnitud del gradiente y más pequeño será el *learning rate*, que nos ayuda coger una proporción de ese gradiente para luego restarlo a los pesos.

Learning rate, si los rangos de input son entre 0 y 1, se calcula probando inicialmente por 0.1 o 0.01. Si vemos que el error no disminuye se va aumentado o disminuyendo el *learning rate* hasta encontrar un punto donde el error de modelo disminuye en cada *epoch*.

En las redes neuronales además de *weights* también podemos agregar otro parámetro que es *bias*. Entonces, si tenemos una neurona y dos entradas, la entrada a esa neurona sería $z = x1 * w1 + x2 * w2 + bias$. Pero para no inicializar otra matriz de *bias*, el truco que se puede utilizar es agregar al *dataset* de inputs una nueva columna de 1. Al agregar la nueva columna de 1 podemos multiplicar esa columna por matriz *weights* y no es necesario agregar *bias* porque la columna de 1s funciona como *bias* : $bias = 1 * w0$. La entrada final para una neurona se calcula entonces de la siguiente manera:

$$z = x1 * w1 + x2 * w2 + 1 * w0$$

3.9 Arquitecturas de las redes neuronales y Frameworks

Anteriormente hemos estudiado los fundamentos de las redes neuronales desde cero, dado que nos dan una idea de cómo funcionan las redes neuronales y podemos utilizar ese conocimiento para aprender otras arquitecturas de las redes neuronales como:

- *Convolutional networks*: utilizado para tareas que tengan datos en forma de imágenes.

- *NLP (Natural Language Processing):* utilizado para tareas relacionadas con lenguaje. Por ejemplo, traducción por *machine*, *chatbots* o clasificar títulos de noticias y páginas web en distintas categorías.

- *GAN (Generative Adversarial Networks):* utilizado para crear imágenes y arte desde cero es decir por ejemplo la red aprende crear imágenes de un caballo y crea una imagen que no es tomada por nadie sino la red ha creado, también se utiliza esta arquitectura para imitar arte de pintores clásicos, por ejemplo, crear una foto al que se le aplica el estilo de van Gogh o de Picasso.

Convolutional Network [5] : las imágenes son pixeles que son organizadas en *grid*(matrices) como lo podemos observar en la figura 59 donde una imagen de 0 no es nada más que una matriz de 1s y 0s donde 1 representa el color negro y 0 el blanco.

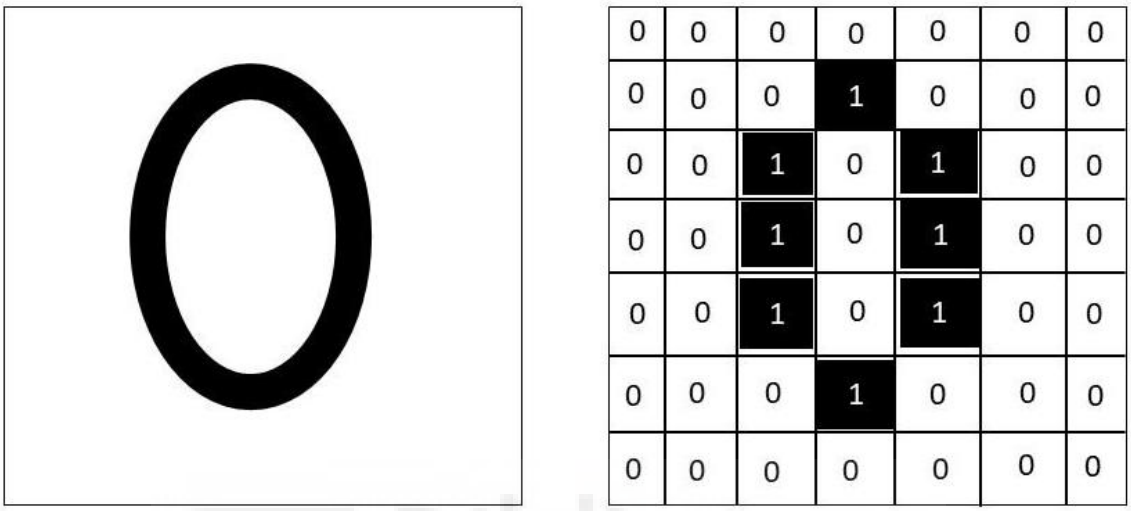


Figura 59. Imágen es una matriz de pixeles

En la figura 59 vemos la imagen de 0 representada en una *grid*(matriz) 7x7. Si a cada píxel lo tomamos como un input entonces para esta imagen tenemos 7x7=49 inputs, pero hoy en día las imágenes son en *RGB* (colores) y los píxeles son matrices de 1000 o más por lo tanto tomar cada pixel como input a veces es costoso computacionalmente y tomar cada pixel como input tampoco nos da una idea sobre lo que hay en la imagen, para ellos están las redes convolucionales.

Una convolución consiste en aplicar un *filter (kernel)* que es una matriz sobre la imagen y calcular producto escalar entre la matriz(*filter*) y la imagen y con ese producto escalar ir rellenando otra matriz que se llama *feature map* (Figura 60).

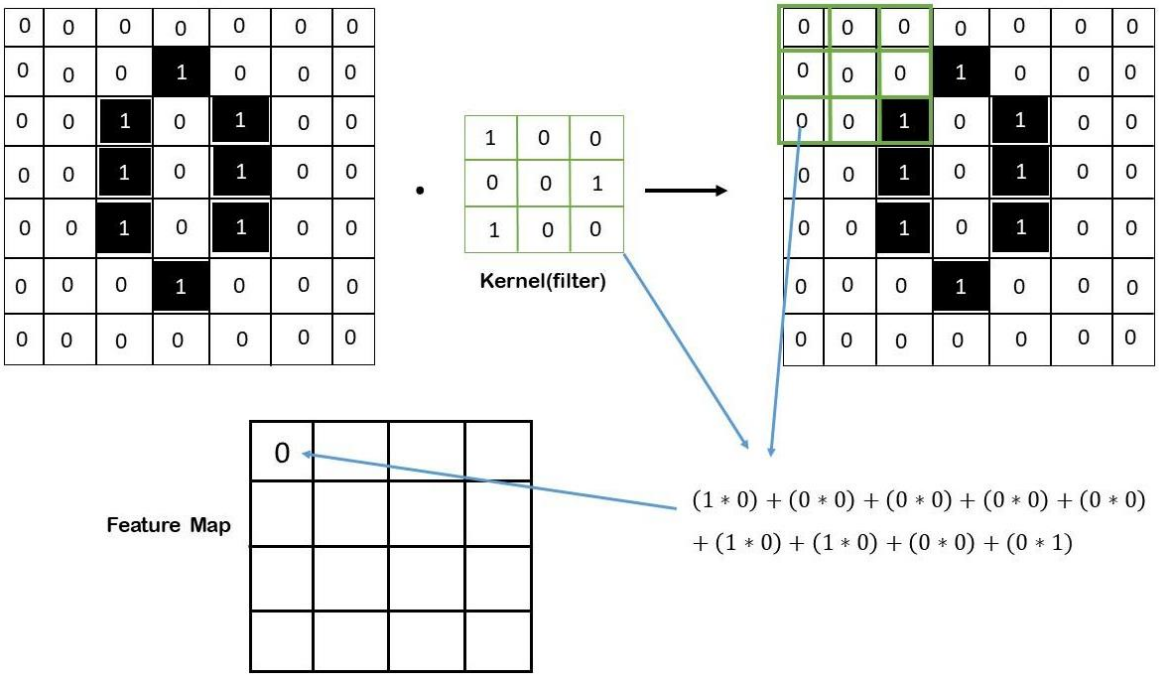


Figura 60. Convolutional operation

Como vemos en la figura 60 se aplica sobre la matriz de pixeles una operación convolucional con *kernel* y obtenemos el primer elemento de nuestro *feature map*. Los valores de *kernel* se inician de forma aleatoria y se entrenan mediante *backpropagation*, se inician varios *kernels*. El *Kernel* va desplazándose sobre la matriz de píxeles para calcular el producto escalar y rellenar *feature map* (figura 61).

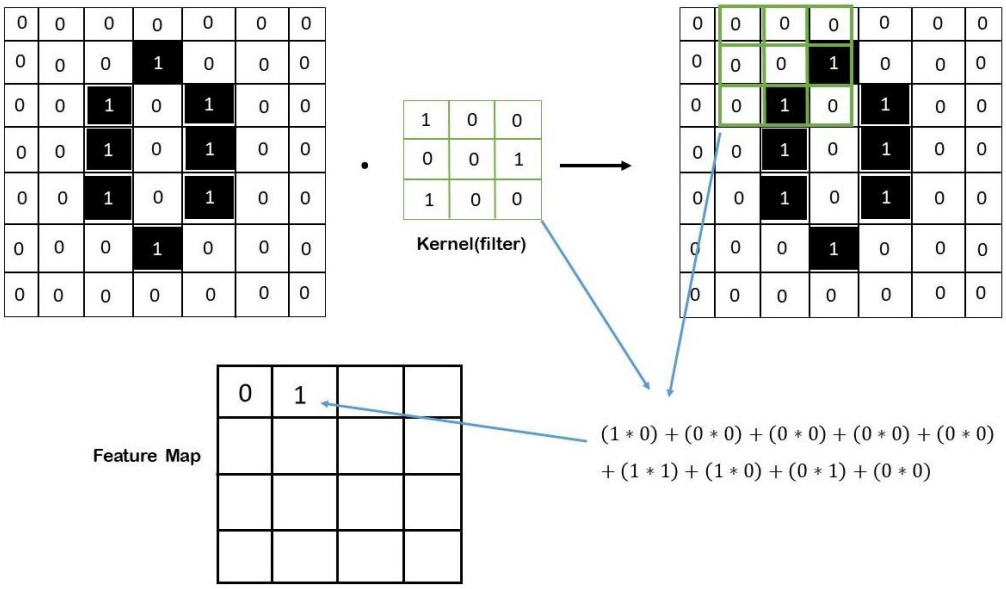


Figura 61.Kernel desplaza sobre la matriz

Una vez que se completa el *feature map*, se le aplica una función de activación y es preparado para pasar por otros tipos de filtros como *max pooling* en cual creamos una ventana que se mueve sobre el *feature map* y esa ventana es una matriz y saca el valor máximo de la parte de *feature map* sobre la cual está situada (figura 62).

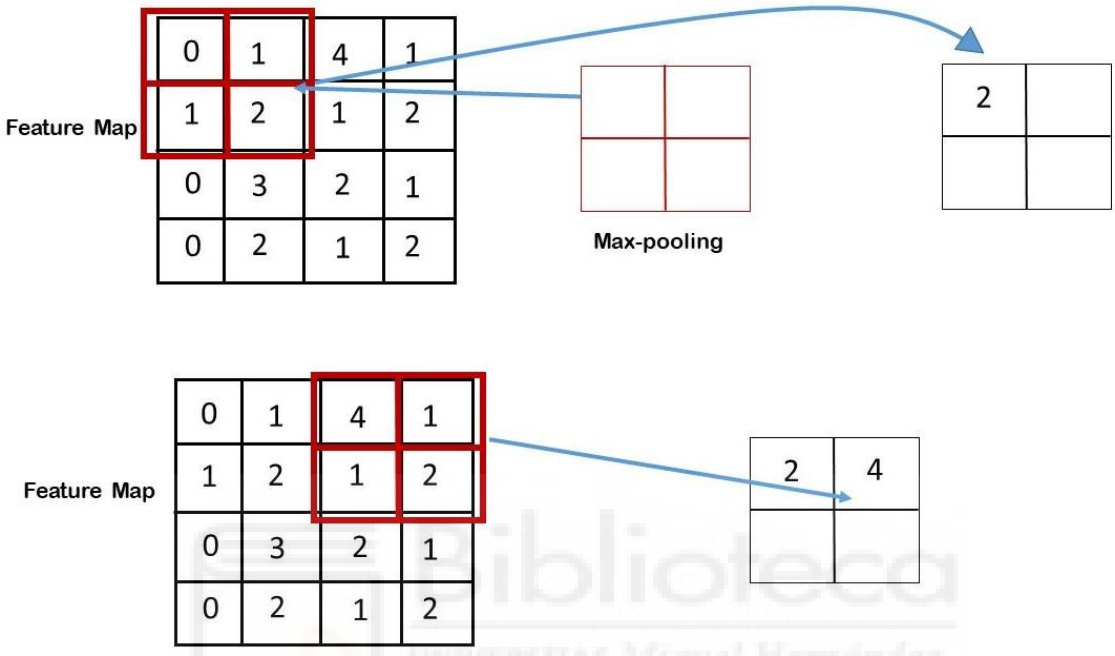


Figura 62. Max pooling

Al aplicar estos filtros lo que hacemos es reducir el tamaño de los inputs que en el siguiente paso van a entrar en la red neuronal que hemos visto anteriormente (*MLP*). Al aplicar estos filtros reducimos el tamaño, pero manteniendo información crucial de cada imagen. Una vez aplicados los filtros el siguiente paso es aplanar(*flattening*) la imagen para que pueda ser introducida en una red neuronal *fully connected* que hemos descrito anteriormente, es decir de una matriz pasar a una fila.

$$\begin{pmatrix} pixel1 & pixel2 \\ pixel3 & pixel4 \end{pmatrix} \rightarrow \begin{pmatrix} pixel1 & pixel2 & pixel3 & pixel4 \end{pmatrix}$$

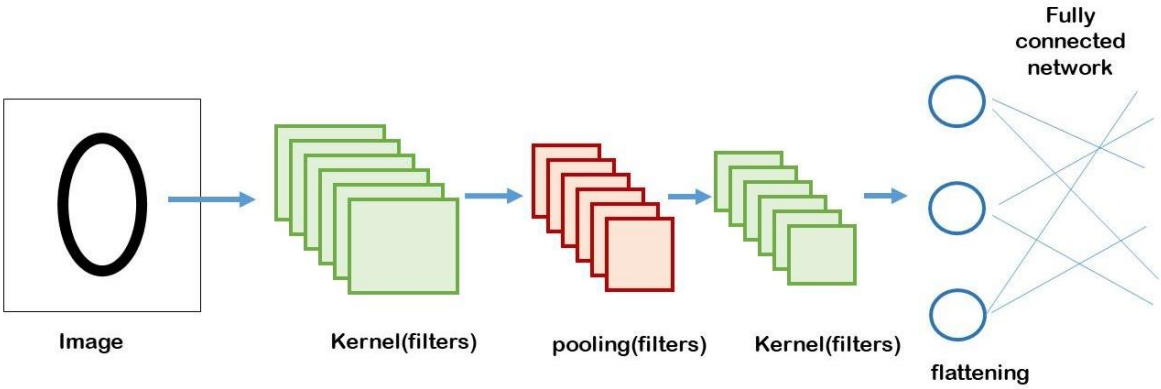


Figura 63. Convolutional arquitectura

En la figura 63 vemos una posible arquitectura de una red *convolutional*, donde la imagen para reducir su tamaño, preservando sus características es pasado por varios filtros y en el paso final es aplanado para introducirlo en una red neuronal *fully connected*. Podemos poner varias capas *pooling* y *kernels* dependiendo de la tarea.

En *GANs (Generative adversarial Networks)*: La idea detrás de *GANs* es que hay dos redes neurales que compiten entre ellas, una se encarga de crear fotos por ejemplo de caballos y la otra actúa como un detective para detectar si esa imagen creada es real o falsa. Compiten entre ellos la red que se encarga de crear imágenes cada vez debe crear imágenes realistas de caballos para que la red detective no pueda detectar si es real o falso, la red detective también se va mejorando para detectar las imágenes falsas.

Una vez que se domina la base de las redes neuronales (*mlp* o *fully connected*) el siguiente paso es elegir una rama (*convolutional, nlp, gans*) para ir profundizando más en otra arquitectura. En las redes neuronales con varias capas, calcular el gradiente puede ser complicado, por eso existen distintos *framework* que facilitan crear redes neuronales. Una vez que sabemos cómo funcionan las redes podemos utilizar esos *frameworks* para programarlos.

Los *frameworks* para programar las redes neuronales que más se utilizan hoy en día son: *Tensorflow, keras* [4] y *pytorch*. En este trabajo en la parte práctica utilizaremos *keras* para programar algunos ejemplos de las redes neuronales.

Tensorflow y *keras* trabajan con tensores, tensores son *arrays* multidimensionales (figura 64):

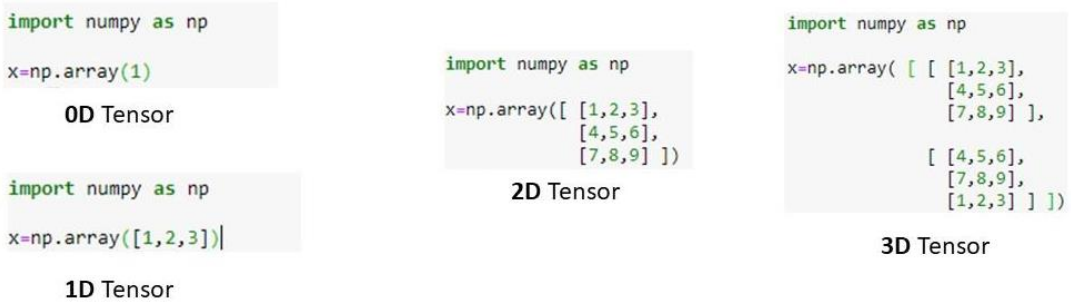


Figura 64. Tensores multidimensionales

Para trabajar con *keras* debemos transformar nuestro *dataset* en tensores.

4.- Resultados

4.1 Regression

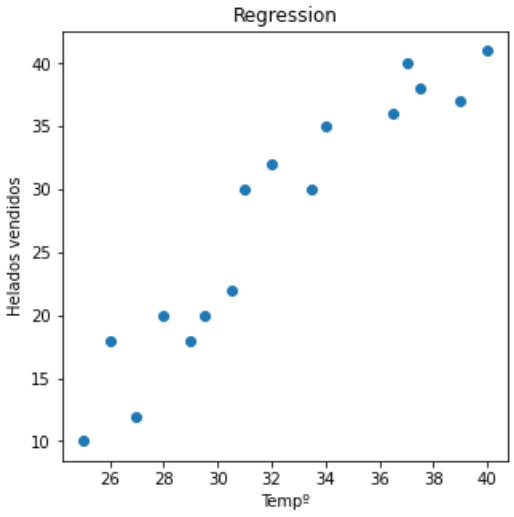


Figura 65. Temperatura y helados: ejemplo regression

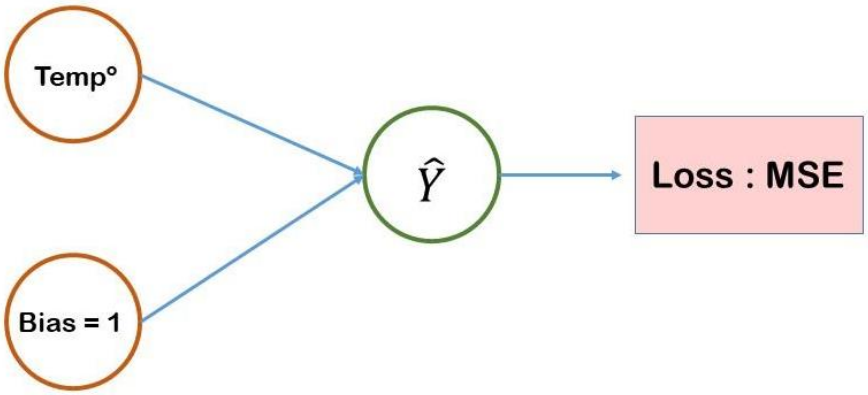
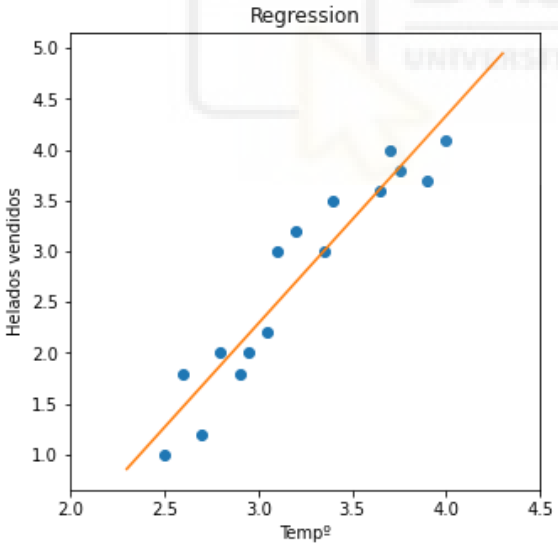


Figura 66. Red neuronal para regresión

Para el ejemplo de la figura 65 vamos a utilizar una red neuronal que tiene dos entradas uno que nos dice la temperatura y otro el término *bias*. Las entradas antes de multiplicar por los pesos son divididas entre 10 y después se multiplican con dos parámetros pesos para producir la predicción $\hat{Y}$ (Figura 66). El resultado es la siguiente recta ajustada a los datos :



MSE : 1.5546

Figura 67. Modelo regresión ajustado

Vemos en la figura 67 nuestra red calcula la mejor recta (modelo regresión) para nuestros datos y en la figura 68 vemos como nuestro gradiente (pasos blancos) llega al mínimo (cruz verde) de *loss function*.

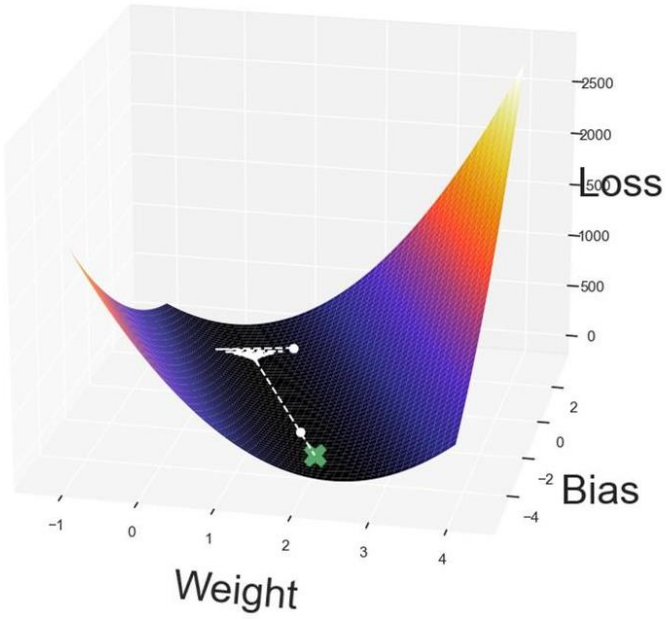


Figura 68. Gradient Descent para encontrar el mínimo

4.2 classification 2 classes

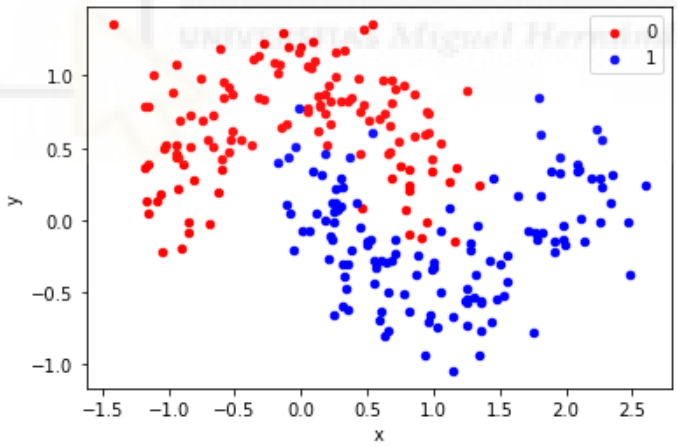


Figura 69. Classification dataset para 2 clases

En la figura 69 vemos un *dataset* que no es separable linealmente entonces vamos a utilizar función de activación *sigmoid* para clasificar este *dataset*. Vamos a ajustar dos modelos de redes neuronales sobre este modelo. Uno va a hacer *overfitting*, otro va a dibujar una frontera de decisión adecuada. También

veremos cómo cambiando la función de activación usando una función *ReLu*, el tiempo para entrenar se reduce bastante.

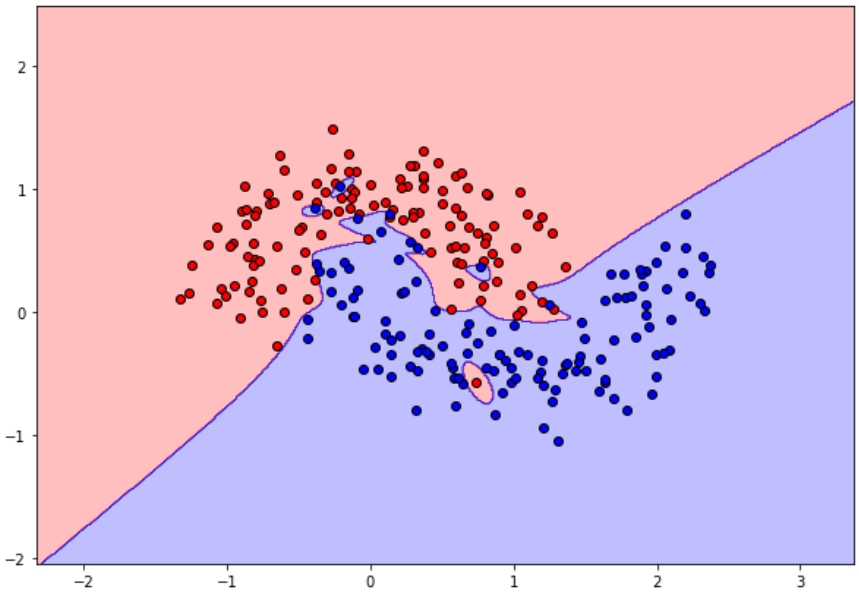
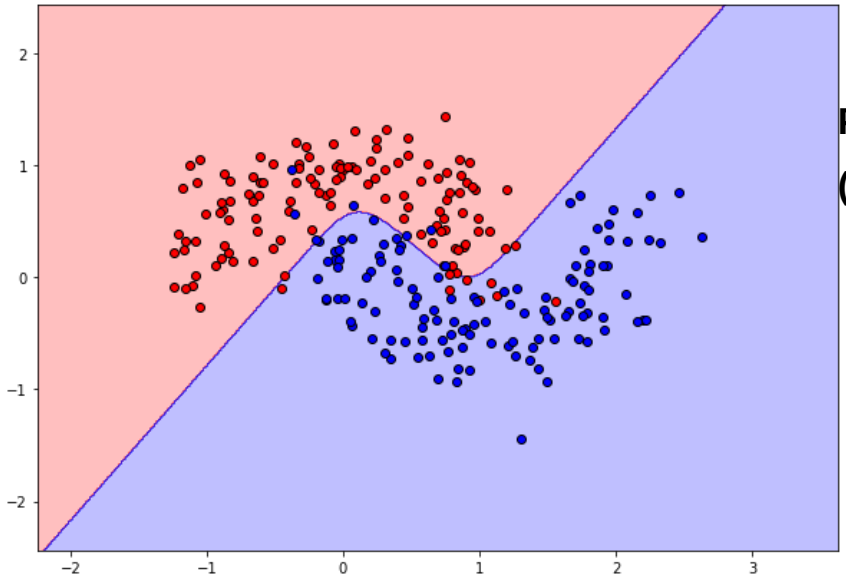


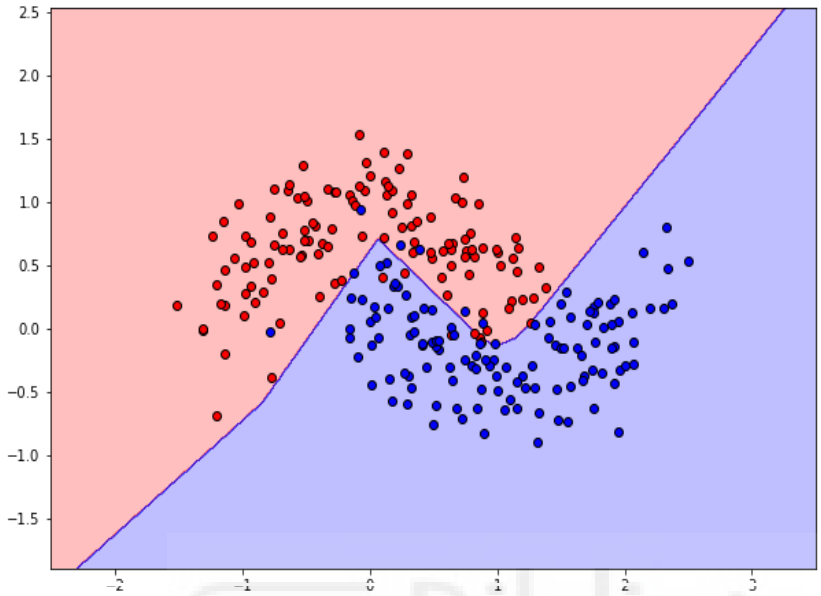
Figura 70. Overfitting con un modelo complejo

Vemos en la figura 70 que las fronteras de decisión son muy complejas, la red clasifica bien todos los puntos, pero no va generalizar bien porque se ha sobreajustado a este *dataset*. Para este modelo hemos utilizados 3 capas ocultas de *sigmoid* con 100 neuronas en la primera capa ,70 en la segunda y 50 neuronas en la tercera capa oculta y un *batch* de 100 datos con el método de *gradient descent: Adam*.



Precisión: 93.60% (accuracy)

En la figura 71 vemos un modelo que es adecuado para generalizar y para dicho modelo hemos utilizado una capa oculta con 10 neuronas y función de activación *sigmoid*. Dicha precisión la hemos conseguido entrenando la red por 6000 *epochs*, es decir, la red neuronal ha recorrido todo el *dataset* 6000 veces.

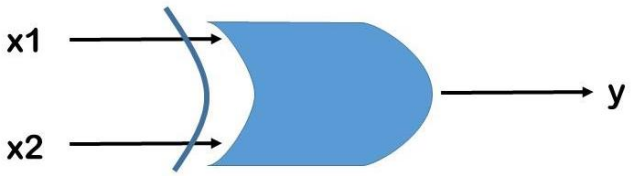


Precisión: 93.60% (accuracy)

En la figura 72 hemos cambiado la función de activación de la capa oculta por una función *ReLu* y en vez de 10 neuronas tenemos 8. En 3000 *epochs* con *ReLu* hemos sido capaces de lograr una precisión igual que la que hemos logrado en *sigmoid* con 6000 *epochs*.

4.3 XOR Gate

En este ejemplo vamos a aprender el funcionamiento de la puerta lógica XOR (Figura 73) mediante las redes neuronales.



x1	x2	y
0	0	0
1	0	1
1	0	1
1	1	0

Figura 73. Puerta Lógica XOR

Este ejemplo simple lo vamos a utilizar para ver la diferencia entre *Gradient Descent* y *Stochastic Gradient Descent (SGD)*. La red neuronal utilizada tiene una capa oculta con 5 neuronas y en la capa output y oculta(hidden) se utiliza función de activación *sigmoid* y un *learning rate* de 1.

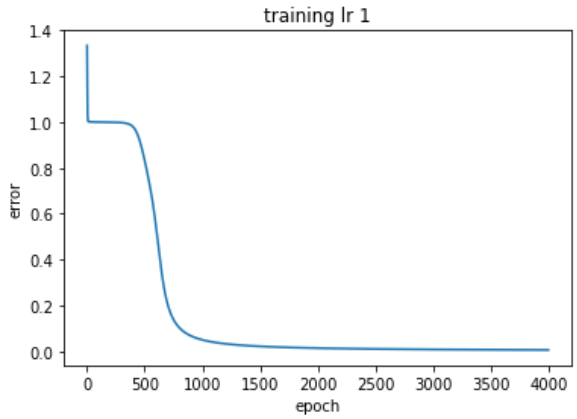


Figura 74. Gradient Descent

En *gradient descent* (figura 74) vemos como después de 1000 *epochs* el error llega al 0.

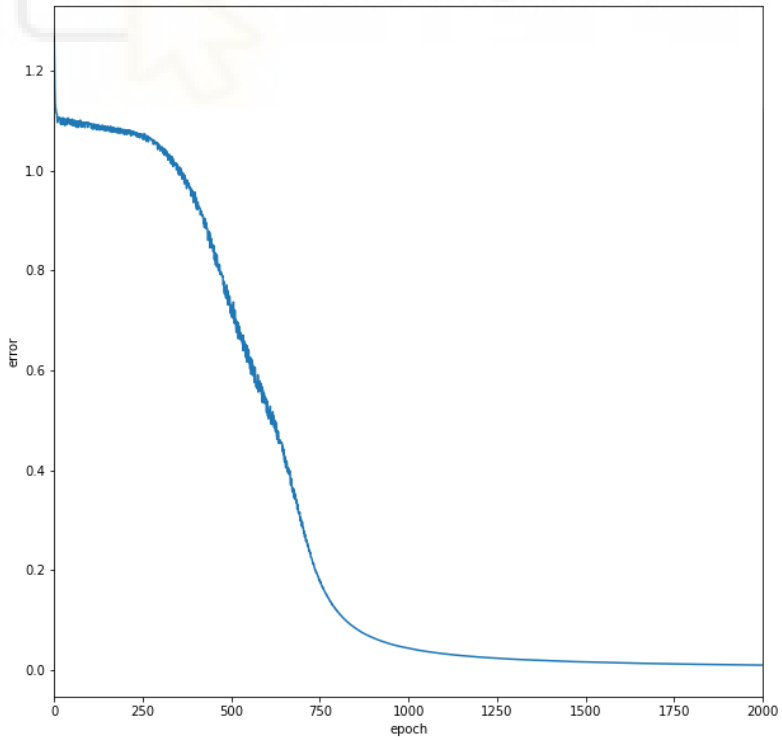


Figura 75. Stochastic Gradient Descent (SGD)

En la figura 75 observamos que la línea, hasta 750 *epochs* no es plana. Hay oscilaciones, debido a que para entrenar hemos utilizado *SGD (Stochastic gradient)*, en que el gradiente se calcula sobre un ejemplo solo. Se aprecian mejor las oscilaciones haciendo zoom (Figura 76).

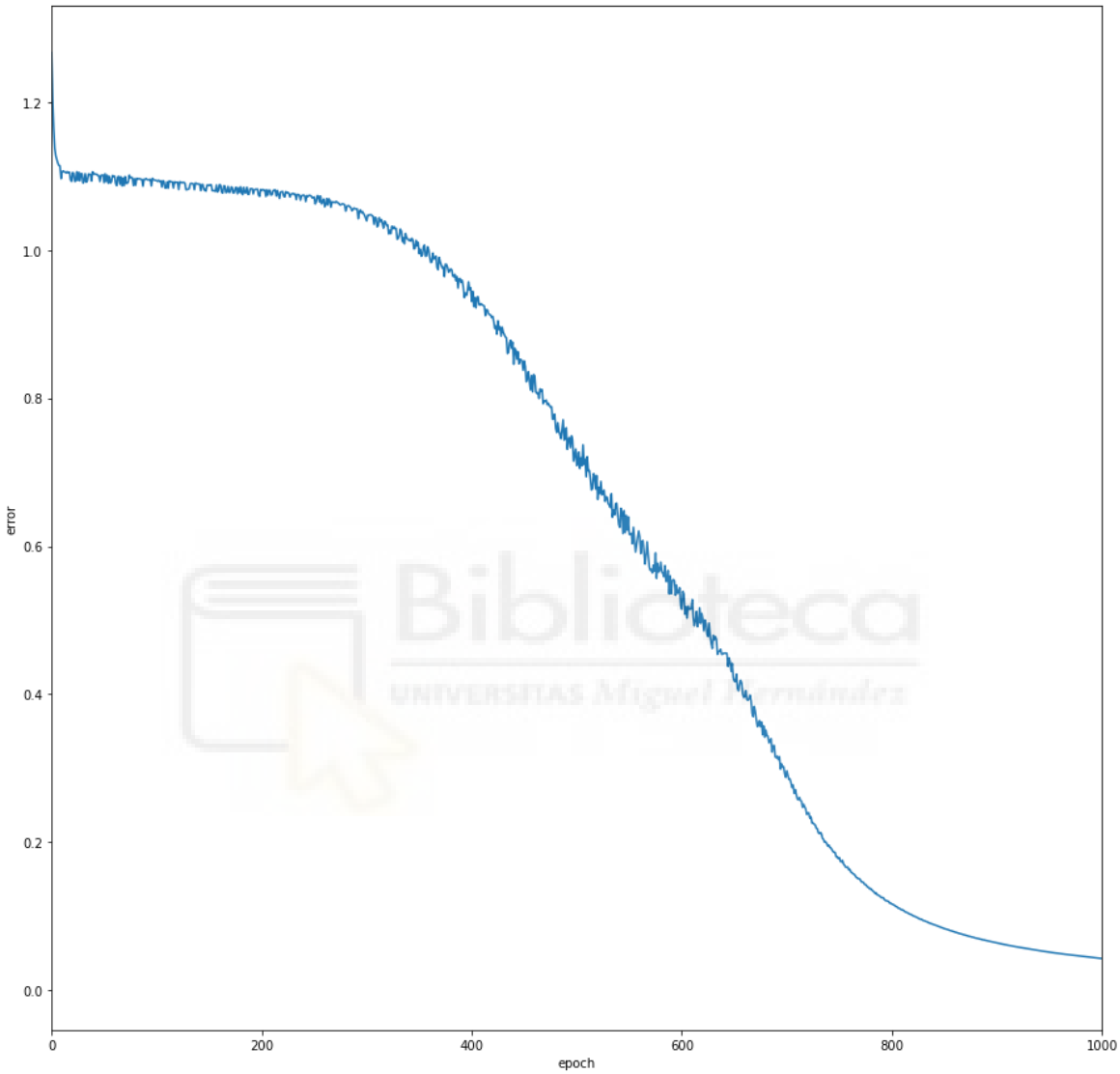


Figura 76.SGD ZOOM

4.4 Multiclass Classification: MNIST Dataset

The MNIST database (Modified National Institute of Standards and Technology database), es un *database* de dígitos de 0 al 9 escrito a mano. Este *database* contiene 60000 imágenes para *train data* y 10000 para el *test*, las imágenes están en blanco y negro de tamaño 28x28 pixeles, algunas de estas imágenes se muestran en la figura 77.

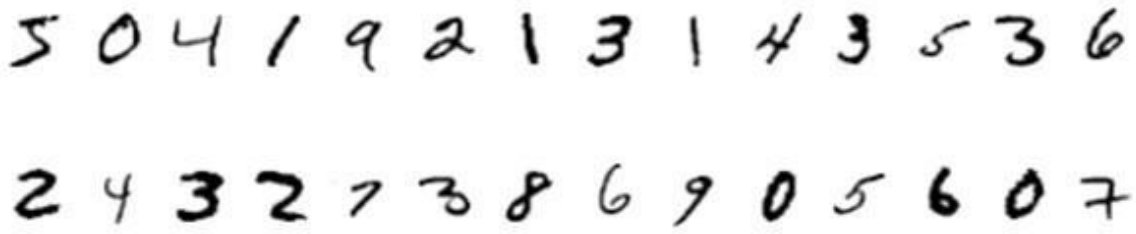


Figura 77. MNIST dígitos algunos ejemplos

Cada imagen es una matriz de 28x28 pixeles y antes de entrenar la matriz es aplanada (*flattening*) en un vector que tiene 784 números, porque 28x28=784. Cada número en ese vector aplanado es un valor entre 0 y 255, 0 : blanco y 255: negro (figura 78).

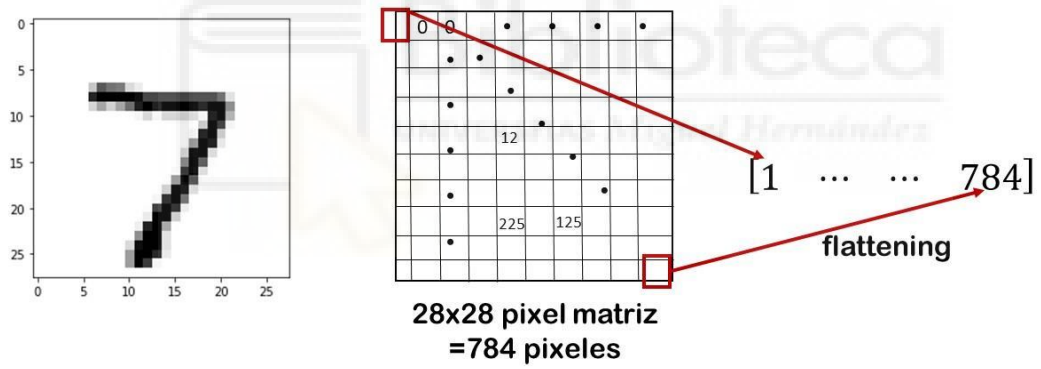


Figura 78. Imagen es transformada en un vector mediante flattening

Cada valor de vector es dividido entre 255 para que los valores tengan un rango entre 0 y 1. Para los valores de y que son *labels* que indican el número en la imagen, al ser un problema de clasificación multiclase son transformados en matrices *dummy (one-hot-encoding)* :

$$LABELS \rightarrow [0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9]$$

$$5 \rightarrow [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0]$$

$$7 \rightarrow [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0]$$

Para clasificar este *dataset* hemos utilizado dos capas ocultas con la función de activación *ReLu* y en la capa output hemos utilizado *softmax* por ser problema de clasificación multiclase. Hemos entrenado dicho modelo por 50 *epochs* con el *loss función* entropía cruzada (*Crossentropy*), y el algoritmo para calcular el gradiente, *Adam*.

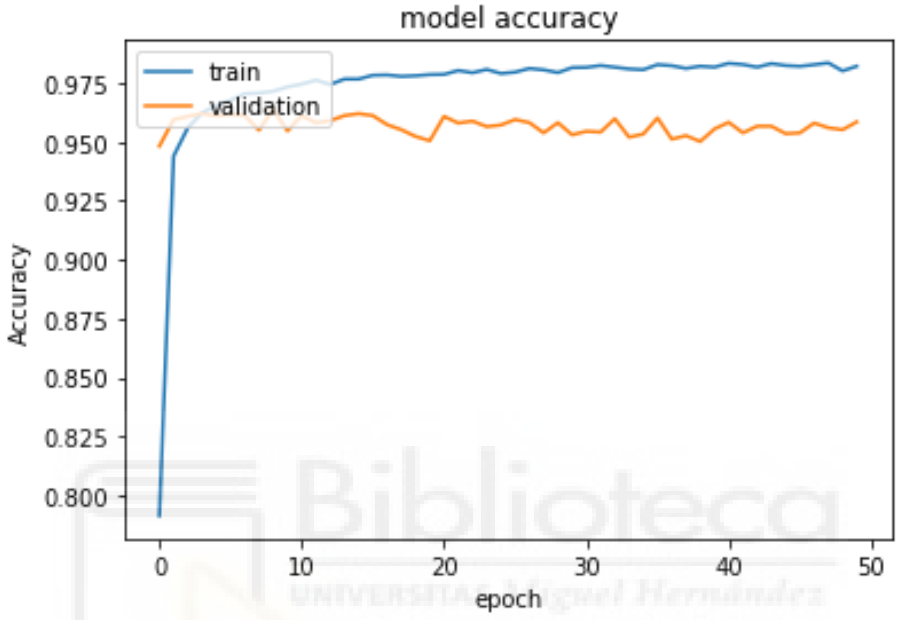


Figura 79. Accuracy del modelo

Como vemos en la figura 79 durante el entrenamiento hemos dividido el *dataset* en dos *sub-datasets*: *train y validation*. En *train set* hemos obtenido una precisión de casi 98% mientras que en la de *validation* 96%. Para ver que el modelo no es *overfitted* hemos creado otra partición llamada *test*, la cuál hemos evaluado una vez que el modelo termina la fase de entrenamiento. Sobre dicho *dataset* hemos obtenido una precisión de 95%.

5.- Conclusiones

En este trabajo, el objetivo principal era aprender la mecánica según la cuál funcionan las redes neuronales. Hemos empezado explicando lo que es el *Machine learning* y sus distintas ramas, a continuación, vimos que las redes neuronales pertenecen a un tipo de *machine learning* llamado *supervised learning*. En este tipo de aprendizaje, al modelo se le proporcionan inputs (ejemplos) y outputs (salidas), y seguidamente el modelo aprende a representar la relación entre inputs y outputs mediante una función, la cuál luego se utiliza para hacer las predicciones.

Las redes neuronales fueron inspiradas por las neuronas biológicas. En el apartado de introducción a las redes neuronales vimos que las éstas necesitan la no-linealidad para clasificar *datasets* complejos, y dicha no-linealidad es conseguida mediante una función de activación. En el *forward propagation*, estudiamos como pasar los inputs desde la capa inicial hasta la capa final. Se demostró que dicho proceso se agiliza utilizando la multiplicación matricial. Una vez aprendido el paso de *forward propagation*, explicamos el algoritmo de *Gradient Descent*, que es crucial para optimizar los parámetros de las redes neuronales.

Para entrenar los *weights* de una red neuronal, en el apartado de *Backpropagation*, comprendimos como utilizando la regla de cadena podíamos calcular las derivadas de una función respecto a los parámetros, los cuales no tenían influencia directa en dicha función. En el *backpropgation*, vimos que el gradiente sólo no era suficiente para optimizar los pesos, sino que necesitábamos otro parámetro que era *learning rate*. Este nos sirve para controlar los pasos en la dirección del gradiente y no tener problemas de convergencia.

Por otro lado, comprendimos que la función de activación *sigmoid* presentaba problemas cómo el *vanishing gradient*, dónde las capas iniciales dejan de optimizar sus parámetros. Para solucionar dichos problemas vimos las distintas

funciones de activación cómo *ReLu* o *Softmax*. Tanto en las redes neuronales como en otros algoritmos de *Machine learning*, pueden haber problemas de sobreajuste (*Overfitting*). Entonces en el aparatado de regularización estudiamos algunos métodos cómo *L1* o *L2 regularization*, y *early stopping*, para afrontar esos problemas.

Una vez comprendida la base sobre la que están constituidas las redes neuronales, vimos las distintas arquitecturas que existen y en qué campo se utilizan. En la arquitectura *convolutional* estudiamos cómo las imágenes, consistentes en matrices de pixeles, se les aplican distintos filtros para reducir sus tamaños, pero preservando las características esenciales. Una vez aplicado los filtros, se aplanan (*flattening*) las matrices para ser transferidas a una red neuronal *multiperceptron (MLP).*

Por último, en la práctica vimos como una red neuronal se aplica a distintos tipos de problemas como pueden ser regresión o clasificación. En la parte práctica, en el ejemplo de clasificación de dos clases ajustamos varios modelos. En uno de ellos, que era el modelo complejo con muchas capas, se producía el sobreajuste(*overfitting*). En otros dos que dibujaban una frontera adecuada, en uno se utilizó la función de activación *sigmoid*, mientras que en el otro aplicamos *ReLu* y observamos que el modelo que utilizó *ReLu*, tardó menos tiempo en optimizar sus parámetros. En el ejemplo de *X-OR Gate* se demostró la diferencia entre *Gradient Descent* y *Stochastic Gradient Descent*, y este último producía oscilaciones antes de llegar al mínimo. En el ejemplo de clasificación de dígitos, el modelo de red neuronal nos proporcionó una precisión de 96% demostrando la potencia de estos modelos.

5.- Bibliografía

- [1] Make your own Neural Network by Tariq Rashid (2016 Kindle)

- [2] Programming Machine Learning by Paolo Perrotta (2020 Pragmatic Bookshelf)
- [3] Grokking Deep Learning by Andrew Trask (Manning Publications 2019)
- [4] Deep Learning with Python by Francois Chollet (Manning Publications 2018)
- [5] Deep Learning Illustrated by Grant Beyleveld, Jon Krohn, Aglaé Bassens (Addison Wesley 2019)

6.- Anexos

Regression code :

```
# temp vs helados
import numpy as np
import matplotlib.pyplot as plt
x=np.array(
[[25],[26],[27],[28],[29],[29.5],[30.5],[31],[32],[33.5],[34],[36.5],[
37],[37.5],[39],[40]] )
y=
np.array([[10],[18],[12],[20],[18],[20],[22],[30],[32],[30],[35],[36],
[40],[38],[37],[41]] )

import matplotlib.pyplot as plt
plt.figure(figsize=(5,5))
plt.scatter(x,y)

plt.xlabel("Temp°")
plt.ylabel("Helados vendidos")
plt.title("Regression")
plt.show()


x=x/10
y=y/10
bias=np.ones(len(x))
x=np.insert(x,0,bias,axis=1)

lr=0.01
epoch=1500
np.random.seed(714)
weights=np.random.random((2,1))*3
steps = [[0, 0, 0]]
for e in range(epoch):
    output=np.dot(x,weights)
    error=np.sum((output-y)**2)
```

```
    steps.append([float(weights[1]),float(weights[0]),error])

    # backprop
    update= np.dot((output-y).T,x)

    weights -= lr*update.T

    print(error)

steps=np.array(steps)
weightz=steps[:,0]
biaz=steps[:,1]
lozz=steps[:,2]


linea=np.linspace(23,43,100)
linea=linea/10
linea=linea.reshape((100,1))
linea.shape
biasl=np.ones(len(linea))
linea=np.insert(linea,0,biasl,axis=1)


pred=np.dot(linea,weights)



x=np.array(
[[25],[26],[27],[28],[29],[29.5],[30.5],[31],[32],[33.5],[34],[36.5],[
37],[37.5],[39],[40]] )
y=
np.array([[10],[18],[12],[20],[18],[20],[22],[30],[32],[30],[35],[36],
[40],[38],[37],[41]] )
x=x/10
y=y/10
plt.figure(figsize=(5,5))
plt.scatter(x,y)
plt.plot(linea,pred)
plt.xlabel("Temp°")
plt.ylabel("Helados vendidos")
plt.title("Regression")
plt.xlim(2.0, 4.5)
plt.show()



%matplotlib

steps=np.array(steps)
weightzs=steps[:,0]
biazs=steps[:,1]
lozzs=steps[:,2]


x=np.array(
[[25],[26],[27],[28],[29],[29.5],[30.5],[31],[32],[33.5],[34],[36.5],[
37],[37.5],[39],[40]] )
y=
np.array([[10],[18],[12],[20],[18],[20],[22],[30],[32],[30],[35],[36],
[40],[38],[37],[41]] )
x=x/10
```

```
y=y/10
bias=np.ones(len(x))
x=np.insert(x,0,bias,axis=1)

b1 = np.linspace(-5, 3, 2000)
w1 = np.linspace(-1, 4, 2000)

B1,W1  = np.meshgrid(b1,w1)


losses=[]
for b, w in zip(np.ravel(B1), np.ravel(W1)):
    weightz=[[b],[w]]
    predz=np.dot(x,weightz)
    error=np.sum((predz-y)**2)
    losses.append(error)

losses=np.array(losses)
L = losses.reshape((2000,2000))


import seaborn as sns
from matplotlib import cm
from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d import Axes3D
sns.set(rc={"axes.facecolor": "white", "figure.facecolor": "white"})
ax = plt.figure().gca(projection="3d")


ax.set_xlabel("Weight", labelpad=20, fontsize=30)
ax.set_ylabel("Bias", labelpad=20, fontsize=30)
ax.set_zlabel("Loss", labelpad=5, fontsize=30)
ax.plot_surface(W1, B1,L, cmap=cm.CMRmap,linewidth=0,
antialiased=True, color='black')
plt.plot(float(weights[1]),float(weights[0]),1.5533527212309681 ,"gX",
markersize=16)



plt.show()




plt.plot(weightzs[1], biazs[1], lozzs[1], "wo")
plt.plot(weightzs[500], biazs[500], lozzs[500], "wo")
plt.plot(weightzs[1:], biazs[1:], lozzs[1:], color="w",
linestyle="dashed")
```

classification 2 classes code :

```
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
```

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

from sklearn.datasets import make_moons
from matplotlib import pyplot
from pandas import DataFrame
from keras.utils import to_categorical


X, y = make_moons(n_samples=250, noise=0.25)


model = Sequential()
model.add(Dense(8,activation='relu' ,input_shape=
(2,)))

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='Adam',loss='binary_crossentropy',metrics=['ac
curacy'])
model.summary()

model.fit(X, y,
batch_size=32, epochs=3000,
verbose=1)

%matplotlib inline
from matplotlib.colors import ListedColormap

minx, maxx = X[:, 0].min() - 1, X[:, 0].max() + 1
miny, maxy = X[:, 1].min() - 1, X[:, 1].max() +1

xgrid = np.arange(minx, maxx, 0.01)
ygrid = np.arange(miny, maxy, 0.01)

meshx,meshy = np.meshgrid(xgrid, ygrid)
inputsx = np.array([meshx.ravel(), meshy.ravel()]).T
pred=np.round(model.predict(inputsx))

Z_pred = pred.reshape(meshx.shape)

plt.figure(figsize=(10,7))


plt.contourf(meshx, meshy, Z_pred,
                     alpha = 0.25,
                     cmap = ListedColormap(( '#FF0000','#0000FF')))



clase1 = np.where(y == 0)
clase2 = np.where(y == 1)
plt.scatter(X[clase1,0],X[clase1,1], c='r',
label='clase1',linewidth=1,edgecolor='black')
plt.scatter(X[clase2,0],X[clase2,1], c='b',
label='clase2',linewidth=1,edgecolor='black')
```

XOR code :

```
import numpy as np

epoch=4000
x=np.array( [[0,0],[0,1],[1,0],[1,1]] )
y= np.array( [[0],[1],[1],[0]])
lr=1
np.random.seed(1685)


hidden_nodes=3
wih= np.random.random((2,hidden_nodes))
who= np.random.random((hidden_nodes,1))

losses=[]
for e in range(epoch):
    error=0
    z1=np.dot(x,wih)
    a1=sigmoid(z1)
    z2=np.dot(a1,who)

    a2=sigmoid(z2)
    error=np.sum( (a2-y)**2)

    # back

    da2=a2-y
    dz2=sigmoidderiv(a2)
    dwho=a1
    updatewho=np.dot((da2*dz2).T,dwho)
    da1=who
    dz1=sigmoidderiv(a1)
    dwih=x
    updatewih=np.dot((np.dot(da2*dz2,who.T)*dz1).T,dwih)


    who -= lr*updatewho.T
    wih -= lr*updatewih.T


    print(error)
    losses.append(error)


import matplotlib.pyplot as plt
plt.figure()
plt.plot( losses)

plt.xlabel("epoch")
plt.ylabel("error")
plt.title("training lr 1")

plt.show()


#SGD
np.random.seed(2564)
losses=[]
losses2=[]
hidden_nodes=5
```

```
matriz=np.array( [[0,0,0],[0,1,1],[1,0,1],[1,1,0]] )
np.random.shuffle(matriz)
x=matriz[:,0:2]
y=matriz[:,2].reshape((4,1))


wih= np.random.random((2,hidden_nodes))
who= np.random.random((hidden_nodes,1))

lr=1
epoch=2000
for e in range(epoch):
    zrror=0
    matriz=np.array( [[0,0,0],[0,1,1],[1,0,1],[1,1,0]] )
    np.random.shuffle(matriz)
    x=matriz[:,0:2]
    y=matriz[:,2].reshape((4,1))
    for i in range(len(x)):
        error=0
        z1=np.dot(x[i:i+1],wih)
        a1=sigmoid(z1)
        z2=np.dot(a1,who)

        a2=sigmoid(z2)
        error=(a2-y[i:i+1])**2

    # back

        da2=a2-y[i:i+1]
        dz2=sigmoidderiv(a2)
        dwho=a1
        updatewho=np.dot((da2*dz2).T,dwho)
        da1=who
        dz1=sigmoidderiv(a1)
        dwih=x[i:i+1]
        updatewih=np.dot((np.dot(da2*dz2,who.T)*dz1).T,dwih)


        who -= lr*updatewho.T
        wih -= lr*updatewih.T


        print(error)
        zrror=zrror+error
        losses.append(float(error))
    losses2.append(float(zrror))


import matplotlib.pyplot as plt
plt.figure(figsize=(12, 12))
plt.plot( losses2)

plt.xlabel("epoch")
plt.ylabel("error")

plt.xlim(0, 2000)
plt.show()



plt.figure(figsize=(12, 12))
```

```
plt.plot( losses2)

plt.xlabel("epoch")
plt.ylabel("error")

plt.xlim(0, 1000)
plt.show()
```

Multiclass Classification, MNIST Dataset code :

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot as plt
from keras.regularizers import l2

(X_train, y_train), (X_valid, y_valid) = mnist.load_data()
X_train.shape

y_train.shape

plt.figure(figsize=(15,15))
for k in range(50):
    plt.subplot(2,25,k+1)

    plt.axis('off')
    plt.imshow(X_train[k], cmap='Greys')

plt.show()

plt.imshow(X_valid[0], cmap='Greys')

plt.show()



X_train = X_train.reshape(60000, 784).astype('float32')
X_valid = X_valid.reshape(10000, 784).astype('float32')

X_train /= 255
X_valid /= 255

n_classes = 10
y_train = keras.utils.to_categorical(y_train, n_classes)
y_valid = keras.utils.to_categorical(y_valid, n_classes)


model = Sequential()
model.add(Dense(64, activation='relu', input_shape=
(784,),activity_regularizer=l2(0.001)))
model.add(Dense(32, activation='relu', input_shape=
(784,)))
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='Adam',loss='categorical_crossentropy',metrics
=['accuracy'])
model.summary()
```

```
model.fit(X_train, y_train,
batch_size=128, epochs=50,
verbose=1,
validation_split=0.1)

import matplotlib.pyplot as plt
import numpy
plt.plot(model.history.history['accuracy'])
plt.plot(model.history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()

model.evaluate(X_valid,y_valid,verbose=0)
```