Curso de especialización en Inteligencia Artificial y Big Data

UD05. DESARROLLO DE APLICACIONES DE IA: REDES NEURONALES

Carlos Sáenz Adán

Tabla de contenido

1. Modelado de redes neuronales

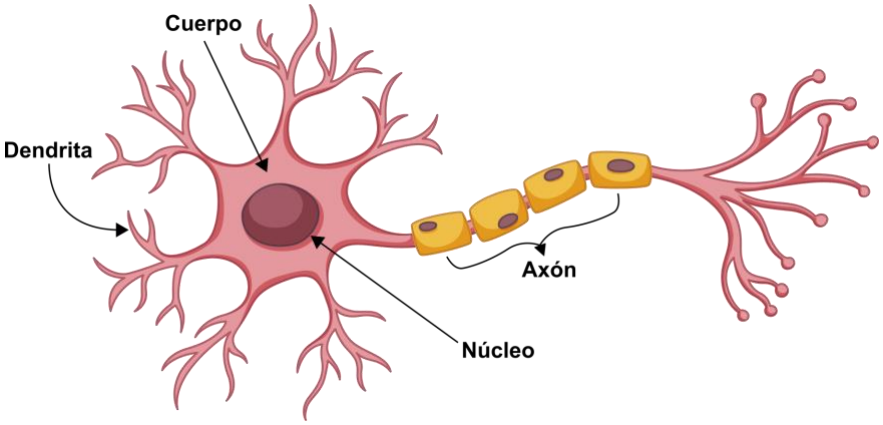
En esta sección veremos estos tres conceptos clave respecto a la programación de redes neuronales:

- Modelos de tipo Sequential, sus tipos de capas y los parámetros de éstas.
- Principales protagonistas del entrenamiento del modelo: función de coste y optimizadores.
- Otros parámetros de gobierno del entrenamiento como las iteraciones epochs o el número de muestras de entrenamiento (batch_size).

1.1. Fundamentos básicos

Las redes neuronales, independientemente de que su origen sea biológico o artificial, se constituyen como un sistema compuesto de neuronas conectadas entre sí que reciben señales y las transmiten entre ellas. Las neuronas son la unidad básica de este esquema, constituidas por un cuerpo celular y ramificaciones. Solo como curiosidad, esta unidad simple de procesamiento se encuentra la mayor parte del tiempo a la espera de señales procedentes de las ramificaciones que conectan a una con otra neurona.

En esencia, la neurona está formada por un cuerpo o soma donde se encuentra el núcleo. Las ramificaciones que sirven de entrada de información a las neuronas se denomina dendritas. Por el contrario, la señal saliente de la red neuronal proviene del axón, una ramificación más alargada que la dendrita. Las conexiones entre neuronas es un mecanismo que se denomina sinapsis y se realiza uniendo el axón a una o varias dendritas.



Las neuronas emiten señales que se propagan de una a otra mediante una compleja reacción electroquímica, las cuales controlan la actividad del cerebro a corto plazo. Por el contrario, estas señales también habilitan la conectividad de las neuronas, a largo plazo. Estos mecanismos muy elaborados son la base del aprendizaje. La mayor parte del procesamiento de la información tiene lugar en la corteza cerebral, la capa externa del cerebro.

Una red neuronal artificial no busca imitar el comportamiento de su equivalente biológica. Al contrario, se ignoran la mayoría de los mecanismos internos de las neuronas porque replicar su comportamiento no es en absoluto necesario para satisfacer el objetivo deseado. A modo de ejemplo, cualquier artefacto capaz de navegar, ya sea bajo o sobre el agua, no está construido siguiendo fielmente la estructura de un pez, es decir, aletas y branquias. Algo similar ocurre con las redes neuronales artificiales, dado que el objetivo no es simular

el sistema biológico, sino imitar, de algún modo, el procesamiento del cerebro humano para añadir comportamiento inteligente a sistemas con cerebros artificiales.

Obviamente, los cerebros biológicos y los ordenadores presentan propiedades totalmente diferentes. Por un lado, una red neuronal, cuyo origen no sea artificial, es capaz de procesar, y simultáneamente, información en cada una de las unidades básicas de su estructura, es decir, las neuronas. Por el contrario, los computado- res disponen de dos componentes para replicar este comportamiento, haciendo uso de un cerebro artificial, denominado CPU que procesa información, y de una memoria que guarda dicho resultado. Por lo tanto, la gran diferencia existente es que las neuronas biológicas son capaces de procesar información por su cuenta. En otras palabras, las neuronas tratan grandes cantidades de información paralelamente, mientras que una CPU realiza este cometido de uno en uno.

No obstante, inspirado en el procesamiento paralelo, las siguientes secciones presentan la arquitectura de una red neuronal artificial capaz de procesar simultáneamente numerosos datos, replicando a alto nivel el comportamiento de una red neuronal biológica.

1.2. Elementos básicos de una Red Neuronal Artificial

En el siguiente vídeo, veremos una introducción a lo que son las redes neuronales donde se comienzan explicando las redes neuronales naturales y su analogía con las redes neuronales artificiales.

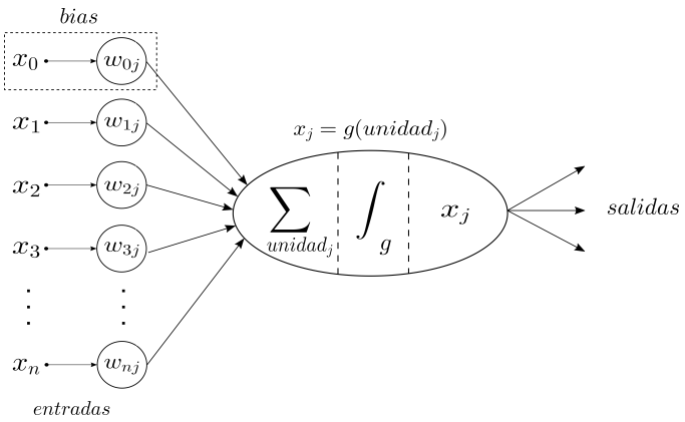
https://www.youtube.com/watch?v=CU24iC3grq8

El perceptrón

Una red neuronal artificial, al igual que una red biológica real, está compuesta de nodos o unidades denominadas neuronas artificiales o **perceptrones**. Cada una está conectada por un enlace directo al cual se le asigna un peso numérico que determina la influencia de la conexión sináptica entre las neuronas. Estos pesos se utilizan como multiplicadores de las entradas de la red neuronal, resultando en una suma ponderada del producto de los pesos y entradas.

Por ejemplo, si contamos con una red neuronal compuesta de 5 entradas x1, x2, x3, x4, x5, también disponemos de 5 pesos asociados a cada entrada w1, w2, w3, w4, w5. Además, se suele añadir un término extra denominado sesgo o «bias», una medida que indica lo fácil que es disparar la salida de la neurona. Como resultado, la salida en este paso consiste realizando la siguiente suma ponderada: $x0 \times w0 + x1 \times w1 + x2 \times w2 \times \cdots \times x5 + w5$. Una vez computado el resultado, la neurona efectúa una operación más, aplicando una función de activación cuyo término es prestado de la neurociencia, la cual determine la salida de la neurona. En otras palabras, una predicción o decisión a partir de las entradas proporcionadas a la red.

Para dotar de inteligencia a esta estructura, utilizando el símil de la red neuronal biológica, el aprendizaje resulta de la modificación de la influencia de las conexiones sinápticas entre neuronas. Esto se consigue visualizando la red neuronal artificial como un sistema dinámico que modifica los valores de sus pesos. De un modo simplificado, este es el funcionamiento de una red neuronal artificial. Veamos a continuación los detalles técnicos de su estructura.



La figura muestra el esquema general de un elemento mínimo de procesamiento en una red neuronal (el perceptrón). En esencia, una red cuenta con numerosas entradas (neuronas) y, en base a su simplicidad o complejidad, una o varias salidas. La salida x_j representa la activación de la entrada x_i y el peso w_{ij} en el enlace desde la unidad i hasta la unidad j. El valor de la red a la unidad j se puede definir como:

$$unidad_j = \sum_{i=0}^{n} x_i w_{ij}$$

Nótese que el número de interconexiones en una red determina indudablemente la velocidad en la cual se computa esta operación.

Una vez resuelto el valor de la combinación entre los valores de la entrada y los pesos, se le aplica una función de activación *g* para resolver la salida, es decir, la predicción o decisión como consecuencia de las entradas introducidas en la red. Esta función se denota como:

$$x_j = g(unidad_j) = g(\sum_{i=0}^{n} x_i w_{ij})$$

La función de activación *g* trata de imitar el mecanismo de los impulsos eléctricos en la comunicación entre dos o más neuronas. Este comportamiento se intenta replicar, de algún modo, con un umbral mediante la **función escalón** (véase *parte a* de la siguiente Figura) o con una versión suavizada usando la **función sigmoide** (véase *parte b* de la siguiente Figura). En ocasiones poco frecuentes se utiliza también la función identidad, cuyo objetivo es no hacer nada y producir la combinación lineal. Es por ello que rara vez se utiliza esta función en redes neuronales, dado que su comportamiento no conduce a nada nuevo.

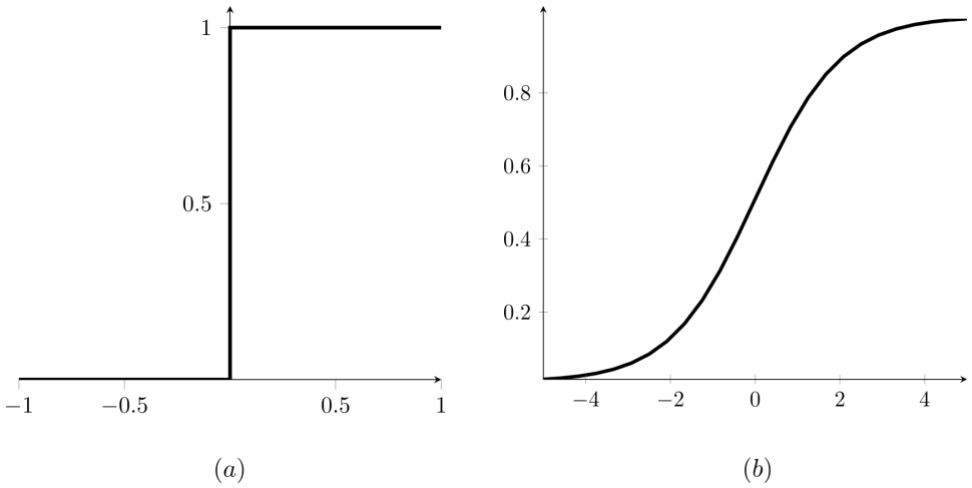


Figura 1. Funcion escalón y sigmoide

La función **escalón** (Parte a de Figura 1) produce una salida de 1 cuando el resultado de la combinación de entradas y pesos es mayor de 0. De lo contrario, su salida será 0. Como es de esperar, una salida con valor 1 significa que la unidad de salida se activa.

$$\theta(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$$

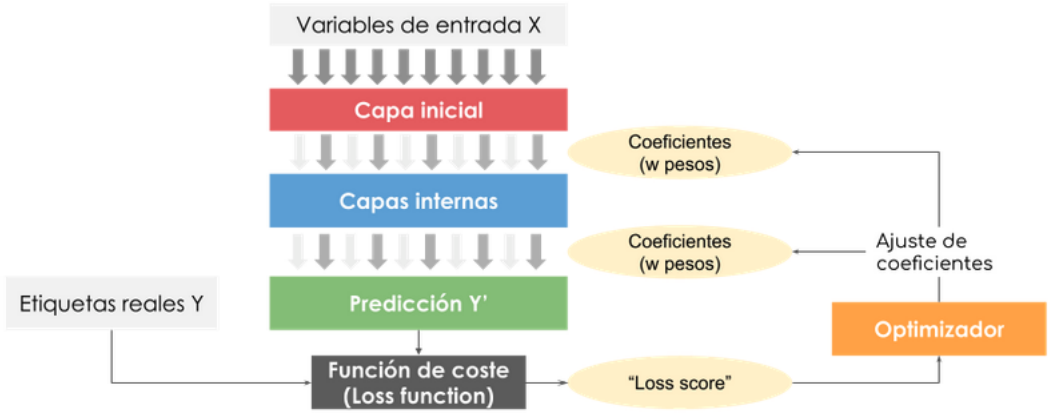
Por el contrario, la función **sigmoide** (Parte b de Figura 1) es una modificación suavizada de la función escalón que produce salidas entre 0 y 1. El resultado de las salidas se puede interpretar como probabilidades, lo cual es útil para predecir cómo de probable es que ocurra algo, en lugar de utilizar valores numéricos enteros. El uso de esta función no lineal se emplea para mejorar el aprendizaje de los pesos y el sesgo de la red. Al realizar pequeñas modificaciones en ambos elementos se produce un ligero cambio en la salida, resolviendo los valores adecuados para una tarea concreta, a diferencia de valores enteros que pueden cambiar el comportamiento del resto de la red.

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

No obstante, es imprescindible destacar que la convención a usar (salidas con números enteros o reales) viene determinada por el tipo de problema a resolver. En otras palabras, no existe una función que sea la «panacea» ante la resolución de un problema cualquiera.

Las capas

Una red neuronal va a estar compuesta de varios perceptrones conectados entre sí, apilados en capas. En Keras, la clase a la que tenemos que llamar para crear un modelo con varias capas será "**Sequential**". A lo largo de esta sección, siempre empezaremos así la construcción del modelo.



Carmen Bartolomé (CC BY-SA)

La capa o "layer" de tipo Dense, es la que representa verdaderamente una red neuronal con todos los nodos conectados con las diferentes variables de entrada, en mayor o menor medida según el valor del coeficiente de cada conexión. En la imagen superior, hay dos capas (con sus correspondientes perceptrones) que operan con combinaciones de las variables de entrada. A esto, se le llama red "densamente" conectada.

La clase Dense, crea un objeto que implementa la operación

output = activation(dot(input, kernel) + bias)

donde:

- *activation* es la función de activación que actúa sobre el resultado que se da en cada neurona.
- *kernel* es la matriz de coeficientes o pesos "w" que se generan de forma aleatoria
- *bias* es el vector del sesgo que solo es distinto de cero si se pasa como argumento True.

1.3. Definición de redes neuronales utilizando Keras y TensorFlow

Toda la información que aparece en esta sección se ilustra con la implementación en Python utilizando la librería Keras de Tensorflow. Todo el código lo puedes encontrar en el documento compartido del siguiente enlace: https://colab.research.google.com/drive/1xF4Xcc7XKGfYNGTwLPbxrHXWk4T6yx-c

La clase Dense implementa la operación `output = activation(dot(input, kernel) + bias)` que hemos visto anteriormente. Pero además presenta números parámetros, de los que se explicarán los que se consideran más importantes en este momento del aprendizaje. Estos parámetros, entre otros, se pueden encontrar en el esquema:

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
```

```
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Units

El parámetro **units** se refiere al número de neuronas que debe tener la capa. Debe ser un número entero y positivo.

Activation

La función de **activación**, en realidad, es opcional. Si no se indica una, no se aplicará activación y el resultado de la neurona será directamente el cálculo lineal. Para problemas lineales, no hay inconveniente, pero si necesitamos que el modelo genere una solución de tipo no lineal (curvas o superficies curvas), es imprescindible aplicar funciones de activación, que irán "dibujando" esa superficie a costa de encender y apagar los nodos de cálculo que son las neuronas según vayan dando valores que ayuden a tener el resultado final correcto.

Por ejemplo, para definir un modelo de tipo DNN (Deep Neural Network) o red neuronal profunda, con dos capas de red neuronal, el código podría ser:

```
model= keras.models.Sequential()

model.add(keras.layers.Dense(32, activation='relu', input_shape=(12,)))
model.add(keras.layers.Dense(32))
```

En este ejemplo, hay 12 variables de entrada. Hay dos capas tipo red neuronal, ambas con 32 neuronas. La primera, tiene como función de activación, la ReLu, mientras que la segunda, no tiene función de activación.

Input_shape

La capa de entrada debe tener la misma forma que tus datos de entrenamiento. Si tienes 30 imágenes de 50x50 píxeles en RGB (3 canales), la forma de los datos de entrada es (30,50,50,3). Por lo tanto, la capa de entrada debe tener esta forma. En el caso de capas Dense requiere una entrada de la forma (batch_size, input_size).

En algunas ocasiones, se recurre a **la capa de tipo Flatten**. Es una capa que "aplana" una estructura de datos de entrada de más de una dimensión, para que tengamos un vector, o array de una dimensión, que es lo que aceptan las capas de redes neuronales. En el caso de trabajar con imágenes, lo normal es tener, como datos de entrada, una serie de matrices o arrays de N x N pixels. Por ejemplo, si tenemos un dataset con 1000 imágenes de 12 x 12 pixeles, la estructura de datos de entrada o dataset.shape sería: (1000, 12, 12). Al aplicar la capa Flatten, obtenemos una estructura de salida (1000, 144). Para este ejemplo, el código sería:

```
import keras
```

```
model = keras.Sequential()
model.add(keras.layers.Flatten(input_shape = (12,12)))
model.add(keras.layers.Dense(64, activation = 'relu'))
model.add(keras.layers.Dense(1,activation = 'sigmoid'))
```

Inicialización del kernel

El algoritmo de retropropagación requiere un punto de partida en el espacio de posibles valores de los pesos. La idea de utilizar un enfoque no determinista frente a uno determinista se resume en el rendimiento del algoritmo, fundamentalmente porque la segunda opción resulta ineficiente para problemas complejos.

Por normal general, los pesos no deben inicializarse al valor 0, ya que esto imposibilita que el algoritmo de ajuste busque de forma efectiva los valores correctos para realizar adecuadas predicciones. Históricamente, la inicialización de pesos se ha venido definiendo mediante heurísticas simples, como por ejemplo:

- Pequeños valores aleatorios entre [0, 1]
- Pequeños valores aleatorios entre [-1, 1]
- Pequeños valores aleatorios entre [-0.1, 0.1]
- Otras similares.

En general, la inicialización de los pesos se realiza siguiendo una distribución gaussiana o uniforme. Sin embargo, la escala de la distribución inicial tiene un gran efecto tanto en el resultado del procedimiento de optimización como en la capacidad de generalización de la red. En definitiva, la inicialización de los pesos afecta considerablemente en el proceso de aprendizaje de la red. Estas técnicas continúan funcionando en general. Sin embargo, se han desarrollado nuevos enfoques que se han convertido en la norma de facto, dado que pueden dar lugar a un entrenamiento del modelo muchos más optimizado.

En esencia, estas nuevas heurísticas se clasifican en función del tipo de función de activación utilizada, principalmente destinadas a la función sigmoide, tangente hiperbólica y ReLU.

Debes saber…

El método utilizado para inicializar los pesos de una red con funciones de activación sigmoide o tangente hiperbólica se denomina **inicialización de Glorot** o **inicialización de Xavier.**

Por otro lado, la estrategia de inicialización para la función de activación ReLU (y sus variantes, incluida la activación ELU) se denomina **inicialización He**.

Estos inicializadores los puedes encontrar dentro del modulo tf.keras.initializers. También puedes acceder a este enlace para ver más información sobre los inicializadores.

https://faroit.com/keras-docs/2.0.6/initializers/

Se puede definir de dos formas, (1) a través de su nombre y (2) mediante un objeto de esa clase

```
(1)   Dense(64, kernel_initializer='he_uniform')
(2)   Dense(64, kernel_initializer=tf.keras.initializers.he_uniform)
```

```
(3)    Dense(64, kernel_initializer= tf.keras.initializers.HeUniform())
```

Número de neuronas

Una de las dudas más corrientes es sobre el número de neuronas a definir en cada capa. En los problemas de clasificación, hay un criterio muy claro:

- Si es clasificación binaria, la capa de salida tendrá una única neurona
- Si es clasificación múltiple, la capa de salida debe tener tantas neuronas como clases o categorías de clasificación tenga el problema.

Pero no hay un criterio claro para las capas internas. Hay algunos planteamientos matemáticos que pueden ayudar, pero la tendencia es, precisamente, probar mucho e ir adquiriendo experiencia que aporte la intuición necesaria para hacer una primera estimación que nos aporte un buen modelo lo antes posible.

Te recomendamos que empieces por configuraciones muy sencillas, con un número de neuronas dentro del orden de magnitud de las variables de entrada. Después, podrás ir probando a aumentar capas y neuronas en diferentes configuraciones.

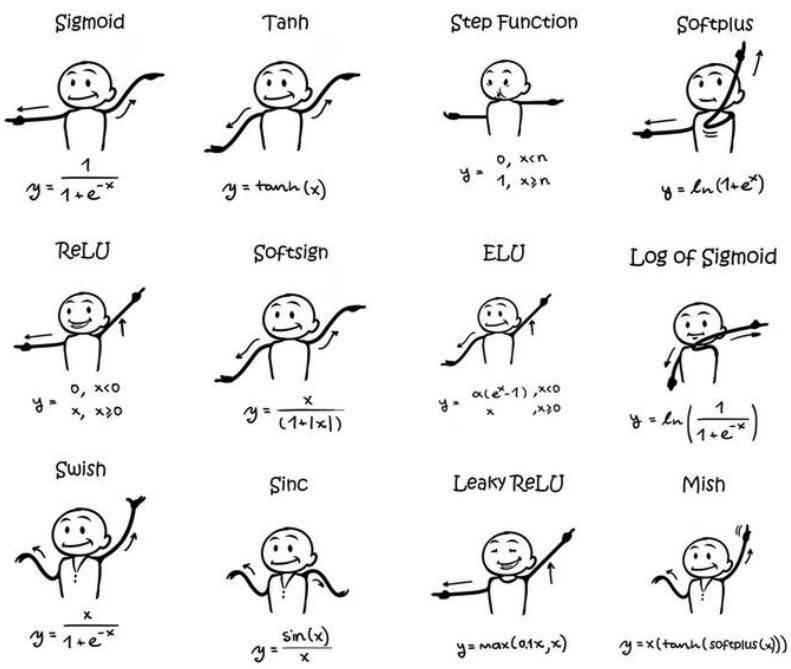
En todo caso, lo más útil es aprender de los modelos que otros ya han creado y probado.

Funciones de activación

Como introducción a esta sección, puedes visualizar el siguiente vídeo sobre las funciones de activación: https://www.youtube.com/watch?v=_0wdproot34 En este video puedes ver las características principales de estas funciones y sus ventajas e inconvenientes.

Hay muchas posibles funciones de activación, pero se suele utilizar siempre una de estas:

- ReLu
- Sigmoid
- Softmax
- Tanh



El criterio para aplicar una u otra, reside en su comportamiento matemático, pero no hay un criterio absoluto. En general, lo recomendable es partir de una configuración básica e ir probando con cambios controlados. Algunos consejos sobre dicha configuración básica serían:

- Utiliza ReLu para capas internas u ocultas
- En la capa de salida, si estás en un problema de clasificación binaria, utiliza Sigmoid
- En la capa de salida de un problema de clasificación múltiple, utiliza Softmax
- En capa de salida, si te encuentras en un problema de regresión utiliza ninguna, ReLu o tanh

Para saber más...

Sobre las funciones de activación, puedes leer el análisis que hace B. Chen en este artículo. Y sobre la función Softmax, también tienes un artículo muy descriptivo y completo que te puede orientar mejor.

1.4. Compilar Red Neuronal Artificial

La parte de nuestro algoritmo que se encarga de configurar cómo va a ser el entrenamiento, viene controlada por la función "compile". Echemos un vistazo a la línea de código de un posible algoritmo para clasificación:

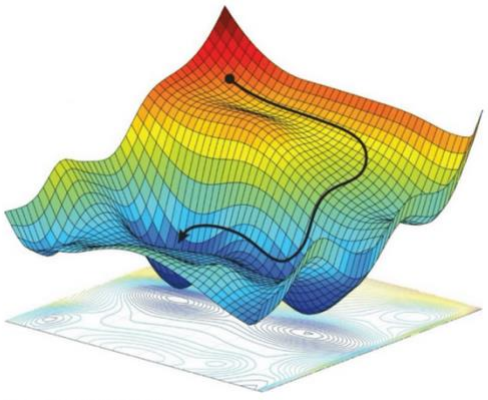
```
model.compile(optimizer= 'Adam',
              loss = 'sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

A continuación, veremos los diferentes parámetros que existen a la hora de configurar la compilación de una red neuronal, y que fijarán la estrategia a seguir durante su entrenamiento.

Tipos de función de coste (Loss)

El parámetro loss, representa lo que llamamos "función de coste" o "función de pérdida" y es una métrica necesaria para que el proceso de ajuste de los coeficientes o pesos de las redes neuronales en las capas de nuestro modelo, se vayan actualizando adecuadamente hacia el modelo definitivo ya entrenado.

Esta figura, representa lo que podría ser la superficie generada por la función Loss en un problema de dos variables. Los distintos puntos de esta superficie dependen del estado del modelo, en función de los valores de los pesos o coeficientes que vamos recorriendo en los ejes. El entrenamiento del modelo, básicamente, consiste en buscar los valores de los pesos que corresponden al valor mínimo de esta superficie. Con cada iteración, nos vamos moviendo por ella, calculando los distintos valores del error hasta encontrar el menor de ellos.



En keras, tenemos disponibles varias funciones de coste o "loss functions" adecuadas para nuestros modelos. Vamos a repasar las más utilizadas en deep learning, identificando los casos en los que utilizar cada una de ellas. En general, para los modelos basados en neuronas, es necesario utilizar lo que se conoce como

cálculo de la "entropía cruzada", en contraposición al error cuadrático medio o MSE que se venía utilizando en otros modelos.

Las siguientes tres funciones hacen un cálculo del índice de error entre las clases reales dadas por las etiquetas y las predicciones que va dando el modelo. En realidad, se proporcionará una media de la pérdida o error de todas las instancias de la muestra en cada iteración (epoch) del proceso de entrenamiento. Pero cada una de ellas está programada para un tipo de problema de clasificación:

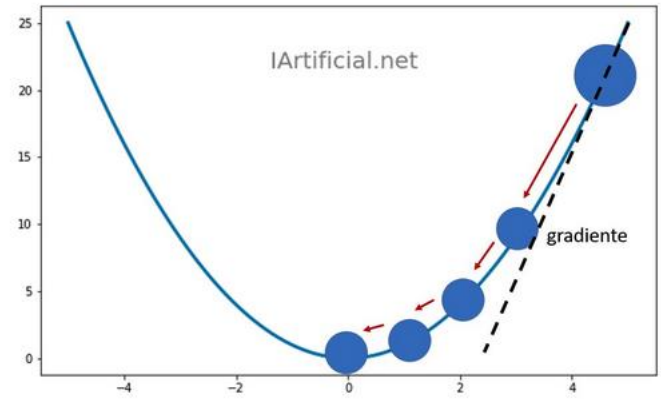
- **Binary Crossentropy**: es la función de coste indicada para trabajar con problemas de clasificación binaria, junto a la función de activación sigmoide.
- **Categorical Crossentropy**: adecuada para problemas de clasificación múltiple, pero con etiquetas o variables de salida de tipo categórico en formato one-hot encoding.
- **Sparse Categorical Crossentropy**: es la función de coste para problemas tanto binarios como múltiples, pero con las etiquetas o clases de salida dadas como números enteros.

En Keras puedes encontrar estas funciones en el módulo `tf.keras.losses`

Optimizadores

Para entrenar el modelo, necesitamos alcanzar la configuración de los pesos w en toda la red que hace que el error sea mínimo. El método matemático que se utiliza para buscar el valor mínimo de una función es la derivada, y, en un caso multidimensional, el gradiente, que se compone de las derivadas parciales de la función respecto a cada una de las variables. Cuando nos acercamos a un mínimo, la derivada, que nos da el valor de la pendiente de la tangente a la curva, se va haciendo más negativa. Esta técnica se denomina "descenso del gradiente", y los métodos de optimización se basan en favorecer esa evolución hacia el mínimo.

$$w_{k+1} = w_k - \eta \frac{1}{n} \sum_{i=t}^{t+n-1} \nabla f_{w_k}(x^i)$$

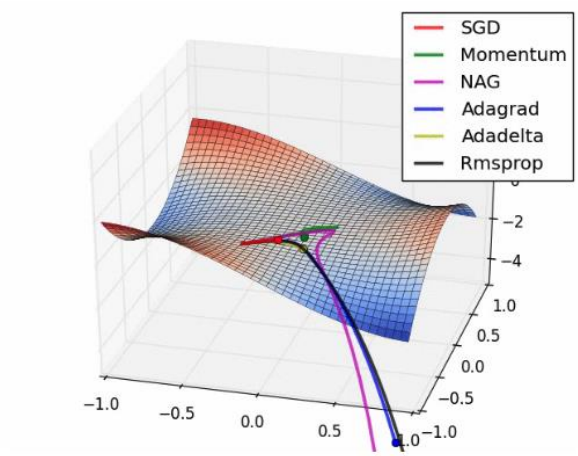


IArtificial.net (CC BY-SA)

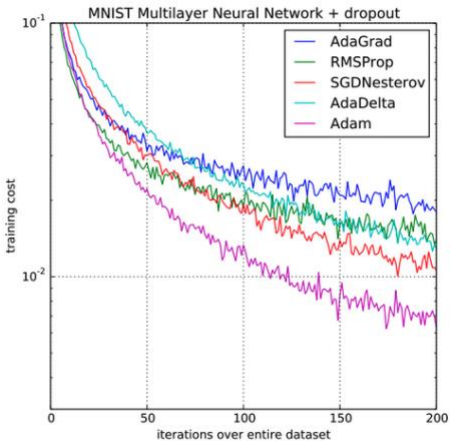
Como aproximación inicial al mundo de los optimizadores, te recomendamos que utilices, de momento, uno de estos dos: **RMSprop** o **Adam**. A medida que vayas aprendiendo más y adquiriendo experiencia, podrás ir explorando las posibilidades particulares que te ofrecen los demás. En estos gráficos, puedes comprobar el buen desempeño, en general, que tiene cada uno, pero si tenemos que caracterizar de alguna manera sus

ventajas, puedes considerar que:

- **RMSprop** presenta una convergencia hacia el mínimo más rápida
- **Adam** presenta un mejor comportamiento general. Es una buena opción en tus primeros entrenamientos.



Optimization Open Textbook (CC0)

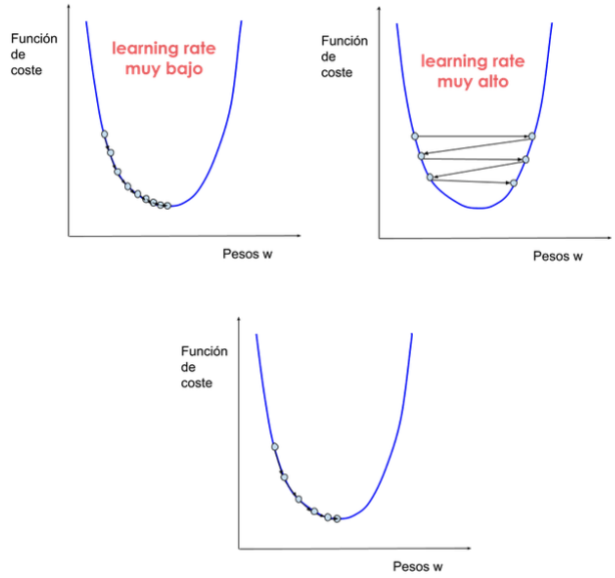


Optimization Open Textbook (CC0)

En Keras puedes encontrar estas funciones en el módulo `optimizers`

Ratio de aprendizaje (Learning rate)

Es el parámetro que controla la magnitud con la que modificamos los pesos en función de la pendiente de la función de coste. Podríamos decir que es un factor que amplifica el efecto de la pendiente de la función de coste. Si el ratio de aprendizaje es pequeño, aunque estemos en una zona de fuerte pendiente, se avanzará a pasos pequeños, y en la zona de baja pendiente, según vamos alcanzando el mínimo, cada vez avanzará mucho más despacio, lo que hará el entrenamiento muy lento. Si el learning rate, por el contrario, es demasiado alto, en la zona de pendiente muy elevada, hará pegar un salto al factor de corrección de los pesos, que nos iremos a la otra cara de la función de coste. El proceso de entrenamiento será muy inestable e incluso podría no converger. El rango de un learning rate de partida **está entre 0,0001 y 0,001**. A veces hay que tomar uno más alto o más bajo de los valores de ese rango, pero puedes empezar por ahí e ir probando.



En keras, al llamar a la función de optimización, podemos pasarle un valor de learning rate como argumento por clave:

```
opt = keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss='categorical_crossentropy', optimizer=opt)
```

Para saber más...

Para conocer mejor el uso de los optimizadores con keras, puedes consultar la documentación que hay en su API reference. También puedes leer este artículo en el que el autor hace un experimento probando diferentes optimizadores para varios casos.

1.5. Recomendaciones para la definición de la arquitectura

La definición de la arquitectura de una red neuronal es una tarea que requiere mucha experiencia, pero se pueden dar ciertas recomendaciones que se muestran en la siguiente tabla:

Hiperparámetro	Clasificación binaria	Clasificación binaria multietiqueta	Clasificación multiclase	**Regresión**
N.º de neuronas de entrada	Una por característica de entrada	Una por característica de entrada	Una por característica de entrada	Una por característica de entrada
N.º de capas ocultas	Normalmente entre 1 y 5	Normalmente entre 1 y 5	Normalmente entre 1 y 5	Normalmente entre 1 y 5
N.º de neuronas por capa oculta	Normalmente entre 10 y 100	Normalmente entre 10 y 100	Normalmente entre 10 y 100	Normalmente entre 10 y 100
Activación capa interna u oculta	ReLu	ReLu	ReLu	ReLu (o SeLu)
N.º de neuronas de salida	1	1 por etiqueta	1 por clase	
Activación capa de salida	Sigmoid	Sigmoid	Sofmax	Ninguna, ReLu, logística
Función de pérdida	Entropía cruzada	Entropía cruzada	Entropía cruzada	Error cuadrático medio

1.6. Entrenamiento de una red neuronal

Tras toda la configuración del modelo y de los parámetros que permitirán el entrenamiento del mismo, ya solo queda proceder a éste. Keras proporciona el método fit para el entrenamiento de un modelo basado en redes neuronales profundas. Un ejemplo de la aplicación de esta función sería:

```
model.fit(X_train, y_train, epochs = 20, batch_size = 40)
```

El método fit necesita tres parámetros ineludibles: los datos de entrada X, las etiquetas o datos de salida y finalmente, el número de epochs.

El parámetro **epochs** representa el número de iteraciones del entrenamiento. En cada iteración, tiene lugar el proceso por el cual el optimizador busca un mínimo de la función de coste. A base de hacer varias iteraciones con distintas muestras de datos, se va buscando un mínimo cada vez más "mínimo". ¿Recuerdas que queremos quedarnos con el punto de la función de coste que constituye el mínimo global?

Las muestras que se utilizan en cada iteración o epoch, se gestionan con el parámetro **batch_size**. Si no se indica nada, este parámetro toma el valor, por defecto, de 32 muestras. En el ejemplo, estamos indicando que queremos 40 muestras o lotes, lo cual implica que el conjunto total de datos se dividirá entre ese número de lotes. Por ejemplo, si tenemos 60.000 registros, entre 40 lotes, nos da 1500 registros por iteración. Esto quiere decir que, en cada epoch, la función de optimización irá fijando el valor de los pesos para minimizar el valor de la función de coste según el comportamiento de la red con esos 1500 casos de X y si se aciertan o no sus correspondientes etiquetas y.

A la hora de entrenar un modelo de red neuronal podemos indicar un conjunto de validación con el objetivo de reducir el sobreajuste. Para ello podemos utilizar el parámetro **validation_data** que es una tupla donde el primer elemento es el conjunto de características para realizar la predicción y el segundo elemento es la predicción que se debería hacer para el anterior conjunto.

```
model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val,y_val))
```

Aquí hay otra opción: el argumento **validation_split** le permite automáticamente parte de reserva de sus datos de entrenamiento para su validación. El valor del argumento representa la fracción de los datos a ser reservados para la validación, por lo que se debe establecer en un número mayor que 0 y menor que 1. Por ejemplo, **validation_split=0.2** medios de "uso 20% de los datos para la validación".

```
Model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_split=0.2)
```

2. Proyectos basados en redes neuronales

En el aula virtual tendrás acceso a 4 ejemplos de proyectos donde las redes neuronales son el clasificador utilizado. Es importante que veas las diferentes formas de definir la arquitectura, compilación y entrenamiento de las redes.

Proyecto 1. Conversión Celsius a Fahrenheit

En el siguiente enlace se muestra cómo desarrollar una primera red neuronal para la conversión de grados Celsius a Fahrenheit https://www.youtube.com/watch?v=iX_on3VxZzk.

Proyecto 2. Clasificador de imágenes

Tienes un ejemplo de un modelo para clasificación de imágenes, en el que puedes ver todos los elementos e hiperparámetros que hemos visto. Fíjate bien en qué función de coste se ha utilizado, y qué optimizador.

Proyecto 3. Clasificador comentarios películas

También te presento un ejemplo donde se utiliza una red neuronal con el objetivo de clasificar comentarios sobre películas.

Proyecto 4. Reconocimiento de dígitos

Otro ejemplo que presento es la identificación de dígitos utilizando redes neuronales. Puedes ver el ejemplo en el siguiente enlace.

3. Bibliografía

- Aurélien Géron. Aprende Machine Learning con Scikit-Learn, Keras y TensorFlow. O'Reilly.
- Aurélien Géron. Aprende Machine Learning con Scikit-Learn, Keras y TensorFlow. O'Reilly.
- Peter Bruce, Andrew Bruce y Peter Gedeck. Estadística práctica para ciencia de datos con R y Python. Marcombo.
- Materiales formativos FP Online del Ministerio de Educación y Formación Profesional. Módulo de Programación de Inteligencia Artificial.
- https://keras.io
- https://www.tensorflow.org