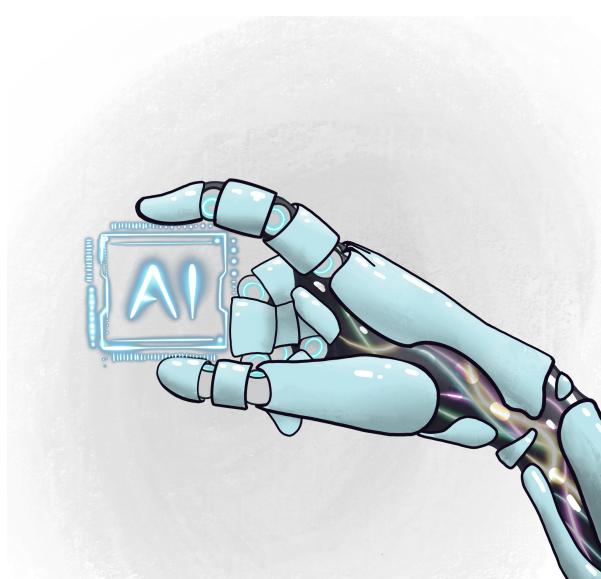


Programación de redes neuronales profundas.

Caso práctico



@casfatesvano (<https://twitter.com/casfatesvano>) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

Lorena es una MLops en prácticas en Pick&Deliver, una empresa de servicios de logística y paquetería para empresas. Le han encargado que investigue y desarrolle un modelo predictivo con Python para detectar el riesgo de que un pedido se entregue pasado el plazo máximo de entrega.

Ha probado varios modelos de aprendizaje automático, con una tasa de acierto buena pero no suficiente. Lorena se ha decidido a probar con una red neuronal profunda, y ya tiene claro que va a trabajar con la API (Interfaz de programación de aplicaciones) de Tensorflow y Keras para Python. Tras analizar muchos ejemplos y documentación, sabe que debe decidir parámetros importantes de la arquitectura del modelo. ¿Cuántas capas debe tener? ¿Qué optimizador es el más adecuado para su problema?

En esta unidad volvemos a tratar algunos conceptos vistos en el módulo 2 sobre redes neuronales profundas. Pero esta vez lo hacemos de forma más concreta y aplicada, orientada a la programación del modelo en Python, utilizando Keras.

En concreto, veremos estos tres conceptos clave:

- ✓ 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).

Cuando finalices esta unidad, deberías ser capaz de:

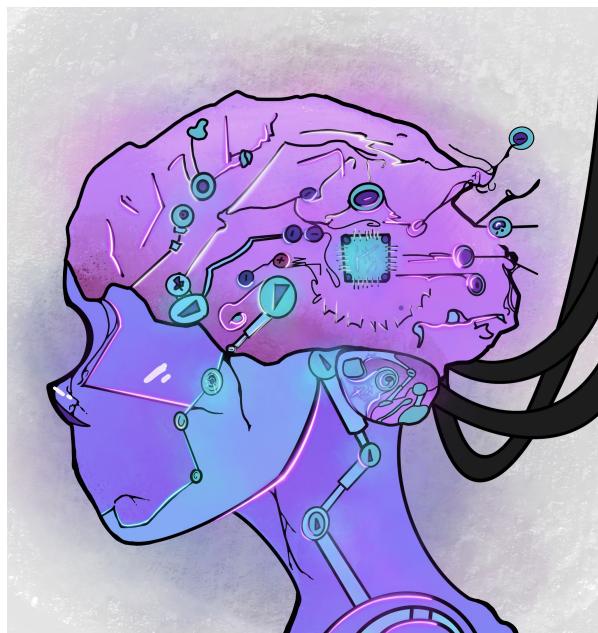
- ✓ Configurar un modelo de red neuronal profunda sencillo.
- ✓ Programar su entrenamiento, eligiendo los parámetros y funciones más adecuados para el tipo de problema.
- ✓ Entender las métricas de monitorización del entrenamiento y saber ajustar los parámetros en función de éstas.



[Ministerio de Educación y Formación Profesional](https://www.educacionyfp.gob.es/portada.html)
(<https://www.educacionyfp.gob.es/portada.html>) (Dominio público)

1.- Las capas y sus parámetros.

Caso práctico



@casfatesvano (<https://twitter.com/cASFATESVANO>) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

Lorena está en el equipo de Miguel, el Director de tecnología de Pick&Deliver. Él está haciendo de mentor en sus prácticas, así que todos los días tienen una breve reunión para revisar sus tareas y resolver dudas. Hoy, Lorena le cuenta a Miguel las decisiones que ha ido tomando sobre cómo desarrollar un modelo basado en redes neuronales profundas para predecir indicadores clave en la empresa. Pero ahora debe empezar a diseñar un modelo y no sabe bien por dónde empezar. Miguel le aconseja que revise bien la documentación, construya un primer modelo, y vaya haciendo pequeñas variaciones para comprobar qué cambios contribuyen a mejorar la precisión del modelo. Lorena ya conoce las características principales de una red neuronal profunda, y sobre todo, sabe que es el tipo de modelo que mejor comportamiento va a tener cuando se trata de un gran volumen de datos con un gran número de variables con relaciones complejas entre ellas. En el fondo, no es tan difícil entender en qué va a consistir la red neuronal profunda, pues es algo inspirado en el propio cerebro humano y sus redes de neuronas.

(La imagen que encabeza este fragmento de caso práctico ha sido realizada utilizando la herramienta de diseño basada en inteligencia artificial DALLE2, en un proceso de varias iteraciones a partir de instrucciones dadas y un tratamiento final por parte del

autor @casfatesvano de la imagen que dio la herramienta).

Partiendo del mismo dataset de la unidad anterior basado en datos de entregas de pedidos de venta de comercio electrónico, después de probar diversos modelos de aprendizaje automático, aplicamos ahora la técnica de un modelo de varias capas de redes neuronales. Aquí puedes ver un ejemplo de cómo se podría aplicar. La preparación de los datos es muy similar al ejemplo trabajado en la unidad 4. La precisión que se alcanza no es muy diferente a la de aplicar otras técnicas más clásicas de aprendizaje automático, y es que las redes neuronales profundas marcan la diferencia con grandes datasets, especialmente en el caso de datos no estructurados, como imágenes o lenguaje natural. Pero vamos a empezar por aplicar la técnica a un dataset ya conocido, para que puedas ir viendo las diferencias en su planteamiento.

La clave de esta técnica es la estructura por capas, que en el caso de utilizar la librería Keras, viene dada por la clase Sequential. El tipo principal de capa será la red neuronal, que viene dada por la clase Dense. Será necesario utilizar, para el entrenamiento, una función de coste (Loss) y un optimizador.

Echa un vistazo al código propuesto, y no te preocupes si no entiendes todo, pues en las siguientes secciones analizaremos el significado y sentido de cada parte.

Importación de librerías y carga de los datos

Trabajamos sobre el dataset [E-commerce Shipping Data
\(https://www.kaggle.com/datasets/prachi13/customer-analytics\)](https://www.kaggle.com/datasets/prachi13/customer-analytics)

In [1]:

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Creamos una carpeta en Drive llamada Datasets y subimos en ella la carpeta obtenida en Kaggle, dentro de la cual está el archivo Train.csv que es con el que vamos a trabajar. Conectamos el notebook a Drive y ponemos la ruta del archivo en la función read_csv

In [2]:

```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

In [3]:

```
data = pd.read_csv('/content/drive/MyDrive/Datasets/Shipment_arrival/Train.csv')

# Si prefieres subir el dataset directamente a La carpeta temporal del notebook
ok
# solo tienes que ejecutar esto:
#
# data = pd.read_csv('/content/Train.csv')
```

Preprocesado de los datos

Eliminamos la columna del identificador de cada registro porque no nos aporta nada

In [7]:

```
data = data.drop('ID', axis=1)
```

Empezamos a convertir variables categóricas en numéricas. Para el caso del género, solo hay dos valores posibles, así que utilizamos la función replace

In [8]:

```
data['Gender'] = data['Gender'].replace({'F': 0, 'M': 1})
```

Para los campos que contienen más de dos categorías, recurrimos a una técnica muy útil denominada one-hot encoding, que consiste en desdoblart una columna en tantas como categorías tenga, y cada instancia tendrá valor 1 para la categoría del valor original y valor 0 en las demás.

In [9]:

```
def onehot_encode(df, column):
    df = df.copy()
    dummies = pd.get_dummies(df[column], prefix=column)
    df = pd.concat([df, dummies], axis=1)
    df = df.drop(column, axis=1)
    return df

for column in ['Warehouse_block', 'Mode_of_Shipment', 'Product_importance']:
    data = onehot_encode(data, column=column)
```

Separamos el dataset en los datos de entrada 'X' y los de salida 'y'

In [11]:

```
y = data['Reached.on.Time_Y.N']
X = data.drop('Reached.on.Time_Y.N', axis=1)
```

Y también hacemos la separación entre los datos que usaremos para el entrenamiento 'train' y los que reservamos para evaluar después el modelo 'test'

In [64]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True, random_state=1)
```

Para varias de las técnicas que se van a aplicar, es necesario que los datos estén normalizados. En general, esto se hace operando con la media y la varianza, pero si no conoces la técnica matemática, no te preocunes, el módulo StandardScaler lo hace por ti.

Puedes consultar la [documentación \(https://scikit-learn.org/stable/modules/preprocessing.html\)](https://scikit-learn.org/stable/modules/preprocessing.html) para saber más.

In [65]:

```
scaler = StandardScaler()
scaler.fit(X_train)
X_train = pd.DataFrame(scaler.transform(X_train),
                       index=X_train.index,
                       columns=X_train.columns)
X_test = pd.DataFrame(scaler.transform(X_test),
                      index=X_test.index,
                      columns=X_test.columns)
```

Generación del modelo y entrenamiento

In [16]:

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

In [102]:

```
model = Sequential()
model.add(Dense(128, activation='tanh', input_shape=(18,)))
model.add(Dense(320, activation='tanh'))
model.add(Dense(512, activation='tanh'))
model.add(Dense(320, activation='tanh'))
model.add(Dense(128, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

In [103]:

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

In [104]:

```
model.fit(X_train, y_train, epochs= 10)
```

```
Epoch 1/10
241/241 [=====] - 3s 9ms/step - loss: 0.5947 - accuracy: 0.6280
Epoch 2/10
241/241 [=====] - 2s 9ms/step - loss: 0.5448 - accuracy: 0.6515
Epoch 3/10
241/241 [=====] - 2s 7ms/step - loss: 0.5318 - accuracy: 0.6509
Epoch 4/10
241/241 [=====] - 2s 8ms/step - loss: 0.5270 - accuracy: 0.6520
Epoch 5/10
241/241 [=====] - 2s 8ms/step - loss: 0.5201 - accuracy: 0.6705
Epoch 6/10
241/241 [=====] - 2s 8ms/step - loss: 0.5250 - accuracy: 0.6563
Epoch 7/10
241/241 [=====] - 2s 7ms/step - loss: 0.5256 - accuracy: 0.6563
Epoch 8/10
241/241 [=====] - 2s 8ms/step - loss: 0.5222 - accuracy: 0.6505
Epoch 9/10
241/241 [=====] - 2s 8ms/step - loss: 0.5173 - accuracy: 0.6518
Epoch 10/10
241/241 [=====] - 2s 9ms/step - loss: 0.5218 - accuracy: 0.6613
```

Out[104]:

```
<keras.callbacks.History at 0x7fdf2e243a90>
```

In [105]:

```
model.evaluate(X_test, y_test)
```

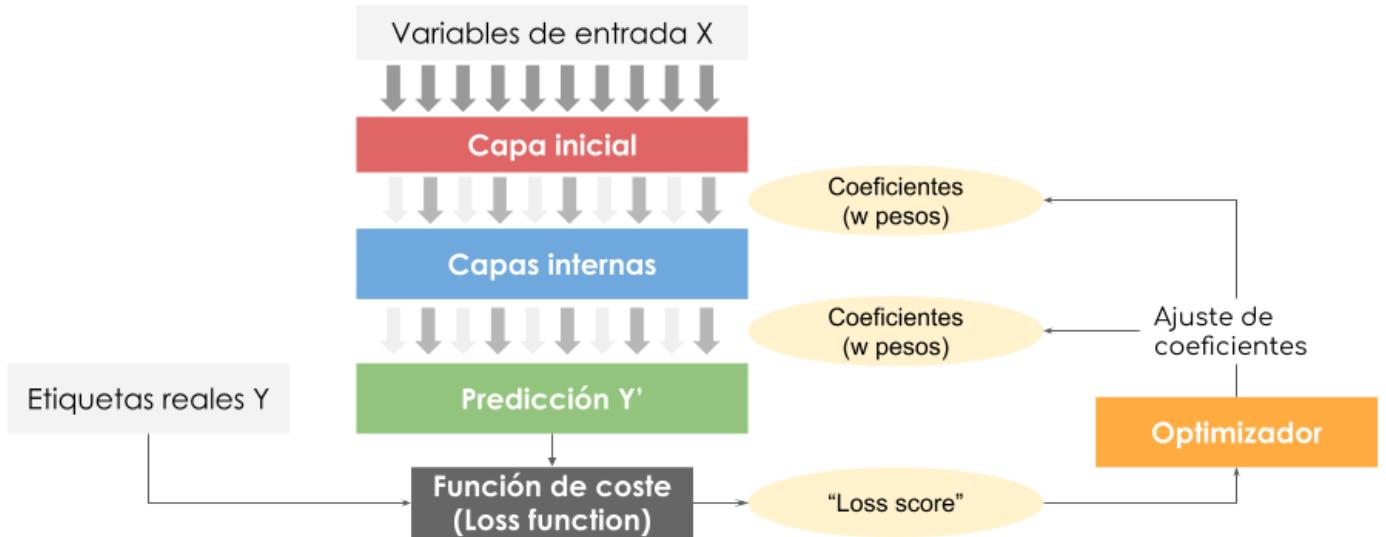
```
104/104 [=====] - 0s 3ms/step - loss: 0.5044 - accuracy: 0.6839
```

Out[105]:

```
[0.504391074180603, 0.6839393973350525]
```

Hemos logrado una precisión de algo más del 68%

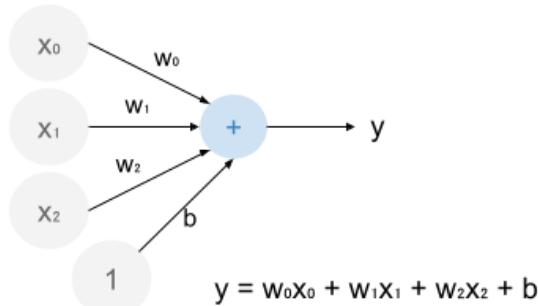
1.1.- Esquema de una DNN.



Carmen Bartolomé ([CC BY-SA \(<http://creativecommons.org/licenses/?lang=es>\)](http://creativecommons.org/licenses/?lang=es))

¿Recuerdas lo que era una red neuronal profunda? En el módulo 2 de este curso has visto una explicación teórica, y vimos un ejemplo en la unidad 4, en el apartado de la librería Tensorflow.

La unidad básica de una red neuronal es el perceptrón. La operación que realiza éste, es tan sencilla como la función lineal. ¿Te acuerdas de la ecuación de la recta? Algo así como $y = ax + b$. Pues en el caso del perceptrón, tenemos tantas variables x como datos de entrada, y en vez de usar el coeficiente "a" multiplicando a " x ", usamos un tipo de coeficientes que se llaman "pesos" o "weights". Con esa nomenclatura, tendríamos que la ecuación lineal de un perceptrón sería:



Carmen Bartolomé ([CC BY-SA \(<http://creativecommons.org/licenses/?lang=es>\)](http://creativecommons.org/licenses/?lang=es))

Y el modelo que representaría este perceptrón con 3 variables de entrada sería:

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(units=1, input_shape=[3])
])
```

De momento, este modelo solo tendría una capa, y sería una capa de tipo Dense (veremos qué características tienen estas capas en seguida). Dentro de los parámetros de la capa, vemos que "units", es decir, cuantas "neuronas" hay, es el valor 1, pues solo estamos operando con un perceptrón. Y el parámetro "input_shape", tiene el valor 3, pues solo tenemos 3 entradas más el sesgo, que es un valor independiente representado por b .

Una red neuronal profunda 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 unidad, siempre empezaremos así la construcción del modelo.

En las siguientes secciones, vamos a ir viendo qué tipo de capas tenemos y cómo fijamos sus parámetros principales.

Autoevaluación

¿A qué clase tenemos que llamar para crear un modelo con varias capas?

- Softmax
- Adam
- Sequential

Softmax es una función de activación.

Adam es un optimizador

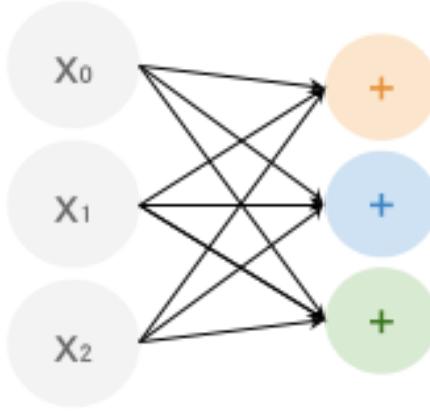
Opción correcta

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

1.2.- Dense.

Capa de red densamente conectada



Carmen Bartolomé ([CC BY-SA \(\[http://creativecommons.org/licenses/?
lang=es\]\(http://creativecommons.org/licenses/?lang=es\)\)](http://creativecommons.org/licenses/?lang=es)

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 tres neuronas o perceptrones que operan con combinaciones de las variables de entrada. A ésto, 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.

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
)
```

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

```
import keras

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.

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:

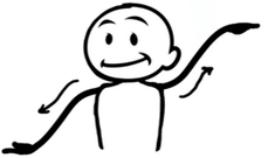
- ✓ Si es clasificación binaria, la capa de salida tendrá una única neurona
- ✓ Si es clasificación múltiple, la capa de salida debe tener tantas neuronas como clases o categorías de clasificación tenga el problema.

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. Despues, 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

Sigmoid	Tanh	Step Function	Softplus
			
$y = \frac{1}{1+e^{-x}}$	$y = \tanh(x)$	$y = \begin{cases} 0, & x < n \\ 1, & x \geq n \end{cases}$	$y = \ln(1+e^x)$
ReLU	Softsign	ELU	Log of Sigmoid
			
$y = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$	$y = \frac{x}{(1+ x)}$	$y = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$	$y = \ln\left(\frac{1}{1+e^{-x}}\right)$
Swish	Sinc	Leaky ReLU	Mish
			
$y = \frac{x}{1+e^{-x}}$	$y = \frac{\sin(x)}{x}$	$y = \max(0.1x, x)$	$y = x(\tanh(\text{softplus}(x)))$

[@sagihaider](https://twitter.com/sagihaider/status/1412705071536693254) (<https://twitter.com/sagihaider/status/1412705071536693254>) ([CC BY-SA](http://creativecommons.org/licenses/?lang=es) (<http://creativecommons.org/licenses/?lang=es>))

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

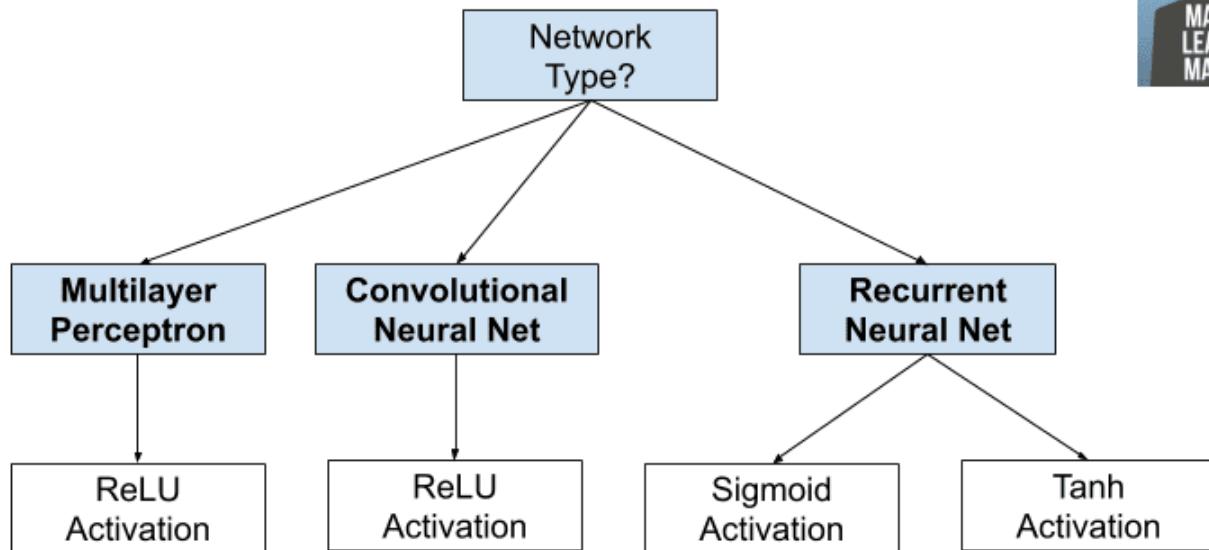
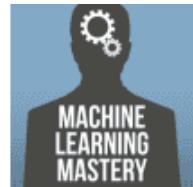
- ✓ ReLU
- ✓ Sigmoid
- ✓ Softmax
- ✓ Tanh

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
- ✓ 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 estos dos esquemas de MachineLearningMastery.com tienes también una orientación para elegir función de activación según se trate de una capa interna o de la capa de salida.

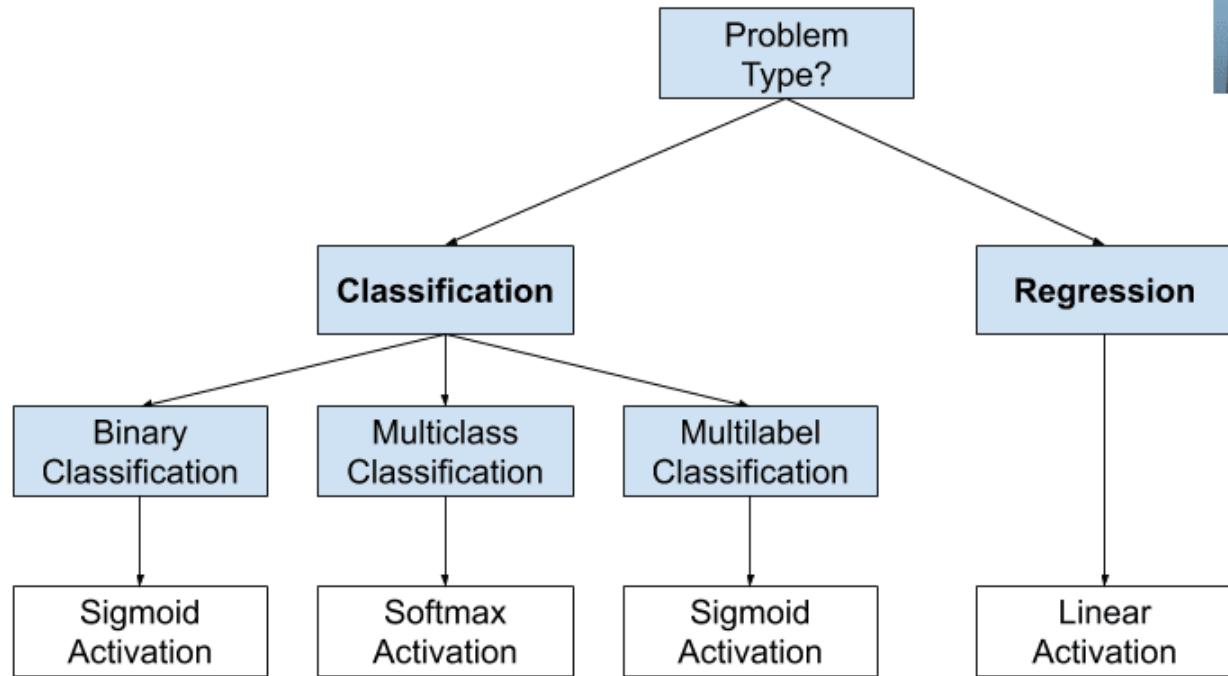
How to Choose an Hidden Layer Activation Function



MachineLearningMastery.com

[MachineLearningMastering.com](https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/) (<https://creativecommons.org/licenses/by-sa/4.0/>) (CC BY-SA (<http://creativecommons.org/licenses/by-sa/4.0/> lang=es))

How to Choose an Output Layer Activation Function



MachineLearningMastery.com

[MachineLearningMastering.com](https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/) (<https://creativecommons.org/licenses/by-sa/4.0/>) (CC BY-SA (<http://creativecommons.org/licenses/by-sa/4.0/> lang=es))

Para saber más

Sobre las funciones de activación, puedes leer el análisis que hace B. Chen en [este artículo](https://towardsdatascience.com/7-popular-activation-functions-you-should-know-in-deep-learning-and-how-to-use-them-with-keras-and-27b4d838dfe6) (<https://towardsdatascience.com/7-popular-activation-functions-you-should-know-in-deep-learning-and-how-to-use-them-with-keras-and-27b4d838dfe6>). Y sobre la función Softmax, también tienes [un artículo](https://towardsdatascience.com/softmax-activation-function-how-it-actually-works-d292d335bd78) (<https://towardsdatascience.com/softmax-activation-function-how-it-actually-works-d292d335bd78>) muy descriptivo y completo que te puede orientar mejor.

1.3.- Flatten.

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 32 x 32 pixels, 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'))
```

Autoevaluación

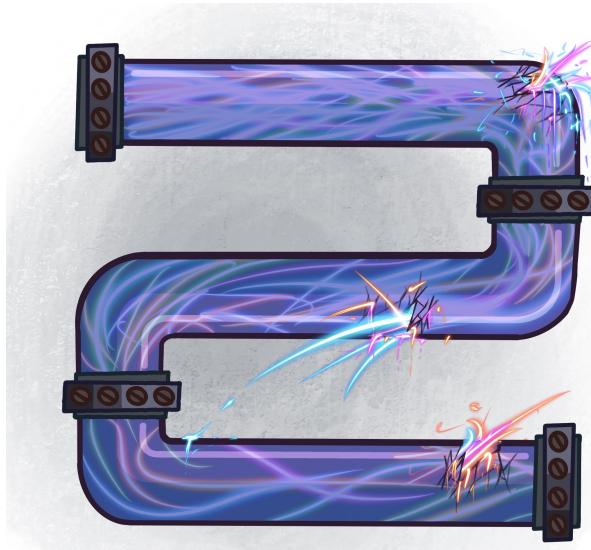
Completa los espacios en blanco para crear un modelo de red neuronal profunda con una capa de entrada en la que los datos de entrada son imágenes de 28x28 pixels y dos capas de red neuronal. El modelo servirá para clasificar imágenes de manzanas, peras, naranjas y melocotones.

```
model = keras.Sequential()
keras.layers.  (input_shape =  )
keras.layers.Dense(128, activation = 'relu')
keras.layers.Dense(, activation = ' ')
```

Vas a necesitar utilizar una capa de tipo Flatten. Las dimensiones de los datos de entrada serán 28 por 28 píxeles. Como el problema tiene 4 clases a identificar (4 tipos de fruta), la capa de salida debe tener 4 neuronas. La capa de salida es conveniente que utilice la función de activación softmax.

2.- Tipos de función de coste (Loss).

Caso práctico



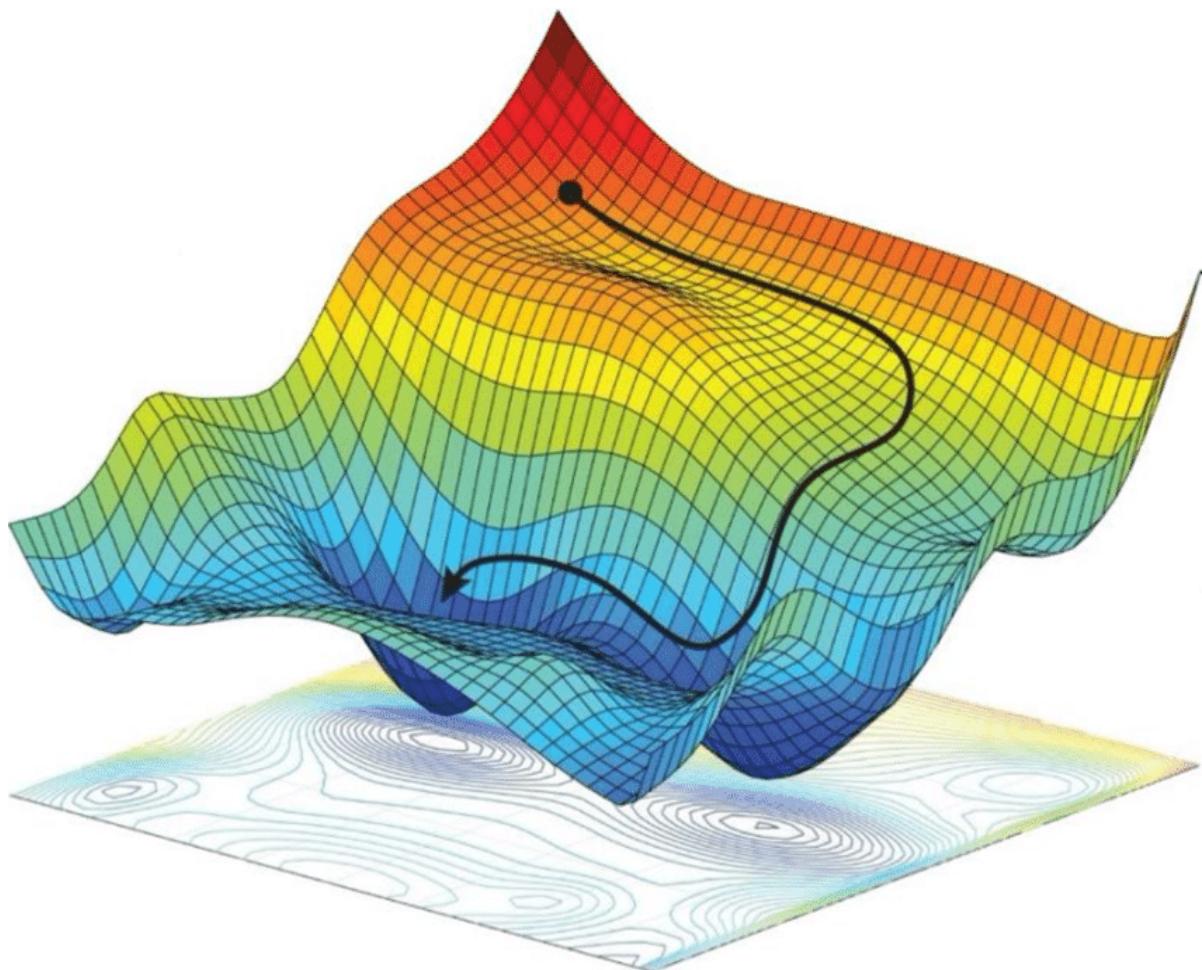
[@casfatesvano](https://twitter.com/casfatesvano) ([CC BY-SA](https://creativecommons.org/licenses/?lang=es))
(<http://creativecommons.org/licenses/?lang=es>)

Lorena y Miguel han estado pensando en varias configuraciones del modelo. Una de las decisiones a tomar, es qué función de coste, o función de medida del error, van a utilizar. Miguel le explica a Lorena que es la función que representa la pérdida de ajuste a la información subyacente que hay en los datos, y que hay diferentes funciones para cada tipo de problema, y es bueno saber cuál es la más adecuada en cada caso. "Entonces, si lo que tenemos es un problema de clasificación binaria, con dos posibles valores de salida, ¿qué función de coste debería utilizar?" pregunta Lorena.

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'])
```

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.



[Arizan & Assibi \(https://www.researchgate.net/figure/Figura-3511-Descenso-de-gradiente-estocastico-SGD-Ariza-R-y-Hassibi-B-2019_fig9_344388136\)](https://www.researchgate.net/figure/Figura-3511-Descenso-de-gradiente-estocastico-SGD-Ariza-R-y-Hassibi-B-2019_fig9_344388136) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>)))

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 en menos 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](https://es.wikipedia.org/wiki/Entrop%C3%ADa_cruzada) (https://es.wikipedia.org/wiki/Entrop%C3%ADa_cruzada)", en contraposición al error cuadrático medio o MSE que se venía utilizando en otros modelos.

Estas tres funciones a continuación, 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.

Debes conocer

En el caso de problemas de regresión, Keras proporciona las funciones de error:

- ✓ MeanSquaredError: penaliza mucho los errores grandes, y varía poco en la proximidad de los valores reales.
- ✓ MeanAbsolutePercentageError: es una métrica fácil de entender, basada en el porcentaje de error.
- ✓ MeanSquaredLogarithmicError: recomendable para ignorar las anomalías o los grandes errores.

Para saber más

Si quieras profundizar más en la justificación del uso de la entropía cruzada, te recomendamos que eches un vistazo a [este artículo sobre la entropía cruzada binaria](https://rubialesalberto.medium.com/funciones-de-error-con-entropia-cross-entropy-y-binary-cross-entropy-8df8442cdf35) (<https://rubialesalberto.medium.com/funciones-de-error-con-entropia-cross-entropy-y-binary-cross-entropy-8df8442cdf35>).

Y para saber aplicar las clases o funciones relacionadas con todo lo que hemos visto, siempre es recomendable recurrir a la [documentación de keras al respecto](https://keras.io/api/losses/) (<https://keras.io/api/losses/>).

3.- Optimizadores.

Caso práctico



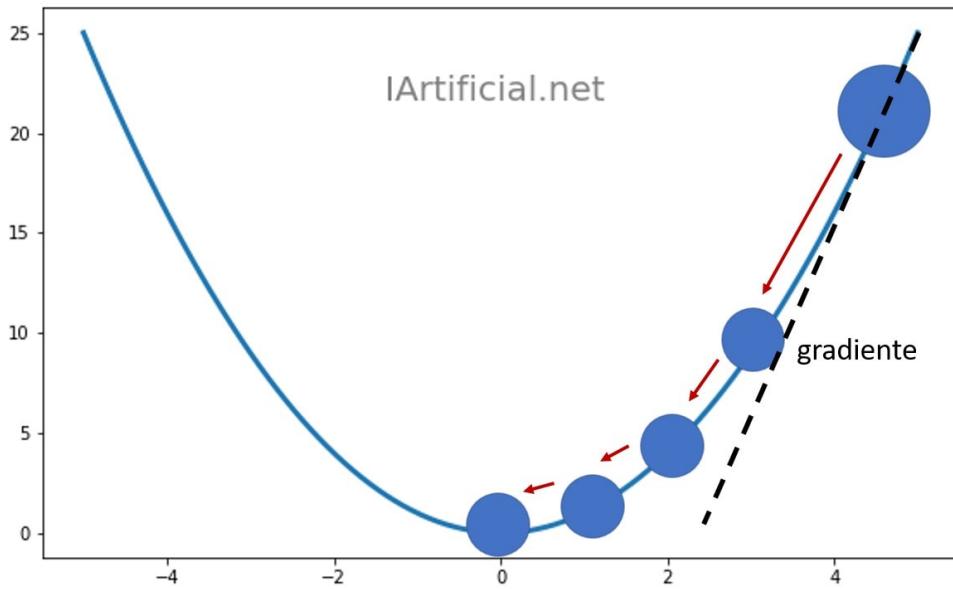
@casfatesvano (<https://twitter.com/casfatesvano>) (CC BY-SA
(<http://creativecommons.org/licenses/?lang=es>))

Lorena ya se ha hecho un esquema de situación para elegir la función de coste, que representa el error del modelo para cada configuración de los coeficientes de la red neuronal profunda, pero ahora hay que elegir un optimizador, que es el encargado de buscar la configuración del modelo con el error mínimo posible. Los distintos métodos de optimización están basados en conceptos matemáticos que a Lorena le cuesta

seguir, pero investiga un poco y analiza comparativas entre ellos. De nuevo, descubre que hay dos optimizadores que se utilizan sistemáticamente en casi todos los entrenamientos que ha revisado: RMSprop y Adam

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.

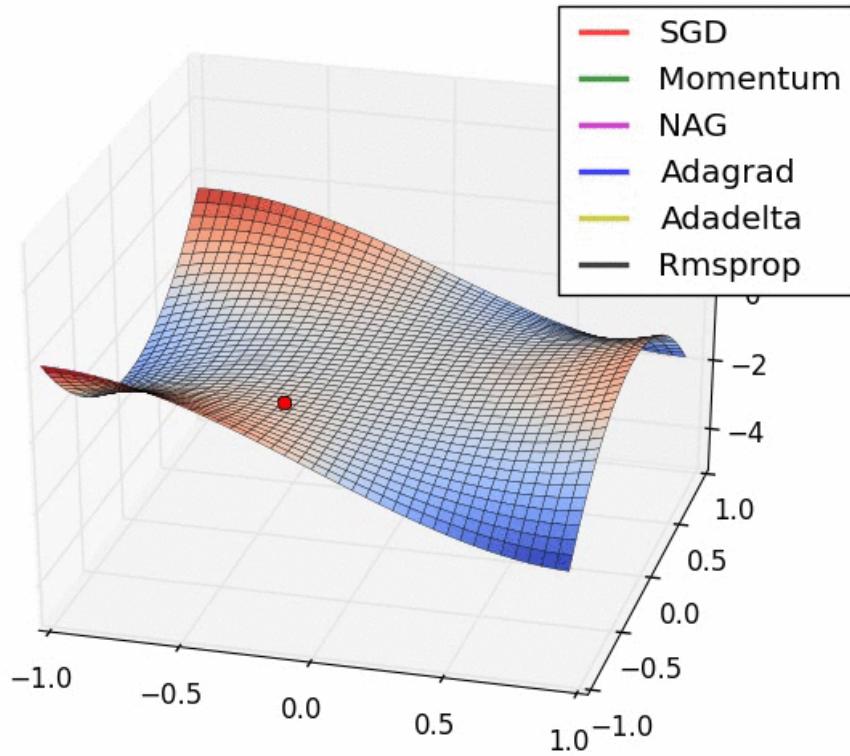
$$w_{k+1} = w_k - \eta \frac{1}{n} \sum_{i=t}^{t+n-1} \nabla f_{w_k}(x^i)$$



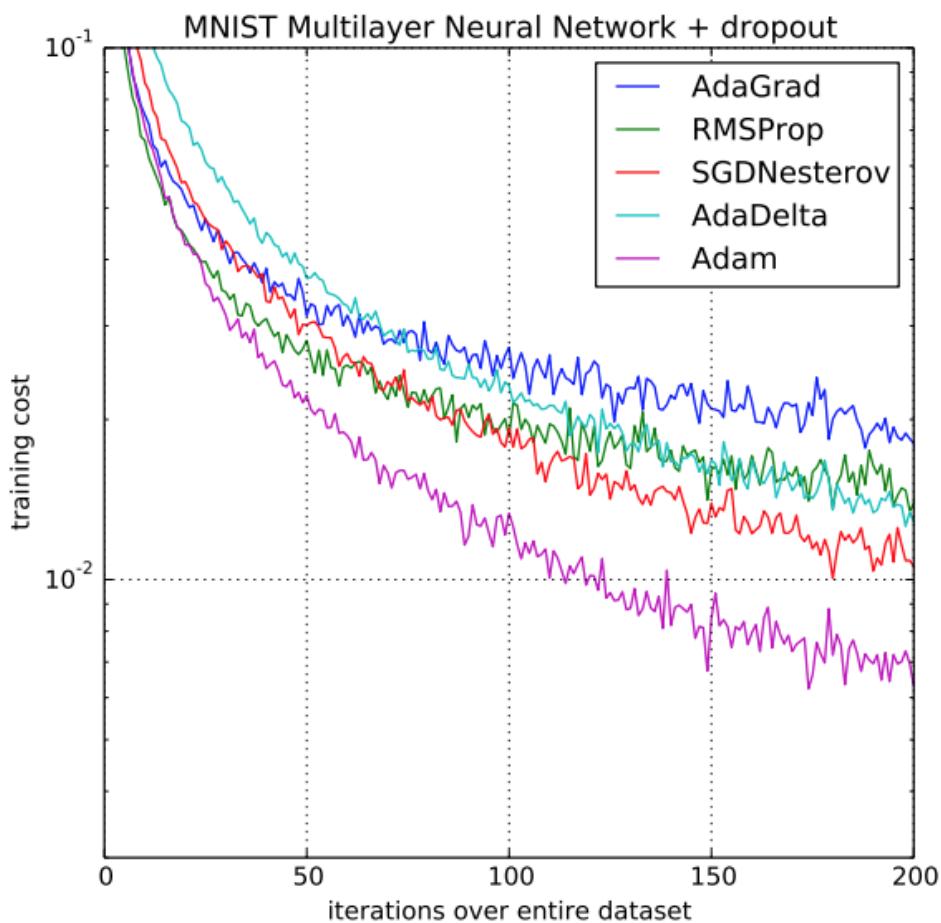
[IArtificial.net \(https://www.iartificial.net/gradiente-descendiente-para-aprendizaje-automatico/\)](https://www.iartificial.net/gradiente-descendiente-para-aprendizaje-automatico/) (CC BY-SA (<http://creativecommons.org/licenses/?lang=es>))

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.



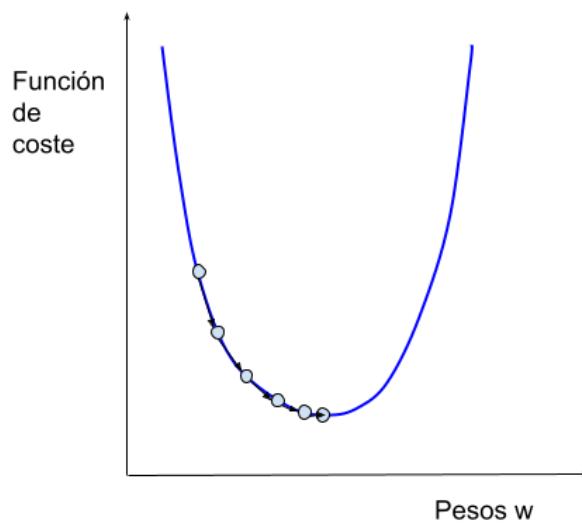
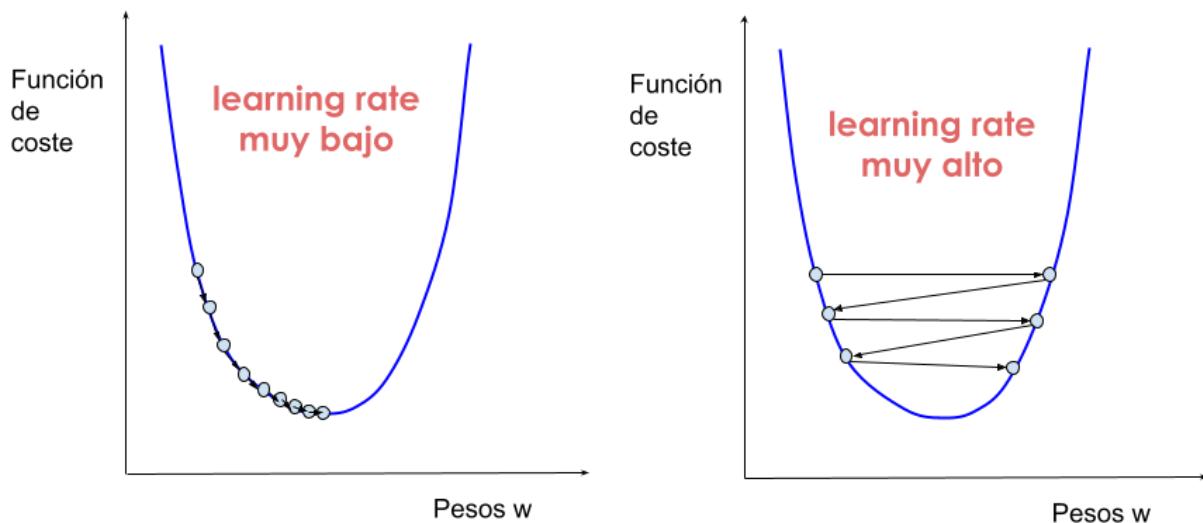
[Optimization Open Textbook \(\[https://optimization.cbe.cornell.edu/index.php?title=Main_Page\]\(https://optimization.cbe.cornell.edu/index.php?title=Main_Page\)\) \(CC0](https://optimization.cbe.cornell.edu/index.php?title=Main_Page)
[\(<http://creativecommons.org/publicdomain/zero/1.0/deed.es>\)](http://creativecommons.org/publicdomain/zero/1.0/deed.es)



[Optimization Open Textbook \(\[https://optimization.cbe.cornell.edu/index.php?title=Main_Page\]\(https://optimization.cbe.cornell.edu/index.php?title=Main_Page\)\) \(CC0](https://optimization.cbe.cornell.edu/index.php?title=Main_Page)
[\(<http://creativecommons.org/publicdomain/zero/1.0/deed.es>\)](http://creativecommons.org/publicdomain/zero/1.0/deed.es)

Ratio de aprendizaje o "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 learning rate 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:

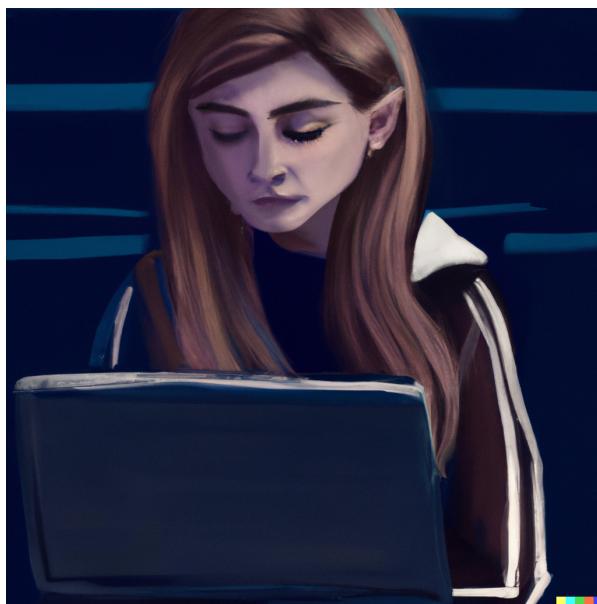
```
opt = keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss='categorical_crossentropy', optimizer=opt)
```

Para saber más

Para conocer mejor el uso de los optimizadores con keras, puedes consultar la [documentación](https://keras.io/api/optimizers/) (<https://keras.io/api/optimizers/>) que hay en su API (Application Programming Interfaces) reference. También puedes leer [este artículo](https://velascoluis.medium.com/optimizadores-en-redes-neuronales-profundas-un-efoque-pr%C3%A1ctico-819b39a3eb5) (<https://velascoluis.medium.com/optimizadores-en-redes-neuronales-profundas-un-efoque-pr%C3%A1ctico-819b39a3eb5>) en el que el autor hace un experimento probando diferentes optimizadores para varios casos.

4.- Entrenamiento de la red profunda.

Caso práctico



DALLE2 (<https://openai.com/dall-e-2/>) (CC0
(<http://creativecommons.org/publicdomain/zero/1.0/deed.es>))

Lorena ya tiene listo su modelo de red neuronal profunda y ha elegido casi todos los parámetros necesarios para su entrenamiento. Ahora llega la hora de la verdad. Debe llamar al método fit() de la librería Keras sobre el modelo y pasarle los parámetros que

requiere dicha función. Está claro que una parte fundamental son los datos, que ya ha separado entre datos de entrada y etiquetas o datos de salida. Pero hay un parámetro especial: "epochs".

Ella sabe que el parámetro "epochs" se refiere al número de iteraciones que se van a dar durante el proceso del entrenamiento del modelo, pero se para a pensar por qué es necesario y qué implica poner un valor u otro.

La imagen de la cabecera está generada con el modelo de inteligencia artificial DALLE2 de openAI, a partir de la instrucción "dibujo de chica joven rubia delante de la pantalla del ordenador" y seleccionada entre varias opciones dadas por la herramienta (se accede a DALLE2 por invitación tras apuntarse a la lista de espera).

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, más, 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 continuación, tienes un ejemplo de un modelo para clasificación de imágenes, en el que puedes ver todos los elementos e hiperparámetros que hemos visto en esta unidad. Fíjate bien: ¿qué función de coste se ha utilizado? ¿Qué optimizador?. Te recomendamos que programes tu propio modelo, creando una copia en Drive del cuaderno original (<https://colab.research.google.com/drive/13uZX8r1Ho-ZfE8a49vbrVPPIM8l8ZQjs?usp=sharing>), probando diferentes configuraciones de capas y neuronas. ¿Consigues mejorar la precisión?

Ejemplo básico de una red neuronal para clasificación de imágenes

Importación de las librerías

In []:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
```

Carga de los datos

Vamos a utilizar el propio dataset que viene precargado en Keras:

https://keras.io/api/datasets/fashion_mnist/ (https://keras.io/api/datasets/fashion_mnist/).

In []:

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
32768/29515 [=====] - 0s 0us/step
40960/29515 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [=====] - 0s 0us/step
26435584/26421880 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
16384/5148 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [=====] - 0s 0us/step
4431872/4422102 [=====] - 0s 0us/step
```

Exploración y visualización de los datos

Utilizamos len, shape, e imshow para explorar y ver con qué estamos trabajando Pistas: adimensionalizar las imágenes (intensidad de tono de pixel) y crear una lista con los nombres de las clases.

In []:

```
train_images.shape
```

Out[]:

```
(60000, 28, 28)
```

In []:

```
len(train_labels)
```

Out[]:

```
60000
```

In []:

```
train_labels
```

Out[]:

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

In []:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

In []:

```
plt.figure()
plt.imshow(train_images[10])
plt.colorbar()
```

Out[]:

```
<matplotlib.colorbar.Colorbar at 0x7f564b93ea90>
```



In []:

```
train_images = train_images/255.0
test_images = test_images/255.0
```

In []:

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(train_images[i], cmap = plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])

# El método show no es necesario en colab
plt.show
```

Out[]:

<function matplotlib.pyplot.show>



Construcción del modelo

Llamamos a la clase Sequential y usamos layers para crear las capas. Con Dense, se crean las capas de red neuronal. Utilizaremos las funciones de activación ReLu en las capas internas y Softmax en la de salida.

Pista: Flatten crea una capa que reduce a 1 la dimensión del array de las imágenes

In []:

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape = (28,28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10,activation='softmax')
])
```

Parámetros del entrenamiento

El método compile nos permite configurar el optimizador, función de coste,etc. En este caso, vamos a recurrir al optimizador Adam, y la función de coste sparse_categorical_crossentropy.

In []:

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

Entrenamiento del modelo

Utilizamos la función fit para el entrenamiento de la DNN. Tenemos que pasar como argumentos los valores de entrada, los de salida (las etiquetas) y el número de iteraciones o epochs.

In []:

```
model.fit(train_images,train_labels,epochs=5)
```

```
Epoch 1/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.5016 - accuracy: 0.8219
Epoch 2/5
1875/1875 [=====] - 5s 2ms/step - loss: 0.3733 - accuracy: 0.8645
Epoch 3/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.3396 - accuracy: 0.8765
Epoch 4/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.3147 - accuracy: 0.8849
Epoch 5/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.2956 - accuracy: 0.8906
```

Out[]:

```
<keras.callbacks.History at 0x7f0956d78a50>
```

Evaluación del modelo

Para evaluar el modelo, recurrimos a la función evaluate y guardamos los valores que nos devuelve, el error o loss, y la precisión o accuracy.

In []:

```
test_loss,test_acc = model.evaluate(test_images,test_labels)
```

```
print('Test accuracy: ', test_acc)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.3582 - acc  
uracy: 0.8724  
Test accuracy: 0.8723999857902527
```

Generación del modelo predictivo y pruebas

Con la función predict, generamos la colección de resultados del modelo para nuevos datos de entrada, y podemos analizar su comportamiento.

In []:

```
predictions = model.predict(test_images)
```

In []:

```
predictions[5]
```

Out[]:

```
array([9.5059222e-06, 9.9998963e-01, 5.8573161e-08, 1.7806494e-07,  
2.5649666e-07, 3.7701500e-12, 3.9674049e-07, 1.9414056e-13,  
1.2030660e-10, 2.8490481e-11], dtype=float32)
```

In []:

```
np.argmax(predictions[5])
```

Out[]:

```
1
```

In []:

```
class_names[1]
```

Out[]:

```
'Trouser'
```

In []:

```
class_names[test_labels[5]]
```

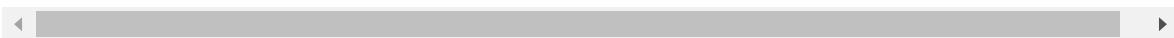
Out[]:

```
'Trouser'
```

In []:

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(test_images[i], cmap=plt.cm.binary)
    predicted_label = np.argmax(predictions[i])
    true_label = test_labels[i]
    if predicted_label == true_label:
        color = 'green'
    else:
        color = 'red'

    plt.xlabel("{} ({})".format(class_names[predicted_label],
                                class_names[true_label]),
                color=color)
```



In []:

```
# MIT License
#
# Copyright (c) 2017 François Chollet
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

Para saber más

Existen muchos más parámetros disponibles para ajustar el entrenamiento del modelo basado en redes neuronales profundas. Aunque algunos los veremos en siguientes unidades, puedes consultar la [sección de la documentación de Keras dedicada al entrenamiento](https://keras.io/api/models/model_training_apis/#fit-method) (https://keras.io/api/models/model_training_apis/#fit-method) para ver todas las opciones disponibles.

Autoevaluación

¿Qué parámetro del entrenamiento fija el número de iteraciones del optimizador?

- Loss
- Epochs
- Iterations

El parámetro Loss selecciona la función de coste

Opción correcta

Este parámetro no interviene en el método fit ni compile

Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto