Curso Udemy: Artificial Neural Networks (ANN) with keras in Python and R

By Daniel Lázaro Lázaro

Artificial Neural Networks (ANN)

La habilidad de las NN para crear sus propias reglas dado un conjunto de datos, y una vez con esas reglas poder hacer distinciones y separar en diferentes clases, esa habilidad de generar reglas, las hace muy versátiles y por ello muy útiles.

Vamos a ver en este curso acerca de las diferentes escalas de las NN como bien son el perceptrón o célula única (single-cell), del multi-nivel perceptrón (multiple cells), de backward-propagation y forward-propagation, del gradiente estocástico descendiente y de su implementación en Python.

Perceptrón: Vamos a ver su analogía con la biología y la neurona como unidad del sistema nervioso, un perceptrón va a ser una neurona artificial (AN) de tal forma que vamos a ver va a actuar como una neurona sigmoide, en una representación vectorial sería una función que recibiría un número "m" dado de inputs y produce un único output, todos ellos, los inputs y el output, son binarios. Hay varias formas en las que el conjunto de inputs nos pueden dar el output deseado, como que su suma ponderada supere un un dado threshold value o valor de umbral, y en caso de superarlo que sea '1' o no superarlo '0' para el output.

Hay que ver esa suma está ponderada tal que si los inputs son {x1, x2, ..., xm} y tienen pesos (weights) asociados {w1, w2, ..., wm} y entonces la magnitud podría ser

S = Sum_i=1 to m (x_i*w_i)

if S > threshold => output = 1

else output = 0

Así podríamos recordar que esos pesos {w1, w2, ..., wm} van a ser muy significantes en el problema ya que son la importancia de cada input, ya que estos inputs {x1, x2, ..., xm} solo son magnitudes binarias acerca de si el input en cuestión cumple o no una característica, es decir, será tal que x_i = {0, 1}

Así, veremos cada objeto o dato tendrá una serie de posibles m inputs o características, y una vez evaluadas, ese objeto individual tendrá una magnitud S asociada, por lo cual podremos ver que será importante también cuál es el valor del umbral.

Ahora se puede extender el concepto de perceptrón de binario, que suena muy limitado, y realmente veremos se puede trabajar con valores continuos, tanto para el input como el output, y no tendremos porqué trabajar con esos valores binarios que nos limitan tanto. Así, podremos trabajar con valores reales y el resto de valores funcionaran de la misma forma (los pesos y el umbral).

Otra forma de ver estas operaciones anteriores como despejar el umbral de tal forma que

S = Sum_i=1 to m (x_i*w_i) - umbral

output = 1 if S > threshold ; Sum_i=1 to m (x_i*w_i)* > *umbral ; Sum_i=1 to m (x_i*w_i) - umbral > 0 ;

Y podemos ver un cambio en la notación: "-umbral = b", donde la letra "b" que es una constante se denota así por el "bias" o sesgo

output = 1 if Sum_i=1 to m (x_i*w_i) + b > 0

Y análogamente

output = 0 if Sum_i=1 to m (x_i*w_i) + b < 0

La representación de la función Output va a ser una función escalón, es decir, es 0 hasta que S supera un determinado valor en cuyo caso pasa a valer 1, es decir, esta es la función de activación.

Se va a llamar función de activación porque va a ser la que nos va a determinar cómo es el output, es una función en la que se va a satisfacer una condición o no, si supera un umbral/threshold.

Hay muchos otros tipos de funciones de activación, uno de los más populares es la "sigmoid function" que es una versión amortiguada de la función escalón. Es una especie de curva que "converge" en el 0 y 1 pero en vez de una transición en un solo punto, presenta ese cambio de una forma más suave. Funcionan también en la regresión logística.

La función sigmoide funciona mejor que la función escalón porque es menos sensible a la variación individual de una única observación. Al estar clasificando erróneamente algo, para arreglar esto un modelo lo que va a necesitar es nuevos valores de el bias y de los weights, y aquí es donde entra el problema, ya que cambios pequeños en los weights o en el bias van a acarrear cambios muy bruscos en esa función escalón, mientras que en la función sigmoide estos cambios son más graduales, por lo que esos cambios van a ser más fáciles de controlar.

Una AN individual con una función de activación sigmoide va a ser llamada "sigmoid neuron" o "logistic neuron".

La función sigmoide es

f(z) = 1/(1 + exp(-z))

Y el output va a ser

output = 1/(1 + exp(- Sum_i=1 to m (x_i*w_i) - b))

Como bien sabemos es una la inversa de una función exponencial, por lo que tomará valores de

output = [0, 1]

y de la forma mencionada previamente. de tal forma que podemos ver el output será un valor en el continuo, por lo cual hemos roto esa barrera binaria.

In [2]:

```
##### Single-Perceptron Model

import numpy as np
import pandas as pd
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
# Un dataset con 3 tipos diferentes de flores (setosa, versivolour y virginica), longit
udes del sépalo y del pétalo, en un
# array de 150x4 tipo numpy array

iris = load_iris()

iris # Nos da la información de dichas características, las 4 columnas son:
# 'Sepal Length', 'Sepal Width', 'Petal Length' y 'Petal Width'

iris.target # Nos da el resultado, el tipo de flores clasificado como '0', '1' ó '2 (se
tosa, versivolour ó virginica)

# Vamos a querer ver el problema como binario, es decir, no 3 características sino solo
2, y vamos a ver si es setosa o no
# (es decir, si no es setosa el default es que será versivolour ó virginica)

y = (iris.target == 0)
# Con esto vamos a modelar los 0's como True (es decir setosa como True), y los 1's y
 2's como False (versivolour ó virginica
# como false)

y = (iris.target == 0).astype(np.int) # Para modelar los True como '1's y los False com
o '0's, y así no variables categóricas

X = iris.data[:,(2,3)] # En caso de no querer usar las 4 características sino solo 'Pet
al Length' y 'Petal Width'

perceptron = Perceptron(random_state = 42)
# Para recibir siempre el mismo resultado

# Perceptron() Es una función que simula un perceptrón y va a tener diferentes posibles
parámetros

# 'penalty': constanque que multiplica al término de regularización
# 'alpha': término de regularización y tiene como default 0.0001
# Y el resto de hiperparámetros que dejaremos como sus respectivos default

# Implementamos el modelo
perceptron.fit(X, y)

y_predicted = perceptron.predict(X)

y_predicted

accuracy_score(y, y_predicted)
print('Accuracy of the model:', accuracy_score(y, y_predicted))
# Y vemos es 1.0 por lo cual tenemos una accuracy del 100% y que hemos encontrado todos
los casos perfectamente

# perceptron.coef_ # Como se ha realizado un ajuste de regresión lineal lo que podemos
 ver aquí son los parámetros de esa recta
```

```
# perceptron.intercept_ # El corte con el origen
```

```
C:\Users\PORTATIL\anaconda3\lib\site-packages\numpy\_distributor_init.py:3
0: UserWarning: loaded more than 1 DLL from .libs:
C:\Users\PORTATIL\anaconda3\lib\site-packages\numpy\.libs\libopenblas.NOIJ
JG62EMASZI6NYURL6JBKM4EVBGM7.gfortran-win_amd64.dll
C:\Users\PORTATIL\anaconda3\lib\site-packages\numpy\.libs\libopenblas.PYQH
XLVVQ7VESDPUVUADXEVJOBGHJPAY.gfortran-win_amd64.dll
  warnings.warn("loaded more than 1 DLL from .libs:\n%s" %
```

```
Accuracy of the model: 1.0
```

Cómo funcionan las ANN. Gradient Descent Y CrossEntropy Error Function

Ahora vamos a "apilar" estas neuronas o células individuales de tal forma que vamos a formar los Networks, AN + N = ANN y podremos tener finalmente nuestros Artificial Neural Networks. Hay dos formas de apilarlas, en paralelo o en secuencial.

El "parallel stacking" consiste en un mismo conjunto de inputs que vamos a pasar a diferentes neuronas y cada una nos va a dar un múltiples outputs diferentes dado un único conjunto de inputs.

El "sequential stacking" consiste en añadir una siguiente capa de neuronas, es decir, tendremos un conjunto de inputs y lo enviaremos a M diferentes neuronas, obteniendo así un conjunto de M outputs, hasta aquí todo esto es solo "parallel stacking", el "sequential stacking" aparece al colocar una segunda tanda M' de neuronas, cuyos inputs van a ser los outputs de las anteriores M neuronas, obteniendo así M' outputs nuevos. Una vez iterado el proceso podremos continuar añadiendo capas de neuronas hasta llegar a un output final.

Una pregunta que cabe hacerse ahora mismo es ¿por qué no podemos coger ese único conjunto de inputs y hacerlo pasar por una única neurona, perceptrón simple, y obtener directamente el único output? Es decir, sin necesidad de otras neuronas o de capas de neuronas. Y la respuesta es que, por ejemplo, en un problema de separación lineal se podrá encargar un único y perceptrón simple, pero cuando la situación es más complicada, como es casi siempre en la vida real, y necesitamos clasificar de forma no lineal, o Clústers..., necesitaremos más de un separador lineal, es decir, acabaremos teniendo una combinación de separadores lineales.

Como notación, podremos ver que este conjunto de inputs es la "input layer", el output es la "output layer" y las capas intermedias son las "hidden layers". El output layer será el último perceptrón que nos devolverá ese único output final.

Además, en el caso descrito podemos ver todos los outputs pasan a ser inputs de la siguiente layer, de tal forma que avanza en una única dirección, y diremos las ANN de este tipo serán "Feed Forward Network" mientras que si además, el output de una neurona de una capa pasa como input a todas las neuronas de la siguiente capa, se dirá la ANN será "Fully Connected Network". Si además un output volviese como input de esa misma neurona sería un "Cyclic Network".

Deep Learning es las Hidden Layers ya que es avanzar en secuencial lo que se puede ver como avanzar en profundidad. A mayor sea el número de capas, un network más profundo, y relaciones más complejas entre input y output.

¿Cómo funcionan exactamente las Neural Networks? Un poco de recapitulación, es que usaremos las neuronas individuales y estas van a ser neuronas sigmoides ya que van a actuar de forma más suave y por ello más manejable. Recordemos la fórmula era

output = 1/(1 + exp(- Sum_i=1 to m (x_i*w_i) - b))

Donde x_i es el conjunto de inputs, w_i los pesos asociados y b el bias. El problema del funcionamiento reside en encontrar los valores de los pesos y el bias para que el output calculado se parezca lo máximo posible al output predicho.

Para un caso sencillo de 2 neuronas en la para el input de 2 variables, y la neurona final, tendremos 2 pesos y 1 bias para cada una de las neuronas de entrada del input, y para la neurona final tendremos esos dos outputs por lo cual necesitaremos 2 pesos y 1 bias. Entonces vemos en total tenemos 6 pesos y 3 bias, es decir, 9 variables a determinar. ¿Cómo determinamos estos valores? Con Gradient Descent, GD, que es una

función de optimización que opera para buscar el mínimo de una función, de esta forma sabemos que hay otras funciones de optimización pero esta es mejor computacionalmente hablando que cualquier otras técnicas, ya que entrenará el modelo hasta una convergencia más rápida.

La forma en la que GD funciona es asignando un valor de w_i y b_i aleatorios inicialmente, calcular el output final asociado a este conjunto de parámetros, y finalmente vemos la función de error l asociado a esta predicción, (todo esto esto lo vamos a ver es hacia adelante, en secuencial y única dirección, Forward Propagation) ahora, cambiaremos los valores de w_i y b_i con intenciones de encontrar otros que minimicen esa función de error, es decir, volvemos atrás, realizamos un cambio en los parámetros y recalculamos la función output y luego la función de error nueva para estos nuevos parámetros (ahora hemos vuelto atrás, por lo que esto es Backward Propagation). Estamos buscando así cuáles de estos w_i y/o b_i son los que tienen mayor impacto y vamos a buscar variarlos para minimizar la función de error.

Con GD estamos buscando ese mínimo de esa función de tal forma que finalmente podemos avanzar hacia él, no buscando conocer la función del problema, lo cual tomaría demasiado tiempo, sino intentaríamos buscar una región pequeña en la que estemos y centrarnos en ella y buscar la dirección en la que avanzamos hacia abajo, es decir, nos vamos a estar basando en la pendiente de la función, que no es otra cosa que el gradiente y en dirección descendiente.

Ahora veremos qué es esta función de error l de la que estamos hablando de minimizar, una función que podemos utilizar es CrossEntropy Error Function:

l(y,y') = -y*log(y') - (1 - y)*log(1 - y')

y es el valor real del output e y' es el valor del output predicho, el por qué utilizamos esta función es porque se trata de una función sin mínimos locales, lo cual es imprescindible en un problema de mínimos, que es encontrar el mínimo global y no un mínimo relativo ya que no será la solución correcta. Como bien sabemos output varía entre 0 y 1 por ser una sigmoide, y queremos encontrar la función que nos permita ver

y' = output -> 1

Es decir, que tienda a uno, maximizar la solución. Así veremos variaremos los pesos y bias ligeramente (D*w_i y D*b_i ; denotando los diferenciales de peso y bias respectivamente) hasta minimizar la función de error

w_i' = w_i - alpha *D*w_i

b_i' = b_i - alpha *D*b_i

Donde alpha será el ratio de aprendizaje, y va a determinar el número de pasos que debemos de realizar hasta el mínimo en esa dirección hacia el lowest point. Si alpha es grande tenemos que tomar muchos pasos en dirección del fondo, la ventaja es que podemos avanzar más rápido, pero el problema es que nos podemos pasar del mínimo. En caso de estar demasiado cerca y tomar demasiados pasos, nos va a alejar del fondo. Es decir, alpha grande ayuda a converger rápido pero tiene problemas a la hora de converger.

Así pues, encontrar D*w_i y D*b_i lo vamos a conseguir con Backward Propagation

Sabemos en un caso sencillo de un perceptrón como neurona sigmoide y un input de dos características (x_1, x_2):

y' output = 1/(1 + exp(- Sum_i=1 to m (x_i*w_i) - b)) = *1/(1 + exp(- x_1*w_1 - x_2*w_2 - b))*

l(y,y') = -y*log(y') - (1 - y)*log(1 - y')

Y sabemos

y = 1

Que a su vez implica

I(y = 1,y') = -1*log(y') - (1 - 1)*log(1 - y') = log(y')

Por lo cual procederíamos a calcular I(y,y'), y aquí es cuando entra en juega el Backpropagation, lo que queremos es minimizar la función de error, lo cual significa, derivada para ver este problema de mínimos

dI(y,y')/d(y') = d(log(y'))/d(y') = -1/y' = -1/output = -(1 + exp(- x_1*w_1* - *x_2*w_2 - b))

Y si queremos hablar de la pendiente de activación y de su error va a ser recordar la función sigmoide como

y' = output = 1/(1 + exp(-z))

y ver entonces

d(y')/dz = exp(-z)/(1 + exp(-z))^2

Donde sabemos que z es tal que

z = x_1*w_1* + *x_2*w_2 + b

Y finalmente podremos ver las derivadas respecto de w_1, w_2 y b

dz/d(w_1) = x_1

dz/d(w_2) = x_2

dz/db = 1

Y con todo esto podemos construir el impacto de cada una de estas variables en la función de error I por medio de la regla de la cadena

dI/d(w_1) = dI/dy' *dy'/dz* dz/d(w_1) = -(1 + exp(- x_1*w_1* - *x_2*w_2 - b)) *exp(-z)/(1 + exp(-z))^2* x_1

dI/d(w_2) = dI/dy' *dy'/dz* dz/d(w_2) = -(1 + exp(- x_1*w_1* - *x_2*w_2 - b)) *exp(-z)/(1 + exp(-z))^2* x_2

dI/db = dI/dy' *dy'/dz* dz/db = -(1 + exp(- x_1*w_1* - *x_2*w_2 - b)) *exp(-z)/(1 + exp(-z))^2* 1

Y así ver la variación de I en función de los pesos y el bias

Finalmente podremos ver

D*w_1 = dI/d(w_2)

D*w_2 = dI/d(w_2)

D*b = dI/db

Y podremos sustituir esto en las expresiones anteriores

w_1' = w_1 - alpha *D*w_1 = w_1 - alpha *(-(1 + exp(- x_1*w_1 - x_2*w_2 - b)))* exp(-z)/(1 + exp(-z))^2 * x_1

w_2' = w_2 - alpha *D*w_2 = w_2 - alpha *(-(1 + exp(- x_1*w_1 - x_2*w_2 - b)))* exp(-z)/(1 + exp(-z))^2 * x_2

b' = b - alpha *D*b = b - alptha *(-(1 + exp(- x_1*w_1 - x_2*w_2 - b)))* exp(-z)/(1 + exp(-z))^2 * 1

Y de esta forma vamos a actualizar esoss valores de los pesos y del bias, donde hemos usado la función sigmoide y la función CrossEntropy Error Function, de tal forma que estamos haciendo uso del GD para minimizar así esa función de error y poder actualizar esos valores de los pesos y bias de forma correcta.

Posibles preguntas sobre Activation Functions

Otra pregunta que podríamos hacernos es, ¿Por qué utilizamos funciones de activación? Es el que nos da el Output de una neurona, es decir, sin una función de activación lo único que tendríamos sería una suma ponderada de los Input pesados con los Pesos, tal que

z = Sum_i=1 to m (x_i*w_i)

Donde tendríamos m Inputs x_i con sus respectivos pesos w_i pero no tendríamos una función donde evaluar z (recordemos que en el caso de la función activación sigmoide, z actúa como la magnitud a sustituir y por ello acabaríamos viendo Output(z) es decir, como función de esta magnitud z. Para un problema de regresión, solo usar z podría ser aceptable, pero para un problema de clasificación no es suficiente, ya que queremos un Output = [0,1] o de tipo binario. Además, si solo trabajásemos con z en lugar del Output veríamos que solo podríamos establecer relaciones lineales con las neuronas, de tal forma que el resultado final del Input y el Output final, por más hidden layers que hubiese o neuronas en cada una de esas hidden layers, sería una relación lineal entre el Input y el Output final del ANN.

Por estos dos motivos es por los que usamos las funciones de activación, por un lado porque así podemos imponer condiciones de frontera en estos problemas de clasificación al Output, tanto en clasificación como en regresión. El segundo motivo es que podemos así añadir no linealidad, y poder ver patrones no lineales más complejos.

También otra pregunta que podríamos hacernos sería por qué hay varios tipos de funciones de activación, antes hemos mencionado la función escalón, y la función sigmoide. Otra función de activación muy similar sería la tangente hiperbólica tanh(x) ó tgh(x) de tal forma que tiene diferentes límites de convergencia, las funciones escalón y sigmoide iban de 0 a 1, mientras que la tangente hiperbólica va de -1 a 1 siendo simétrica y pasando por el origen. Por tener esta centralidad en el origen es que vemos va a tener una mayor eficiencia en su convergencia que la sigmoide.

Otra posible función de activación es ReLU que viene de Rectified Linear Unit, y consiste en valor 0 constante como la función escalón, pero en el eje positivo de x tenemos una línea recta de una dada pendiente, es muy utilizada en las Hidden layers o layers intermedias de las NN de regresión. ReLU tiene el aspecto

if x <= 0 ; => y = 0

if x < 0 ; => y = x

La diferencia con las anteriores es que el límite inferior es cero, como en las anteriores salvo la tangente hiperbólica que tenía -1, pero, en el caso de ReLU, no tiene límite superior.

Es una función que se ejecuta bien computacionalmente, sin mucho coste y se utiliza además en las hidden layers porque introduce no linealidad, sin embargo, en la Output layer no se suele utilizar ya que para problemas de clasificación no está acotada superiormente. Ocurre parecido a como ocurría en problemas de clasificación con la función escalón, que no servía para pequeños cambios por ese cambio tan brusco y por ser entonces una función tan difícil de controlar.

Finalmente podemos ver dónde usar cada tipo de función de activación, cualquiera de ellas nos sirve para clasificación, pero para regresión deberemos usar ReLU, y acerca de en qué neurona colocarlas, Step/Escalón por lo general como Output, Sigmoid/TanH generalmente como Hidden/Output layer y ReLu para las Hidden layers.

Otra pregunta a hacernos es acerca de si las Hidden layers y la Output layer pueden tener diferentes funciones de activación, la respuesta es que sí, ya que por lo general usaremos ReLU en las Hidden layers y Sigmoid en la Output Layer.

La siguiente pregunta es ¿Qué es clasificación multi-clase? ¿Tiene alguna función de activación que sea más conveniente de usar? La clasificación multiclases es cuando tenemos más de dos posibles diferentes Output, para ellos tenemos un tipo de función de activación más específica llamada Softmax, funciona de forma parecida a Sigmoid pero tiene un paso adicional, y es colocar tantas Output Neurons como clases se tengan, si tenemos m clases, tendremos m Outputs entonces, sería colocar una última Hidden layer con m Neuronas y esas m Neuronas van a tener la Sigmoid Activation Function, y así el Output de cada uno de ellos será relativo en el rango [0,1] y entonces veremos cada uno de estos m Outputs nos va a dar cuenta de si es o no una característica. Además, veremos que para ver si es o no cada una de las m características una por una, requeriremos que sea una Fully-Connected Network.

Es decir, si fuese un algoritmo de detección de perros, gatos y loros, tendríamos 3 Output Neurons con Sigmoid y una de ellas nos diría información de probabilidad de ser perro o no, la siguiente de gato o no y la última de loro o no.

Posibles preguntas sobre Gradient Descent

Una pregunta que podemos hacernos es cuál es la diferencia entre Gradient Descent y Stochastic Gradient Descent. El tipo de Gradient Descent que hemos estudiado es Stochastic Gradient Descent, ya que hemos hecho la Forward Propagation, Backward Propagation, volviendo a iterar el proceso variando los pesos y el bias, el Gradient Descent consiste en lo mismo pero a todo el conjunto de entrenamiento y encontrar el error promedio e iterar. Otro posible método es coger fragmentos pequeños del Input o del conjunto de entrenamiento y es Mini Batch Gradient Descent y realizar exactamente lo mismo.

La diferencia entre Stochastic Gradient Descent y Gradient Descent es que SGD varía rápidamente los parámetros pero de forma muy abrupta y puede tener problemas para converger mientras que Gradient Descent varía lentamente pero converge de forma apropiada, es más lento porque actualiza todo el conjunto de training set o entrenamiento.

Épocas/Epoch

Una epoch es en ANN un ciclo a todo el conjunto entero de training data, es diferente de las iteraciones, las iteraciones van a ser las veces que tratemos con uno o varios datos pero no con su totalidad.

Hiperparámetros

Como una serie de recomendaciones veremos siempre tendremos una única Input Layer, Hidden Layers depende del problema concreto pero típicamente entre 1 y 5, y todas esas Hidden Layers con función de activación ReLU para introducir la no linealidad. Para diferentes tipos de problemas como para un problema binario tendremos 1 Output Neuron, con una función de activación de tipo logística y una función de pérdida de tipo Cross-Entropy, para un problema de clasificación multiclase tendremos tantas Neuronas Output como clases tengamos, función de activación Softmax y una función de pérdida de tipo Cross-Entropy, y finalmente para un problema binario multicaracterística tendremos tantas Neuronas Output como características, con una función de activación de tipo logística y una función de pérdida de tipo Cross-Entropy.

Tenemos tantas Input Neuron como características haya. Tenemos además de 1-5 Hidden Layers, cada una de ellas con 10-100 Neuronas y ellas con función de activación ReLU. Finalmente tendremos tantas Output Neurons como características predichas, sin función de activación y todas ellas con función de pérdida ó error tipo MSE (Mean Squared Error).

Esos hiperparámetros que tendremos que dar, son los primeros que hemos mencionado, el número de Input y Output Neuron, los tipos de funciones de activación y los tipos de funciones de pérdida. Son una serie de valores comunes de estos hiperparámetros.

Keras

Keras es una librería o environment para el deep learning que tiene recursos para poder entrenar casi todos los problemas o modelos de Neural Networks, Keras trabaja a nivel de modelo, pero no maneja las operaciones a bajo nivel. Las manipulaciones de datos o trabajos a bajo nivel requieren de librerías más especializadas como son TensorFlow o Theano o CNTK... Lo bueno de Keras es que puede operar sin estas lower-level libraries, sin embargo, TensorFlow es la librería que más se adapta, más escalable, y con mayor producción.

TensorFlow por ser de bajo nivel, vamos a ver requiere de mayor poder de procesamiento, el cual en un ordenador va a venir dado por el GPU ó por el CPU. Vamos a usar Keras para definir el modelo y vamos a ver que Keras llame a TensorFlow para el backend para que se encargue de las operaciones

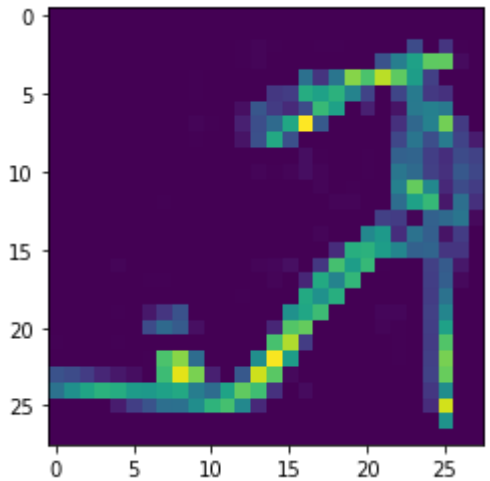
In [3]:

```
##### Implementación/Instalación de Keras y TensorFlow

import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

# !conda install tensorflow

# !conda install pip

# !pip install --upgrade tensorflow==2.0.0rc1

import tensorflow as tf
from tensorflow import keras

#keras.__version__
# '2.4.0'

#tf.__version__
# '2.3.1'
```

In [4]:

```
##### Problema de Clasificación

import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras

# Vamos a tener un dataset de 60.000 imágenes para el train y 10.000 para el test. Las
 imágenes serán de 28x28 píxeles greyscale
# en una escala de 0 a 255. Van a ser 10 tipos de prendas o accesorios y queremos disti
nguirlos
# El dataset es muy conocido que está en keras, es el de 'fashion articles'

fashion_mnist = keras.datasets.fashion_mnist

(x_train_full, y_train_full), (x_test_full, y_test_full) = fashion_mnist.load_data()

object_number = 8

plt.imshow(x_train_full[object_number])
# Para poder visualizar las variables, las imágenes realmente

y_train_full[object_number]
# Es la clase y nos devuelve un número, correspondiente cada uno a una clase

# Creamos una lista de las labels
class_names = ['T-shirt/Top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt'
, 'Sneaker', 'Bag', 'Ankle Boot']

#Y así podemos ver a qué se corresponde
class_names[y_train_full[object_number]]

print('This item is a ', class_names[y_train_full[object_number]] )

# x_train_full[object_number]
# Nos va a dar una descripción numérica de esos píxeles que conforman las imágenes y se
rán arrays de números de 0 a 255
```

```
This item is a  Sandal
```



In [5]:

```
##### DATA NORMALIZATION AND TRAIN-TEST SPLIT

# Se divide entre 255 para normalizar ya que eran arrays de 0 a 255 para ir en esa grey scale
# Solo normalizamos x_train_full y x_test_full porque son las imágenes, recordemos y_train_full y y_test_full son las labels
x_train_n = x_train_full/255.0
x_test_n = x_test_full/255.0

# En un caso general de ML no sabríamos siempre la escala y deberíamos normalizarlo haciendo la suma y dividiendo por la
# desviación estándar, en este caso no es necesario por saber que es entre 0 y 255

# Y como vamos a usar Gradient Descent es que necesitamos normalizarlo y por ello dividimos por 255.0 para tenerlos entre 0 y 1,
# se divide por 255.0 y no 255 por si acaso es el resultado un entero o un real dependiendo de la versión de Python pero por si
# acaso así evitamos problemas

# Ahora vamos a dividirlo en Train-Test-Validation cogiendo los primeros 5.000 de test para validation

# Validation con 5.000 data
x_val = x_train_n[:5000]
y_val = y_train_full[:5000]

# Train con 55.000 valores
x_train = x_train_n[5000:]
y_train = y_train_full[5000:]

# Test con 10.000 valores
x_test = x_test_n
y_test = y_test_full
```

Model Creation

Vamos a ir con este problema de clasificación de imágenes, hay dos formas de crear un ANN en Keras:

1) Sequential model API: muy directa y muy sencilla, modelos sencillos layer por layer

2) Functional API: algo más complicada pero con mayor flexibilidad, como que una de las hidden layer tenga como input el de algunas capas atrás

In [6]:

```
##### 1) Sequential Model API for a Classification Problem

# Lo primero vamos a definir random_seed 42 para poder replicar el mismo modelo todas l
as veces
np.random.seed(42)
tf.random.set_seed(42)

# La estructura del modelo en este problema particular va a consistir en unos datos que
van a ser 28x28 pixeles.
# Esos pixeles los vamos a llevar a la Input Layer, después vamos a colocar 2 Hidden La
yer con función de activación
# de tipo ReLU, y finalmente el Output con 10 características/labels que queremos prede
cir, la Output Neuron tendrá
# una función de activación tipo Softmax

# Lo primero que tenemos que hacer es convertir esa matriz 2D que son los 28x28 píxeles
en un array 1D de 784(=28*28) píxeles

model = keras.models.Sequential()

model.add(keras.layers.Flatten(input_shape = [28,28]))

model.add(keras.layers.Dense(100, activation = 'relu'))

model.add(keras.layers.Dense(100, activation = 'relu'))

model.add(keras.layers.Dense(10, activation = 'softmax'))

# La primera es para definir el modelo como Sequential Model API, la siguiente es para
 convertir de 2D 28x28 a 1D 784 píxeles
# las dos siguientes son las Hidden Layers, como bien sabemos función de activación ReL
U, y unas 100 Neuronas por capa, y
# finalmente la Output layer con 10 Neuronas, una por cada característica y función de
 activación Softmax

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 100)               78500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 89,610
Trainable params: 89,610
Non-trainable params: 0
_________________________________________________________________
```

In [7]:

```
# None significa "No limit in the input data"

#!pip uninstall pydot
#!pip uninstall pydotplus
#!pip uninstall graphviz

#!pip install pydot
#!pip install pydotplus
#!pip install pydotprint
#!pip install graphviz
import pydot

keras.utils.plot_model(model)
```

```
('Failed to import pydot. You must `pip install pydot` and install graphvi
z (https://graphviz.gitlab.io/download/), ', 'for `pydotprint` to work.')
```

In [8]:

```
weights, biases = model.layers[1].get_weights()
print('weigths.shape:',weights.shape,'& biases.shape:', biases.shape)
```

```
weigths.shape: (784, 100) & biases.shape: (100,)
```

In [9]:

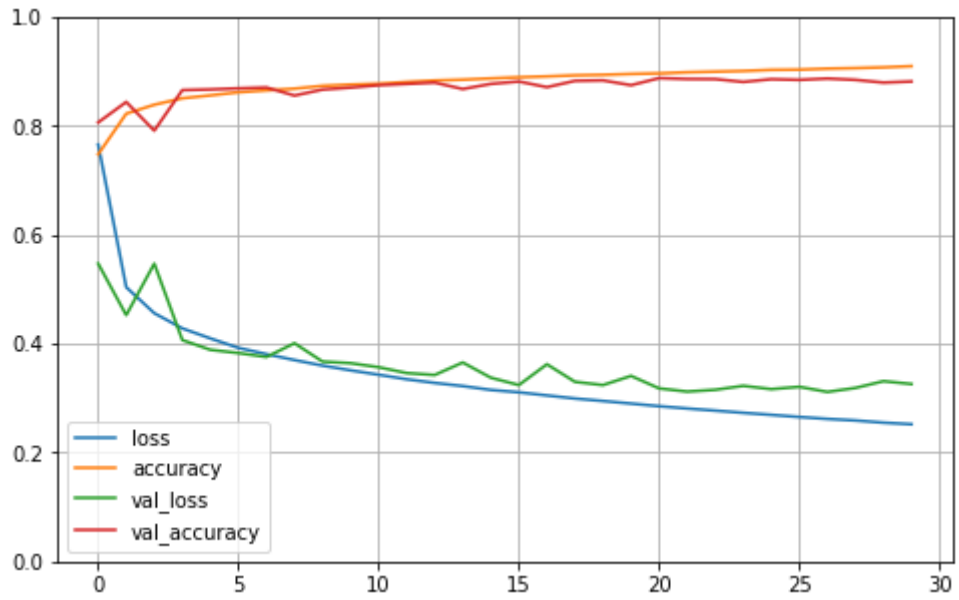
```
# Ahora vamos a ver que antes de entrenar el modelo, tenemos que ver la tasa de aprendi
zaje

model.compile(loss = 'sparse_categorical_crossentropy', optimizer = 'sgd', metrics = [
'accuracy'])
# loss = 'sparse_categorical_crossentropy'; porque tenemos el data en 10 categorías dif
erentes, si tuviésemos probabilidades
# para cada característica entonces tendríamos que usar 'categorical_crossentropy', si
 fuesen muchas y binarias sería
# 'binary_crossentropy'

# optimizer = 'sgd'; SGD Stochastic Gradient Descent, que le estamos contando a Keras q
ue use un algoritmo de backpropagation

# metrics = ['accuracy'] ; por estar usando clasificación, estaremos usando 'accuracy',
si quisiésemos regresión, deberíamos de
# usar 'Mean Squared Error'

model_history = model.fit(x_train, y_train, epochs = 30, validation_data = (x_val, y_va
l))
```

```
Epoch 1/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.7658 -
accuracy: 0.7479 - val_loss: 0.5471 - val_accuracy: 0.8064
Epoch 2/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.5039 -
accuracy: 0.8225 - val_loss: 0.4525 - val_accuracy: 0.8440
Epoch 3/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.4561 -
accuracy: 0.8389 - val_loss: 0.5471 - val_accuracy: 0.7914
Epoch 4/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.4282 -
accuracy: 0.8507 - val_loss: 0.4068 - val_accuracy: 0.8658
Epoch 5/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.4098 -
accuracy: 0.8561 - val_loss: 0.3888 - val_accuracy: 0.8670
Epoch 6/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3924 -
accuracy: 0.8618 - val_loss: 0.3829 - val_accuracy: 0.8690
Epoch 7/30
1719/1719 [==============================] - 3s 1ms/step - loss: 0.3809 -
accuracy: 0.8653 - val_loss: 0.3757 - val_accuracy: 0.8706
Epoch 8/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3700 -
accuracy: 0.8685 - val_loss: 0.4009 - val_accuracy: 0.8558
Epoch 9/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3598 -
accuracy: 0.8736 - val_loss: 0.3671 - val_accuracy: 0.8668
Epoch 10/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3511 -
accuracy: 0.8759 - val_loss: 0.3642 - val_accuracy: 0.8704
Epoch 11/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3430 -
accuracy: 0.8775 - val_loss: 0.3569 - val_accuracy: 0.8748
Epoch 12/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3347 -
accuracy: 0.8807 - val_loss: 0.3461 - val_accuracy: 0.8772
Epoch 13/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3280 -
accuracy: 0.8834 - val_loss: 0.3426 - val_accuracy: 0.8796
Epoch 14/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3222 -
accuracy: 0.8849 - val_loss: 0.3655 - val_accuracy: 0.8678
Epoch 15/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3151 -
accuracy: 0.8873 - val_loss: 0.3376 - val_accuracy: 0.8772
Epoch 16/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3108 -
accuracy: 0.8894 - val_loss: 0.3241 - val_accuracy: 0.8816
Epoch 17/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3050 -
accuracy: 0.8910 - val_loss: 0.3623 - val_accuracy: 0.8712
Epoch 18/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2993 -
accuracy: 0.8929 - val_loss: 0.3300 - val_accuracy: 0.8828
Epoch 19/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2948 -
accuracy: 0.8939 - val_loss: 0.3241 - val_accuracy: 0.8836
Epoch 20/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2901 -
accuracy: 0.8955 - val_loss: 0.3409 - val_accuracy: 0.8748
Epoch 21/30
```

```
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2852 -
accuracy: 0.8965 - val_loss: 0.3179 - val_accuracy: 0.8878
Epoch 22/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2810 -
accuracy: 0.8985 - val_loss: 0.3121 - val_accuracy: 0.8862
Epoch 23/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2772 -
accuracy: 0.8997 - val_loss: 0.3154 - val_accuracy: 0.8860
Epoch 24/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2731 -
accuracy: 0.9009 - val_loss: 0.3226 - val_accuracy: 0.8808
Epoch 25/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2693 -
accuracy: 0.9030 - val_loss: 0.3166 - val_accuracy: 0.8858
Epoch 26/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2654 -
accuracy: 0.9034 - val_loss: 0.3208 - val_accuracy: 0.8846
Epoch 27/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2619 -
accuracy: 0.9052 - val_loss: 0.3115 - val_accuracy: 0.8870
Epoch 28/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2590 -
accuracy: 0.9062 - val_loss: 0.3187 - val_accuracy: 0.8846
Epoch 29/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2550 -
accuracy: 0.9076 - val_loss: 0.3312 - val_accuracy: 0.8796
Epoch 30/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2521 -
accuracy: 0.9097 - val_loss: 0.3260 - val_accuracy: 0.8816
```

In [10]:

```
# Podemos ver la accuracy crece con cada epoch hasta llegar a un 92.78%, lo cual está
 muy bien, y obviamente es mejor que la
# primera de las accuracy en esa primera epoch de 89.73%

# model_history.params
# model_history.history

import pandas as pd

pd.DataFrame(model_history.history).plot(figsize = (8,5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()

# Con cada época la accuracy, y la validation accuracy crecen con cada época, mientras
 que las pérdidas decrecen.
# También se puede ver el modelo no ha convergido, ambas rectas siguen siendo monótonas
pero no paran, habría que
# aumentar el número de épocas/epochs

# Vamos a aumentar el #epochs de 30 a más
```



In [11]:

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

model.add(keras.layers.Flatten(input_shape = [28,28]))

model.add(keras.layers.Dense(100, activation = 'relu'))

model.add(keras.layers.Dense(100, activation = 'relu'))

model.add(keras.layers.Dense(10, activation = 'softmax'))

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

model_history = model.fit(x_train, y_train, epochs = 100, validation_data = (x_val, y_v
al))
```

```
Epoch 1/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.7926 -
accuracy: 0.7331 - val_loss: 0.5580 - val_accuracy: 0.8064
Epoch 2/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.5122 -
accuracy: 0.8197 - val_loss: 0.4578 - val_accuracy: 0.8482
Epoch 3/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.4622 -
accuracy: 0.8359 - val_loss: 0.5353 - val_accuracy: 0.7982
Epoch 4/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.4344 -
accuracy: 0.8476 - val_loss: 0.4145 - val_accuracy: 0.8612
Epoch 5/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.4158 -
accuracy: 0.8539 - val_loss: 0.4008 - val_accuracy: 0.8624
Epoch 6/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3978 -
accuracy: 0.8598 - val_loss: 0.3925 - val_accuracy: 0.8642
Epoch 7/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3858 -
accuracy: 0.8637 - val_loss: 0.3837 - val_accuracy: 0.8662
Epoch 8/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3748 -
accuracy: 0.8663 - val_loss: 0.4026 - val_accuracy: 0.8538
Epoch 9/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3649 -
accuracy: 0.8705 - val_loss: 0.3743 - val_accuracy: 0.8636
Epoch 10/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3558 -
accuracy: 0.8742 - val_loss: 0.3696 - val_accuracy: 0.8650
Epoch 11/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3474 -
accuracy: 0.8759 - val_loss: 0.3607 - val_accuracy: 0.8710
Epoch 12/100
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3389 -
accuracy: 0.8782 - val_loss: 0.3499 - val_accuracy: 0.8754
Epoch 13/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3321 -
accuracy: 0.8808 - val_loss: 0.3488 - val_accuracy: 0.8754
Epoch 14/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3256 -
accuracy: 0.8831 - val_loss: 0.3671 - val_accuracy: 0.8686
Epoch 15/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3184 -
accuracy: 0.8846 - val_loss: 0.3378 - val_accuracy: 0.8766
Epoch 16/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3135 -
accuracy: 0.8870 - val_loss: 0.3272 - val_accuracy: 0.8822
Epoch 17/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3082 -
accuracy: 0.8888 - val_loss: 0.3618 - val_accuracy: 0.8678
Epoch 18/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.3024 -
accuracy: 0.8911 - val_loss: 0.3358 - val_accuracy: 0.8820
Epoch 19/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2978 -
accuracy: 0.8923 - val_loss: 0.3236 - val_accuracy: 0.8854
Epoch 20/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2929 -
accuracy: 0.8944 - val_loss: 0.3416 - val_accuracy: 0.8756
Epoch 21/100
```

```
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2880 -
accuracy: 0.8960 - val_loss: 0.3177 - val_accuracy: 0.8876
Epoch 22/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2835 -
accuracy: 0.8974 - val_loss: 0.3113 - val_accuracy: 0.8870
Epoch 23/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2798 -
accuracy: 0.8980 - val_loss: 0.3120 - val_accuracy: 0.8910
Epoch 24/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2757 -
accuracy: 0.9007 - val_loss: 0.3179 - val_accuracy: 0.8860
Epoch 25/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2715 -
accuracy: 0.9015 - val_loss: 0.3117 - val_accuracy: 0.8900
Epoch 26/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2678 -
accuracy: 0.9037 - val_loss: 0.3184 - val_accuracy: 0.8844
Epoch 27/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2651 -
accuracy: 0.9050 - val_loss: 0.3046 - val_accuracy: 0.8914
Epoch 28/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2615 -
accuracy: 0.9056 - val_loss: 0.3119 - val_accuracy: 0.8872
Epoch 29/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2571 -
accuracy: 0.9077 - val_loss: 0.3226 - val_accuracy: 0.8840
Epoch 30/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2547 -
accuracy: 0.9087 - val_loss: 0.3238 - val_accuracy: 0.8866
Epoch 31/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2509 -
accuracy: 0.9085 - val_loss: 0.3110 - val_accuracy: 0.8906
Epoch 32/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2478 -
accuracy: 0.9105 - val_loss: 0.3049 - val_accuracy: 0.8928
Epoch 33/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2446 -
accuracy: 0.9110 - val_loss: 0.3139 - val_accuracy: 0.8936
Epoch 34/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2411 -
accuracy: 0.9119 - val_loss: 0.3097 - val_accuracy: 0.8924
Epoch 35/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2379 -
accuracy: 0.9146 - val_loss: 0.2997 - val_accuracy: 0.8918
Epoch 36/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2363 -
accuracy: 0.9146 - val_loss: 0.3078 - val_accuracy: 0.8928
Epoch 37/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2330 -
accuracy: 0.9155 - val_loss: 0.3039 - val_accuracy: 0.8922
Epoch 38/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2297 -
accuracy: 0.9172 - val_loss: 0.2971 - val_accuracy: 0.8956
Epoch 39/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2286 -
accuracy: 0.9176 - val_loss: 0.3020 - val_accuracy: 0.8936
Epoch 40/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2249 -
accuracy: 0.9191 - val_loss: 0.3047 - val_accuracy: 0.8962
Epoch 41/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2220 -
```

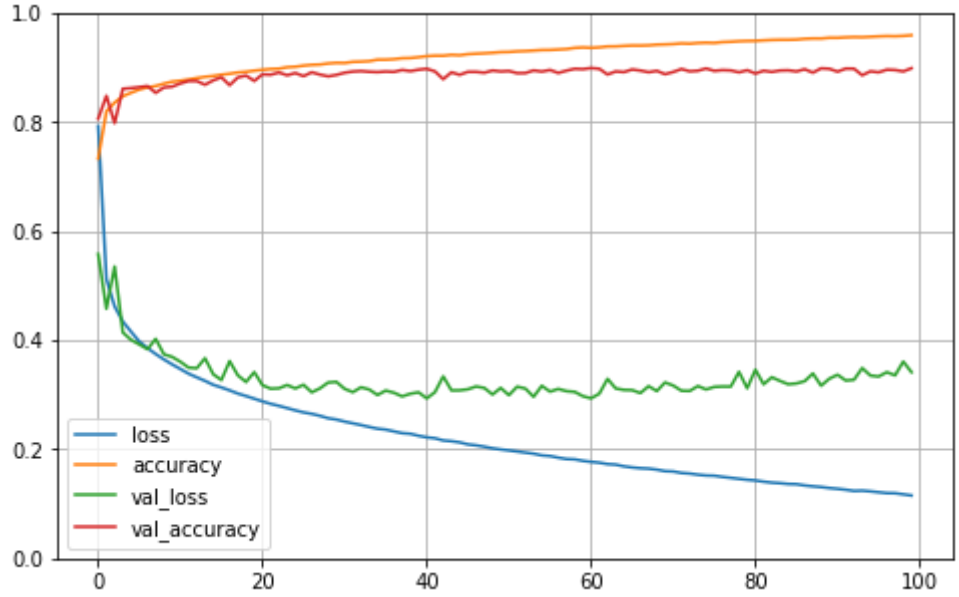
```
accuracy: 0.9210 - val_loss: 0.2935 - val_accuracy: 0.8978
Epoch 42/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2204 -
accuracy: 0.9219 - val_loss: 0.3051 - val_accuracy: 0.8930
Epoch 43/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2163 -
accuracy: 0.9218 - val_loss: 0.3339 - val_accuracy: 0.8788
Epoch 44/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2150 -
accuracy: 0.9234 - val_loss: 0.3081 - val_accuracy: 0.8914
Epoch 45/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2129 -
accuracy: 0.9227 - val_loss: 0.3083 - val_accuracy: 0.8868
Epoch 46/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2091 -
accuracy: 0.9247 - val_loss: 0.3102 - val_accuracy: 0.8922
Epoch 47/100
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2074 -
accuracy: 0.9256 - val_loss: 0.3153 - val_accuracy: 0.8922
Epoch 48/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2049 -
accuracy: 0.9263 - val_loss: 0.3127 - val_accuracy: 0.8906
Epoch 49/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2019 -
accuracy: 0.9272 - val_loss: 0.3006 - val_accuracy: 0.8946
Epoch 50/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1998 -
accuracy: 0.9284 - val_loss: 0.3127 - val_accuracy: 0.8930
Epoch 51/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1981 -
accuracy: 0.9287 - val_loss: 0.2994 - val_accuracy: 0.8938
Epoch 52/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1961 -
accuracy: 0.9299 - val_loss: 0.3150 - val_accuracy: 0.8898
Epoch 53/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1941 -
accuracy: 0.9304 - val_loss: 0.3115 - val_accuracy: 0.8900
Epoch 54/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1921 -
accuracy: 0.9313 - val_loss: 0.2964 - val_accuracy: 0.8948
Epoch 55/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1890 -
accuracy: 0.9322 - val_loss: 0.3165 - val_accuracy: 0.8928
Epoch 56/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1878 -
accuracy: 0.9324 - val_loss: 0.3059 - val_accuracy: 0.8956
Epoch 57/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1848 -
accuracy: 0.9332 - val_loss: 0.3108 - val_accuracy: 0.8902
Epoch 58/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1823 -
accuracy: 0.9340 - val_loss: 0.3068 - val_accuracy: 0.8944
Epoch 59/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1817 -
accuracy: 0.9361 - val_loss: 0.3055 - val_accuracy: 0.8978
Epoch 60/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1790 -
accuracy: 0.9370 - val_loss: 0.2975 - val_accuracy: 0.8972
Epoch 61/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1771 -
accuracy: 0.9363 - val_loss: 0.2936 - val_accuracy: 0.8992
```

```
Epoch 62/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1756 -
accuracy: 0.9371 - val_loss: 0.3021 - val_accuracy: 0.8978
Epoch 63/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1730 -
accuracy: 0.9388 - val_loss: 0.3292 - val_accuracy: 0.8874
Epoch 64/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1720 -
accuracy: 0.9389 - val_loss: 0.3117 - val_accuracy: 0.8930
Epoch 65/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1688 -
accuracy: 0.9398 - val_loss: 0.3090 - val_accuracy: 0.8918
Epoch 66/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1669 -
accuracy: 0.9405 - val_loss: 0.3086 - val_accuracy: 0.8966
Epoch 67/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1654 -
accuracy: 0.9404 - val_loss: 0.3032 - val_accuracy: 0.8946
Epoch 68/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1647 -
accuracy: 0.9409 - val_loss: 0.3159 - val_accuracy: 0.8916
Epoch 69/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1628 -
accuracy: 0.9421 - val_loss: 0.3069 - val_accuracy: 0.8940
Epoch 70/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1600 -
accuracy: 0.9425 - val_loss: 0.3227 - val_accuracy: 0.8880
Epoch 71/100
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1595 -
accuracy: 0.9432 - val_loss: 0.3170 - val_accuracy: 0.8920
Epoch 72/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1567 -
accuracy: 0.9444 - val_loss: 0.3075 - val_accuracy: 0.8974
Epoch 73/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1557 -
accuracy: 0.9439 - val_loss: 0.3071 - val_accuracy: 0.8932
Epoch 74/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1535 -
accuracy: 0.9451 - val_loss: 0.3161 - val_accuracy: 0.8942
Epoch 75/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1521 -
accuracy: 0.9457 - val_loss: 0.3099 - val_accuracy: 0.8984
Epoch 76/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1517 -
accuracy: 0.9452 - val_loss: 0.3145 - val_accuracy: 0.8940
Epoch 77/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1495 -
accuracy: 0.9468 - val_loss: 0.3152 - val_accuracy: 0.8954
Epoch 78/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1478 -
accuracy: 0.9474 - val_loss: 0.3151 - val_accuracy: 0.8948
Epoch 79/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1459 -
accuracy: 0.9484 - val_loss: 0.3421 - val_accuracy: 0.8914
Epoch 80/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1442 -
accuracy: 0.9489 - val_loss: 0.3121 - val_accuracy: 0.8952
Epoch 81/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1430 -
accuracy: 0.9491 - val_loss: 0.3460 - val_accuracy: 0.8892
Epoch 82/100
```

```
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1407 -
accuracy: 0.9499 - val_loss: 0.3194 - val_accuracy: 0.8942
Epoch 83/100
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1390 -
accuracy: 0.9507 - val_loss: 0.3322 - val_accuracy: 0.8922
Epoch 84/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1380 -
accuracy: 0.9510 - val_loss: 0.3255 - val_accuracy: 0.8946
Epoch 85/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1365 -
accuracy: 0.9511 - val_loss: 0.3194 - val_accuracy: 0.8950
Epoch 86/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1361 -
accuracy: 0.9514 - val_loss: 0.3209 - val_accuracy: 0.8930
Epoch 87/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1338 -
accuracy: 0.9528 - val_loss: 0.3251 - val_accuracy: 0.8970
Epoch 88/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1322 -
accuracy: 0.9534 - val_loss: 0.3391 - val_accuracy: 0.8908
Epoch 89/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1309 -
accuracy: 0.9533 - val_loss: 0.3168 - val_accuracy: 0.8986
Epoch 90/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1290 -
accuracy: 0.9547 - val_loss: 0.3296 - val_accuracy: 0.8978
Epoch 91/100
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1277 -
accuracy: 0.9547 - val_loss: 0.3368 - val_accuracy: 0.8926
Epoch 92/100
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1259 -
accuracy: 0.9555 - val_loss: 0.3262 - val_accuracy: 0.8982
Epoch 93/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1238 -
accuracy: 0.9560 - val_loss: 0.3276 - val_accuracy: 0.8980
Epoch 94/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1243 -
accuracy: 0.9558 - val_loss: 0.3490 - val_accuracy: 0.8860
Epoch 95/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1230 -
accuracy: 0.9565 - val_loss: 0.3353 - val_accuracy: 0.8932
Epoch 96/100
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1213 -
accuracy: 0.9573 - val_loss: 0.3336 - val_accuracy: 0.8914
Epoch 97/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1198 -
accuracy: 0.9578 - val_loss: 0.3416 - val_accuracy: 0.8962
Epoch 98/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1196 -
accuracy: 0.9575 - val_loss: 0.3357 - val_accuracy: 0.8956
Epoch 99/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1175 -
accuracy: 0.9582 - val_loss: 0.3609 - val_accuracy: 0.8930
Epoch 100/100
1719/1719 [==============================] - 3s 2ms/step - loss: 0.1154 -
accuracy: 0.9594 - val_loss: 0.3416 - val_accuracy: 0.8988
```

In [12]:

```
import pandas as pd

pd.DataFrame(model_history.history).plot(figsize = (8,5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



In [14]:

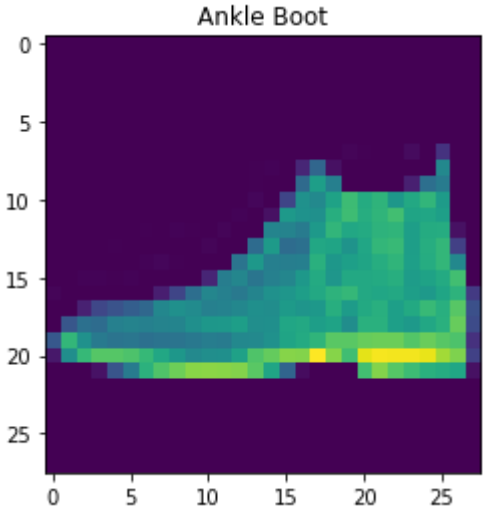
```
# Ahora vamos a ver cómo evaluar el modelo y cómo predecir clases usando el modelo

print('loss, accuracy: ', model.evaluate(x_test, y_test))
```

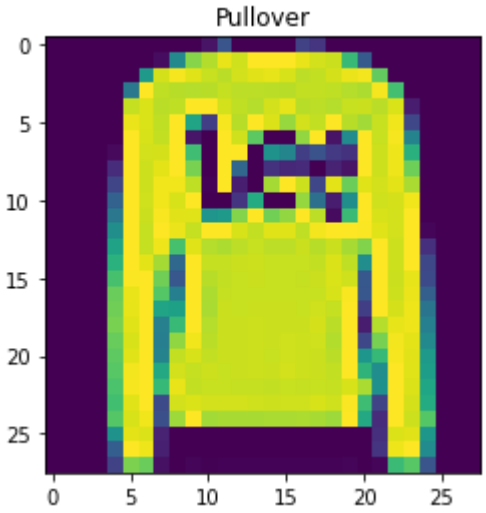
```
313/313 [==============================] - 0s 1ms/step - loss: 0.3864 - ac
curacy: 0.8871
loss, accuracy:  [0.3864385485649109, 0.8870999813079834]
```

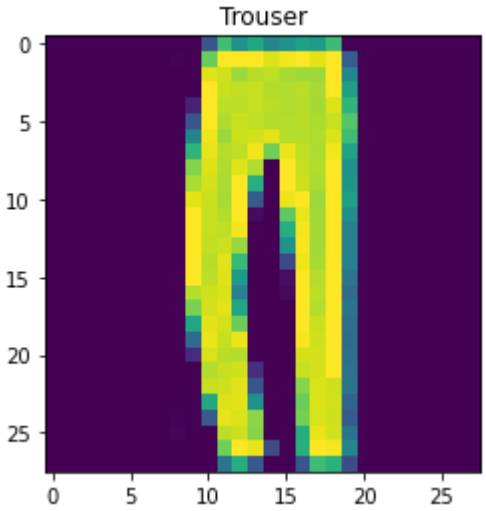
In [15]:

```
# Vamos a coger ahora "nuevos" datos y ver qué clase predice

x_new = x_test[:3]

y_pred_new = model.predict_classes(x_new)
y_pred_new

print('The objects are:',np.array(class_names)[y_pred_new], '\n and the images are')

plt.imshow(x_new[0])
plt.title(np.array(class_names)[y_pred_new[0]])
plt.show()

plt.imshow(x_new[1])
plt.title(np.array(class_names)[y_pred_new[1]])
plt.show()

plt.imshow(x_new[2])
plt.title(np.array(class_names)[y_pred_new[2]])
plt.show()

# Y podemos ver funciona
```

```
WARNING:tensorflow:From <ipython-input-15-a9c0f45ba607>:5: Sequential.pred
ict_classes (from tensorflow.python.keras.engine.sequential) is deprecated
and will be removed after 2021-01-01.
Instructions for updating:
Please use instead:* `np.argmax(model.predict(x), axis=-1)`,   if your mod
el does multi-class classification   (e.g. if it uses a `softmax` last-lay
er activation).* `(model.predict(x) > 0.5).astype("int32")`,   if your mod
el does binary classification   (e.g. if it uses a `sigmoid` last-layer ac
tivation).
The objects are: ['Ankle Boot' 'Pullover' 'Trouser']
 and the images are
```






Trouser
0
5
10
15
20
25
0
5
10
15
20
25

In [16]:

```
##### 1) Sequential Model API for a Regression Problem

import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib as mlp
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras

# Un dataset muy común para la regresión, el objetivo aquí es la predicción del precio
 de una casa usando diferentes variables
# independientes
from sklearn.datasets import fetch_california_housing
# Un dataset de 20.640 datos. Son 8 atributos o características diferentes

housing = fetch_california_housing()

#print('feature_names:',housing.feature_names)
# Las diferentes variables que observamos son 'target', y los nombres de los atributos
 son los siguientes
# {'MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup', 'Latitude',
'Longitude'}

# MedInc: median income in block, HouseAge: media de las edades de las casas, AveRooms
 y AveBedrms como promedio de habitaciones
# y de dormitorios, Population como población promedio, AveOccup como ocupación promedi
o de esas casas y las últimas como la ubi

# La variable objetivo es la media de la casa en unidades de 100.000 distritos de Calif
ornia

# Separamos en train, test y val usando un truquito, primero separar en train_full y te
st y de train_full separarlo en train y
# val
from sklearn.model_selection import train_test_split

# como default es un 25%, así tendremos un 75%trainfull y 25%test
x_train_full, x_test, y_train_full, y_test = train_test_split(housing.data, housing.tar
get, random_state = 42)

# ahora tendremos un 25% del 75% que es 75%/4=18.75% del val y por ello  un 61.25% trai
n
x_train, x_val, y_train, y_val = train_test_split(x_train_full, y_train_full, random_st
ate = 42)

# Y ahora tenemos:
# x_train, x_test, x_val
# y_train, y_test, y_val

# Procedemos a normalizar los datos
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
x_val = scaler.transform(x_val)
```

```
x_train.shape
# (11610, 8)
# Tenemos 11610 datos en las 8 características que hemos mencionado, son solo 11610 ya
 que es un 61.25% de los 20.640 datos


# Lo siguiente es que vamos a definir random_seed 42 para poder replicar el mismo model
o todas las veces
np.random.seed(42)
tf.random.set_seed(42)
```

In [17]:

```
# Vamos ahora a generar el modelo de ANN para regresión con 2 Hidden Layers y sin funci
ón de activación en la única Output
# Neuron y le vamos a pasar también en la primera de estas Input Layer el número de car
acterísticas o variables independientes.
# Otra forma más genérica de escribirlo sería:

# 'num = x_train.shape[:1]
# model.add(keras.layers.Dense(30, activation = 'relu', input_shape = [num]))'

model = keras.models.Sequential()

model.add(keras.layers.Dense(30, activation = 'relu', input_shape = [8]))

model.add(keras.layers.Dense(30, activation = 'relu'))

model.add(keras.layers.Dense(1))

model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 30)                270
_________________________________________________________________
dense_7 (Dense)              (None, 30)                930
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 31
=================================================================
Total params: 1,231
Trainable params: 1,231
Non-trainable params: 0
_________________________________________________________________
```

In [18]:

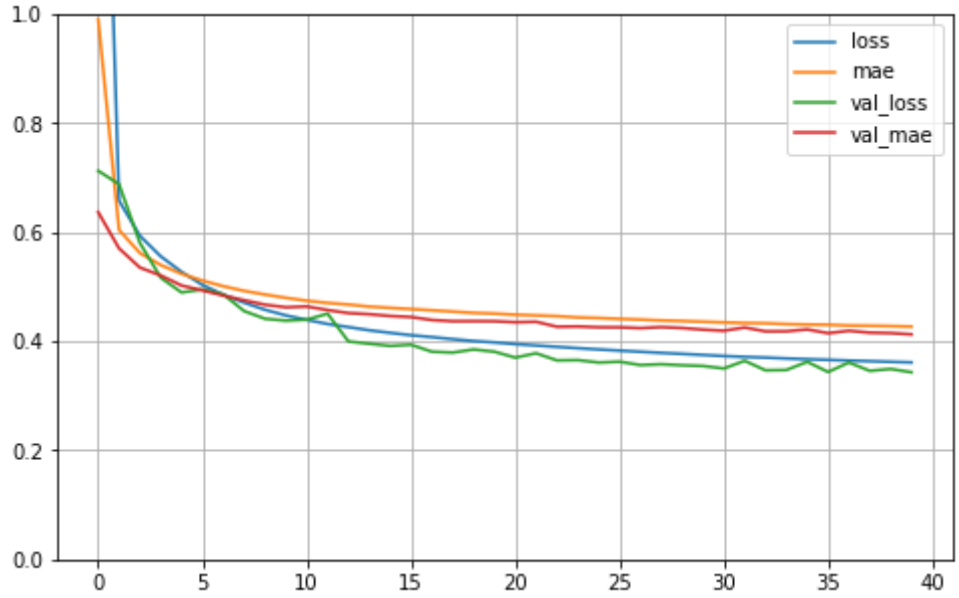
```
model.compile(loss = 'mean_squared_error', optimizer = keras.optimizers.SGD(lr = 1e-3),
metrics = ['mae'])
# lr: el learning rate
# mae: Mean Absolute Error

model_history = model.fit(x_train, y_train, epochs = 40, validation_data = (x_val, y_va
l))
```

```
Epoch 1/40
363/363 [==============================] - 1s 2ms/step - loss: 1.8866 - ma
e: 0.9900 - val_loss: 0.7126 - val_mae: 0.6368
Epoch 2/40
363/363 [==============================] - 0s 1ms/step - loss: 0.6577 - ma
e: 0.6042 - val_loss: 0.6880 - val_mae: 0.5704
Epoch 3/40
363/363 [==============================] - 0s 1ms/step - loss: 0.5934 - ma
e: 0.5618 - val_loss: 0.5803 - val_mae: 0.5352
Epoch 4/40
363/363 [==============================] - 0s 1ms/step - loss: 0.5557 - ma
e: 0.5398 - val_loss: 0.5166 - val_mae: 0.5207
Epoch 5/40
363/363 [==============================] - 0s 1ms/step - loss: 0.5272 - ma
e: 0.5237 - val_loss: 0.4895 - val_mae: 0.5022
Epoch 6/40
363/363 [==============================] - 1s 1ms/step - loss: 0.5033 - ma
e: 0.5113 - val_loss: 0.4951 - val_mae: 0.4934
Epoch 7/40
363/363 [==============================] - 0s 995us/step - loss: 0.4854 -
mae: 0.5010 - val_loss: 0.4861 - val_mae: 0.4838
Epoch 8/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4709 - ma
e: 0.4924 - val_loss: 0.4554 - val_mae: 0.4753
Epoch 9/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4578 - ma
e: 0.4857 - val_loss: 0.4413 - val_mae: 0.4671
Epoch 10/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4474 - ma
e: 0.4797 - val_loss: 0.4379 - val_mae: 0.4623
Epoch 11/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4393 - ma
e: 0.4744 - val_loss: 0.4396 - val_mae: 0.4638
Epoch 12/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4318 - ma
e: 0.4703 - val_loss: 0.4507 - val_mae: 0.4573
Epoch 13/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4261 - ma
e: 0.4674 - val_loss: 0.3997 - val_mae: 0.4517
Epoch 14/40
363/363 [==============================] - 0s 963us/step - loss: 0.4202 -
mae: 0.4636 - val_loss: 0.3956 - val_mae: 0.4497
Epoch 15/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4155 - ma
e: 0.4613 - val_loss: 0.3916 - val_mae: 0.4464
Epoch 16/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4112 - ma
e: 0.4591 - val_loss: 0.3937 - val_mae: 0.4445
Epoch 17/40
363/363 [==============================] - 1s 2ms/step - loss: 0.4077 - ma
e: 0.4569 - val_loss: 0.3809 - val_mae: 0.4390
Epoch 18/40
363/363 [==============================] - 1s 2ms/step - loss: 0.4040 - ma
e: 0.4545 - val_loss: 0.3793 - val_mae: 0.4368
Epoch 19/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4004 - ma
e: 0.4521 - val_loss: 0.3850 - val_mae: 0.4369
Epoch 20/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3980 - ma
e: 0.4508 - val_loss: 0.3809 - val_mae: 0.4368
Epoch 21/40
```

```
363/363 [==============================] - 0s 1ms/step - loss: 0.3949 - ma
e: 0.4488 - val_loss: 0.3701 - val_mae: 0.4350
Epoch 22/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3924 - ma
e: 0.4474 - val_loss: 0.3781 - val_mae: 0.4358
Epoch 23/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3898 - ma
e: 0.4462 - val_loss: 0.3650 - val_mae: 0.4267
Epoch 24/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3874 - ma
e: 0.4438 - val_loss: 0.3655 - val_mae: 0.4271
Epoch 25/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3851 - ma
e: 0.4427 - val_loss: 0.3611 - val_mae: 0.4257
Epoch 26/40
363/363 [==============================] - 1s 1ms/step - loss: 0.3829 - ma
e: 0.4412 - val_loss: 0.3626 - val_mae: 0.4256
Epoch 27/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3809 - ma
e: 0.4398 - val_loss: 0.3564 - val_mae: 0.4242
Epoch 28/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3788 - ma
e: 0.4382 - val_loss: 0.3579 - val_mae: 0.4259
Epoch 29/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3769 - ma
e: 0.4373 - val_loss: 0.3561 - val_mae: 0.4244
Epoch 30/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3750 - ma
e: 0.4359 - val_loss: 0.3548 - val_mae: 0.4214
Epoch 31/40
363/363 [==============================] - 1s 1ms/step - loss: 0.3730 - ma
e: 0.4346 - val_loss: 0.3502 - val_mae: 0.4193
Epoch 32/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3714 - ma
e: 0.4336 - val_loss: 0.3642 - val_mae: 0.4253
Epoch 33/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3701 - ma
e: 0.4330 - val_loss: 0.3468 - val_mae: 0.4182
Epoch 34/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3685 - ma
e: 0.4316 - val_loss: 0.3474 - val_mae: 0.4183
Epoch 35/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3672 - ma
e: 0.4307 - val_loss: 0.3624 - val_mae: 0.4218
Epoch 36/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3660 - ma
e: 0.4304 - val_loss: 0.3438 - val_mae: 0.4149
Epoch 37/40
363/363 [==============================] - 1s 1ms/step - loss: 0.3647 - ma
e: 0.4289 - val_loss: 0.3611 - val_mae: 0.4193
Epoch 38/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3636 - ma
e: 0.4285 - val_loss: 0.3459 - val_mae: 0.4158
Epoch 39/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3624 - ma
e: 0.4279 - val_loss: 0.3492 - val_mae: 0.4150
Epoch 40/40
363/363 [==============================] - 1s 1ms/step - loss: 0.3613 - ma
e: 0.4270 - val_loss: 0.3433 - val_mae: 0.4127
```

In [19]:

```
import pandas as pd

pd.DataFrame(model_history.history).plot(figsize = (8,5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()

# Se pueden ver tanto loss como mae van decreciendo
```



In [20]:

```
# Ahora vamos a ver cómo evaluar el modelo y cómo predecir clases usando el modelo

print('mae_test: ', model.evaluate(x_test, y_test))
```

```
162/162 [==============================] - 0s 846us/step - loss: 0.3601 -
mae: 0.4246
mae_test:  [0.3601399064064026, 0.42462432384490967]
```

Las 1) Sequential Model API tienen muchas limitaciones empezando por aquella que las define, son secuenciales. Para poder conseguir arquitecturas más complejas vamos a necesitar las 2) Functional API. Nos van a permitir detalles en la arquitectura como van a poder ser la posibilidad de múltiples Inputs y/ó múltiples Outputs. Para poder realizar todo esto, se usan las Functional API, que vamos a usar en forma de funciones y vamos a construir cada una de ellas.

La ANN que vamos a construir sigue el siguiente esquema

Input -> Hidden Layer 1 -> Hidden Layer 2 -> Concat -> Output

Y además Input -> Concat

Es decir, tenemos en Concat dos entradas, la primera entrada será el Input crudo y una segunda entrada que es el mismo Input tras las dos Hidden Layer. Aquí el Input diremos va a avanzar en profundidad y en ancho, y no va a ser posible en Sequential API, es aquí donde entran en juego la importancia de esta arquitectura, ya que nos permite la combinación de ese análisis de patrones internos Deep por medio de esa segunda entrada, junto con las reglas sencillas, lo cual nos va a permitir no perder esas informaciones sencillas.

In [21]:

```
##### 2) Functional API

# Lo primero vamos a limpiar keras para poder operar a continuación en el mismo código

keras.backend.clear_session()

##### VAMOS A USAR EL MISMO DATASET QUE ANTES, ASÍ QUE LA PREPARACIÓN ES LA MISMA

import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib as mlp
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras

from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()

from sklearn.model_selection import train_test_split

x_train_full, x_test, y_train_full, y_test = train_test_split(housing.data, housing.tar
get, random_state = 42)

x_train, x_val, y_train, y_val = train_test_split(x_train_full, y_train_full, random_st
ate = 42)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
x_val = scaler.transform(x_val)

x_train.shape

np.random.seed(42)
tf.random.set_seed(42)

##### HASTA AQUÍ LA PREPARACIÓN DE LOS DATOS
```

In [22]:

```
# Vamos a crear las capas una por una como habíamos anticipado

# La ANN que vamos a construir sigue el siguiente esquema

# Input -> Hidden Layer 1 -> Hidden Layer 2 -> Concat -> Output

# Y además Input -> Concat

input_ = keras.layers.Input(shape = x_train.shape[1:])
# Las 8 variables independientes que van a ser la entrada

hidden1 = keras.layers.Dense(30, activation = 'relu')(input_)
# Estamos añadiendo al final el (input_) como si input_ fuese un argumento de la funció
n hidden1
# Esto es la conexión API, ese Input -> Hidden Layer 1

hidden2 = keras.layers.Dense(30, activation = 'relu')(hidden1)
# Ahora en este caso para indicar ese argumento (hidden1) en la función hidden2 es: Hid
den Layer 1  -> Hidden Layer 2

concat = keras.layers.concatenate([hidden2, input_])
# Vamos a combinar aquí el 'input_' como la primera entrada y 'output' de hidden2 como
 esa segunda entrada y vamos a hacer
# esa combinación de esas entradas, es un proceso de 'merging', es el Hidden Layer 2 ->
Concat & Input -> Concat

output = keras.layers.Dense(1)(concat)
# Queremos finalmente que pase por una única neurona que nos de el resultado ó Output f
inal

model = keras.models.Model(inputs = [input_], outputs = [output])
# Creamos un Model objetct indicando así el Input y el Output.

model.summary()
```

```
Model: "functional_1"
__________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected
to
==========================================================================================
input_1 (InputLayer)         [(None, 8)]               0
__________________________________________________________________________________________
dense (Dense)                (None, 30)                270         input_1
[0][0]
__________________________________________________________________________________________
dense_1 (Dense)              (None, 30)                930         dense[0]
[0]
__________________________________________________________________________________________
concatenate (Concatenate)    (None, 38)                0           dense_1
[0][0]
                                                                   input_1
[0][0]
__________________________________________________________________________________________
dense_2 (Dense)              (None, 1)                 39          concatena
te[0][0]
==========================================================================================
Total params: 1,239
Trainable params: 1,239
Non-trainable params: 0
__________________________________________________________________________________________
```

In [23]:

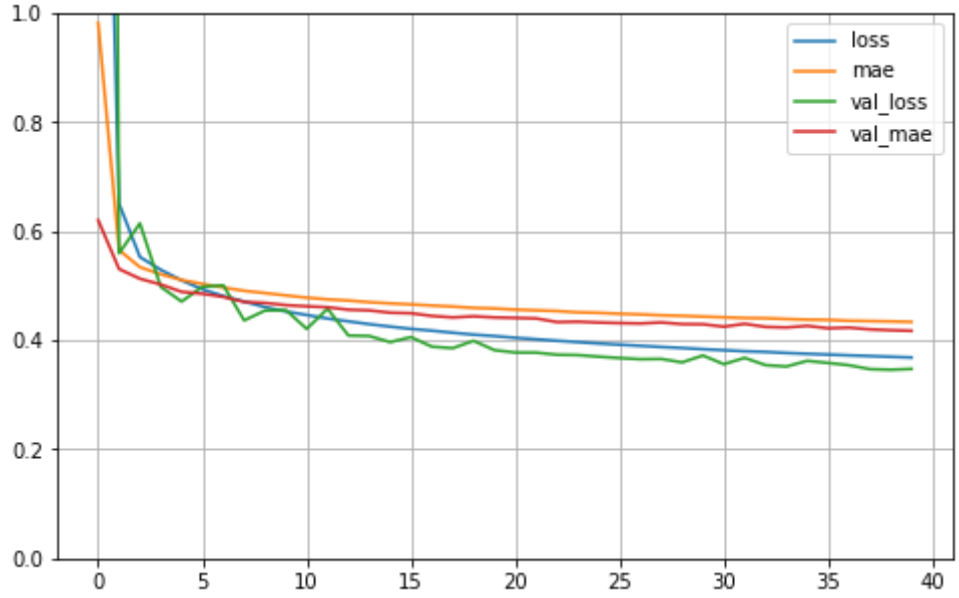
```
##### Exactamente igual que antes

model.compile(loss = 'mean_squared_error', optimizer = keras.optimizers.SGD(lr = 1e-3),
metrics = ['mae'])
# lr: el learning rate
# mae: Mean Absolute Error

model_history = model.fit(x_train, y_train, epochs = 40, validation_data = (x_val, y_va
l))
```

```
Epoch 1/40
363/363 [==============================] - 1s 2ms/step - loss: 2.1327 - ma
e: 0.9818 - val_loss: 7.1207 - val_mae: 0.6202
Epoch 2/40
363/363 [==============================] - 0s 1ms/step - loss: 0.6506 - ma
e: 0.5654 - val_loss: 0.5594 - val_mae: 0.5307
Epoch 3/40
363/363 [==============================] - 0s 1ms/step - loss: 0.5527 - ma
e: 0.5342 - val_loss: 0.6144 - val_mae: 0.5129
Epoch 4/40
363/363 [==============================] - 0s 926us/step - loss: 0.5293 -
mae: 0.5210 - val_loss: 0.4980 - val_mae: 0.5023
Epoch 5/40
363/363 [==============================] - 1s 2ms/step - loss: 0.5099 - ma
e: 0.5105 - val_loss: 0.4711 - val_mae: 0.4889
Epoch 6/40
363/363 [==============================] - 1s 2ms/step - loss: 0.4934 - ma
e: 0.5034 - val_loss: 0.4982 - val_mae: 0.4853
Epoch 7/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4811 - ma
e: 0.4968 - val_loss: 0.5007 - val_mae: 0.4796
Epoch 8/40
363/363 [==============================] - 0s 976us/step - loss: 0.4710 -
mae: 0.4907 - val_loss: 0.4365 - val_mae: 0.4703
Epoch 9/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4609 - ma
e: 0.4866 - val_loss: 0.4546 - val_mae: 0.4685
Epoch 10/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4527 - ma
e: 0.4823 - val_loss: 0.4553 - val_mae: 0.4645
Epoch 11/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4464 - ma
e: 0.4782 - val_loss: 0.4210 - val_mae: 0.4623
Epoch 12/40
363/363 [==============================] - 1s 2ms/step - loss: 0.4398 - ma
e: 0.4750 - val_loss: 0.4580 - val_mae: 0.4606
Epoch 13/40
363/363 [==============================] - 1s 2ms/step - loss: 0.4347 - ma
e: 0.4729 - val_loss: 0.4087 - val_mae: 0.4561
Epoch 14/40
363/363 [==============================] - 1s 2ms/step - loss: 0.4296 - ma
e: 0.4699 - val_loss: 0.4078 - val_mae: 0.4548
Epoch 15/40
363/363 [==============================] - 1s 2ms/step - loss: 0.4251 - ma
e: 0.4676 - val_loss: 0.3967 - val_mae: 0.4506
Epoch 16/40
363/363 [==============================] - 1s 2ms/step - loss: 0.4209 - ma
e: 0.4658 - val_loss: 0.4058 - val_mae: 0.4496
Epoch 17/40
363/363 [==============================] - 1s 1ms/step - loss: 0.4176 - ma
e: 0.4637 - val_loss: 0.3883 - val_mae: 0.4447
Epoch 18/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4139 - ma
e: 0.4618 - val_loss: 0.3855 - val_mae: 0.4419
Epoch 19/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4104 - ma
e: 0.4594 - val_loss: 0.3991 - val_mae: 0.4440
Epoch 20/40
363/363 [==============================] - 0s 1ms/step - loss: 0.4078 - ma
e: 0.4583 - val_loss: 0.3820 - val_mae: 0.4419
Epoch 21/40
```

```
363/363 [==============================] - 0s 1ms/step - loss: 0.4045 - ma
e: 0.4563 - val_loss: 0.3776 - val_mae: 0.4410
Epoch 22/40
363/363 [==============================] - 0s 898us/step - loss: 0.4019 -
mae: 0.4549 - val_loss: 0.3774 - val_mae: 0.4400
Epoch 23/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3992 - ma
e: 0.4536 - val_loss: 0.3735 - val_mae: 0.4334
Epoch 24/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3967 - ma
e: 0.4513 - val_loss: 0.3729 - val_mae: 0.4341
Epoch 25/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3942 - ma
e: 0.4503 - val_loss: 0.3700 - val_mae: 0.4326
Epoch 26/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3920 - ma
e: 0.4487 - val_loss: 0.3674 - val_mae: 0.4316
Epoch 27/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3899 - ma
e: 0.4475 - val_loss: 0.3653 - val_mae: 0.4309
Epoch 28/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3877 - ma
e: 0.4458 - val_loss: 0.3658 - val_mae: 0.4327
Epoch 29/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3859 - ma
e: 0.4448 - val_loss: 0.3595 - val_mae: 0.4297
Epoch 30/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3837 - ma
e: 0.4435 - val_loss: 0.3721 - val_mae: 0.4294
Epoch 31/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3818 - ma
e: 0.4422 - val_loss: 0.3562 - val_mae: 0.4250
Epoch 32/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3799 - ma
e: 0.4409 - val_loss: 0.3677 - val_mae: 0.4302
Epoch 33/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3787 - ma
e: 0.4404 - val_loss: 0.3545 - val_mae: 0.4246
Epoch 34/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3768 - ma
e: 0.4389 - val_loss: 0.3520 - val_mae: 0.4235
Epoch 35/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3752 - ma
e: 0.4378 - val_loss: 0.3623 - val_mae: 0.4264
Epoch 36/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3738 - ma
e: 0.4374 - val_loss: 0.3584 - val_mae: 0.4220
Epoch 37/40
363/363 [==============================] - 1s 2ms/step - loss: 0.3724 - ma
e: 0.4358 - val_loss: 0.3543 - val_mae: 0.4232
Epoch 38/40
363/363 [==============================] - 0s 987us/step - loss: 0.3711 -
mae: 0.4352 - val_loss: 0.3471 - val_mae: 0.4200
Epoch 39/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3698 - ma
e: 0.4346 - val_loss: 0.3460 - val_mae: 0.4184
Epoch 40/40
363/363 [==============================] - 0s 1ms/step - loss: 0.3685 - ma
e: 0.4337 - val_loss: 0.3475 - val_mae: 0.4173
```

In [24]:

```
import pandas as pd

pd.DataFrame(model_history.history).plot(figsize = (8,5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()

# Se pueden ver tanto loss como mae van decreciendo
```



In [25]:

```
# Ahora vamos a ver cómo evaluar el modelo y cómo predecir clases usando el modelo

print('mae_test: ', model.evaluate(x_test, y_test))
```

```
162/162 [==============================] - 0s 989us/step - loss: 0.3673 -
mae: 0.4326
mae_test:  [0.3673333525657654, 0.43257638812065125]
```

Selección de Hiperparámetros

Como hemos ido mencionando, son aquellos que nos permiten ir dando diferentes formas, complejidades a los diferentes tipos de perceptrones multinivel.

Número de Hidden Layers

Para una gran mayoría de problemas vamos a ver que podemos empezar con 1 Hidden Layer y obtener buenos resultados o al menos razonables. Sin embargo, a más profunda sea una red (mayor sea el número de Hidden Layers, será más eficiente ese modelo en términos de eficiencia en parámetros, ya que es mejor avanzar en profundidad antes que tener pocas Hidden Layers con muchas neuronas en cada una).

Esto se debe a que los datos del mundo real están basados en estructuras internas jerárquicas y de esta forma será que veremos ayuda a una más rápida convergencia, y también a la generalización para nueva data. Deep Learning -> Transfer Learning, cuando vamos añadiendo capas y podremos ver ese modelo ya entrenado nos va a servir para poder generalizarlo a nuevos datos.

Por lo general si el problema no es muy complejo usaremos 1 ó 2 Hidden Layers, y si es más complejo, iremos añadiendo más.

En cuanto a cuántas Neuronas tendremos que indicar en cada Hidden Layer, una buena recomendación es usar una forma de pirámide, por ejemplo, en la 1ª Hidden Layer podríamos poner 300 Neuronas, en la 2ª pondríamos 200 Neuronas y finalmente en una 3ª pondríamos 100 Neuronas. Sin embargo, esta técnica está cayendo en desuso ya que se ha comprobado que usar un número constante de Hidden Layers obtiene unos resultados de igual calidad o incluso mejores. Así, ese problema de 3 Hidden Layers podríamos resolverlo colocando 150 Neuronas en cada una de esas Hidden Layers, y a medida que veamos no hay buenos resultados, podríamos ir aumentando estos valores.

Lo siguiente que podríamos ver es que no solo vamos a mejorar la calidad con ese incremento en el número de Neuronas, sino que lo conseguiremos mejorar realmente aumentando la profundidad. Y sino, una opción aún mejor podría ser el aumento de profundidad y a la vez de Neuronas por Hidden Layer.

Learning Rate

Empezar con un valor de la mitad del máximo Learning Rate, es decir, un valor grande con el que el algoritmo vaya a diverger, luego dividirlo por 3 e iterar así el proceso hasta obtener los resultados deseados, es decir, hasta que deje de diverger y ese será nuestro valor óptimo de Learning Rate.

Batch Size

Por lo general 20 < batch_size < 32 ; Ya que estaremos buscando este sea rápido y por eso esa cota superior (batch_size < 32), y la cota inferior (20 < batch_size) viene ya que así sacamos partido del software y del hardware ya que así las multiplicaciones de matrices son más óptimas.

Epoch

En vez de tunearlo, usar el "earlystopping" para evitar overfitting.

Recomendaciones Globales para un código cualquiera de ANN

Dónde usar cada tipo de función de activación, cualquiera de ellas nos sirve para clasificación, pero para regresión deberemos usar ReLU, y acerca de en qué neurona colocarlas, Step/Escalón por lo general como Output, Sigmoid/TanH generalmente como Hidden/Output layer y ReLu para las Hidden layers

Como una serie de recomendaciones veremos siempre tendremos una única Input Layer, Hidden Layers depende del problema concreto pero típicamente entre 1 y 5, y todas esas Hidden Layers con función de activación ReLU para introducir la no linealidad. Para diferentes problemas como para un problema binario tendremos 1 Output Neuron, con una función de activación de tipo logística y una función de pérdida de tipo Cross-Entropy, para un problema de clasificación multiclase tendremos tantas Neuronas Output como clases tengamos, función de activación Softmax y una función de pérdida de tipo Cross-Entropy, y finalmente para un problema binario multicaracterística tendremos tantas Neuronas Output como características, con una función de activación de tipo logística y una función de pérdida de tipo Cross-Entropy.

Tenemos tantas Input Neuron como características haya. Tenemos además de 1-5 Hidden Layers, cada una de ellas con 10-100 Neuronas y ellas con función de activación ReLU. Finalmente tendremos tantas Output Neurons como características predichas, sin función de activación y todas ellas con función de pérdida ó error tipo MSE (Mean Squared Error).

Otras serie de recomendaciones surgen en compilar el modelo:

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

loss = 'sparse_categorical_crossentropy'; porque tenemos el data en 10 categorías diferentes, si tuviésemos probabilidades para cada característica entonces tendríamos que usar 'categorical_crossentropy', si fuesen muchas y binarias sería 'binary_crossentropy'

optimizer = 'sgd'; SGD Stochastic Gradient Descent, que le estamos contando a Keras que use un algoritmo de backpropagation

metrics = ['accuracy'] ; por estar usando clasificación, estaremos usando 'accuracy', si quisiésemos regresión, deberíamos de usar 'Mean Squared Error'