Redes neuronales artificiales con Keras

Este notebook contiene todo el código fuente requerido para la solución de los talleres propuestos.

Ejecutar en Google Colab (https://colab.research.google.com/github/ageron/handson-ml2/blob/master/10_neural_nets_with_keras.ipynb)

Configuración

Primero, importemos algunos módulos comunes, asegurémonos de que MatplotLib traza las figuras en línea y preparemos una función para guardar las figuras. También verifiquemos que Python 3.5 o posterior esté instalado (aunque Python 2.x puede funcionar, está obsoleto, por lo que recomendamos encarecidamente que use Python 3 en su lugar), así como Scikit-Learn ≥0.20 y TensorFlow ≥2.0.

In [116]:

```
# Python ≥3.5 es requerido
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 es requerido
import sklearn
assert sklearn.__version__ >= "0.20"

try:
    # %tensorflow_version solo existe en Colab.
    %tensorflow_version 2.x
except Exception:
    pass

# TensorFlow ≥2.0 es requerido
import tensorflow as tf
assert tf.__version__ >= "2.0"

# Importar librerías comunes
import numpy as np
import os

# para que la salida de este notebook sea estable en todas las ejecuciones
np.random.seed(42)

# Para dibujar figuras estéticas
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# En donde almacenar las figuras
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "ann"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)

# Ignorar las advertencias inútiles (consulte el número 5998 de SciPy)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

Perceptrones

Nota: establecemos `max_iter` y `tol` explícitamente para evitar advertencias sobre el hecho de que su valor predeterminado cambiará en futuras versiones de Scikit-Learn.

In [117]:
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # largo del pétalo, ancho del pétalo
y = (iris.target == 0).astype(np.int)

per_clf = Perceptron(max_iter=1000, tol=1e-3, random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

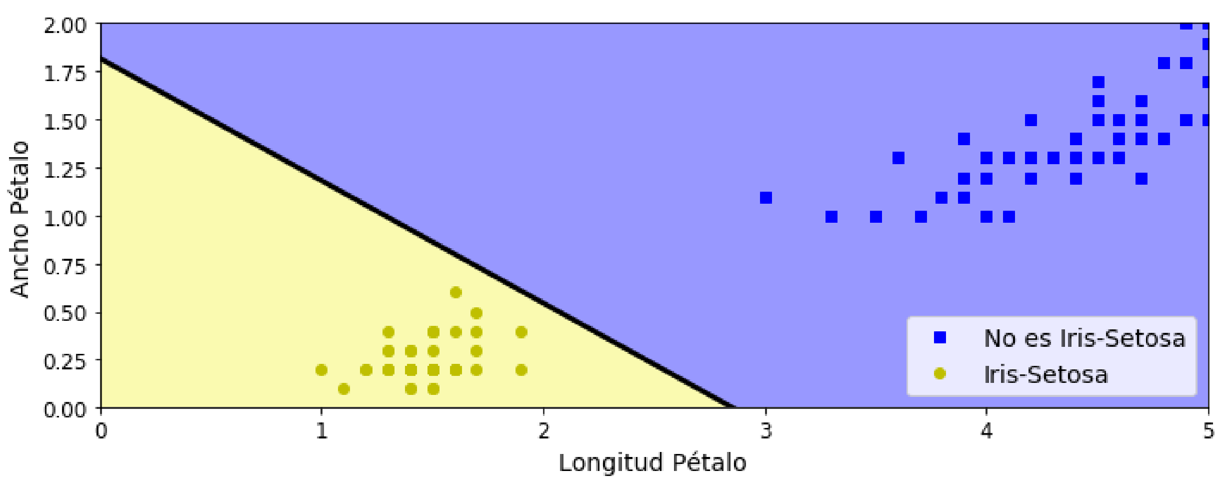
In [118]:
```
y_pred
```

Out[118]:
```
array([1])
```

In [119]:

```
a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

x0, x1 = np.meshgrid(
        np.linspace(axes[0], axes[1], 500).reshape(-1, 1),
        np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]
y_predict = per_clf.predict(X_new)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="No es Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

plt.plot([axes[0], axes[1]], [a * axes[0] + b, a * axes[1] + b], "k-", linew
from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
plt.xlabel("Longitud Pétalo", fontsize=14)
plt.ylabel("Ancho Pétalo", fontsize=14)
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)

save_fig("perceptron_iris_plot")
plt.show()
```

```
Saving figure perceptron_iris_plot
```



In [120]:

```
# Funciones de Activación
```

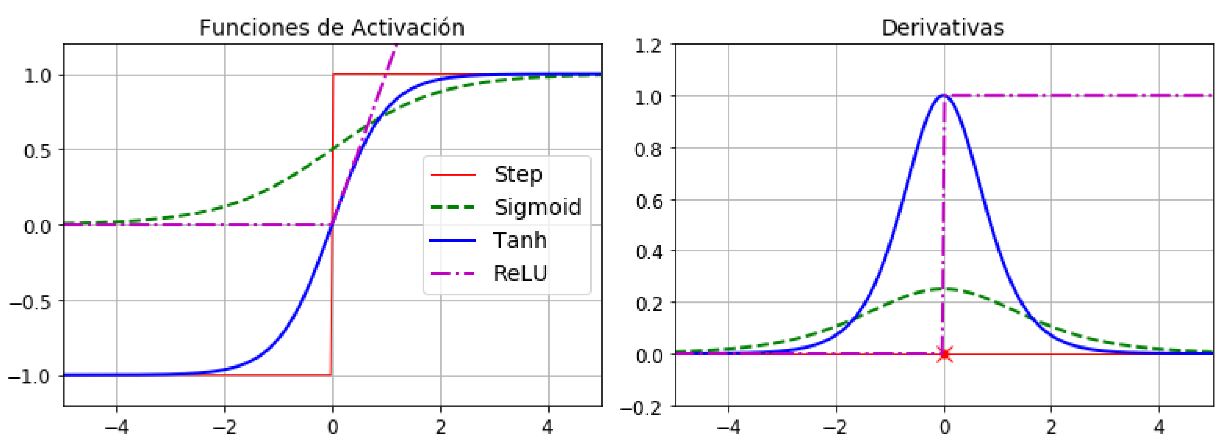
In [121]:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def relu(z):
    return np.maximum(0, z)

def derivative(f, z, eps=0.000001):
    return (f(z + eps) - f(z - eps))/(2 * eps)
```

In [122]:

```
z = np.linspace(-5, 5, 200)

plt.figure(figsize=(11,4))

plt.subplot(121)
plt.plot(z, np.sign(z), "r-", linewidth=1, label="Step")
plt.plot(z, sigmoid(z), "g--", linewidth=2, label="Sigmoid")
plt.plot(z, np.tanh(z), "b-", linewidth=2, label="Tanh")
plt.plot(z, relu(z), "m-.", linewidth=2, label="ReLU")
plt.grid(True)
plt.legend(loc="center right", fontsize=14)
plt.title("Funciones de Activación", fontsize=14)
plt.axis([-5, 5, -1.2, 1.2])

plt.subplot(122)
plt.plot(z, derivative(np.sign, z), "r-", linewidth=1, label="Step")
plt.plot(0, 0, "ro", markersize=5)
plt.plot(0, 0, "rx", markersize=10)
plt.plot(z, derivative(sigmoid, z), "g--", linewidth=2, label="Sigmoid")
plt.plot(z, derivative(np.tanh, z), "b-", linewidth=2, label="Tanh")
plt.plot(z, derivative(relu, z), "m-.", linewidth=2, label="ReLU")
plt.grid(True)
#plt.legend(loc="center right", fontsize=14)
plt.title("Derivativas", fontsize=14)
plt.axis([-5, 5, -0.2, 1.2])

save_fig("activation_functions_plot")
plt.show()
```

```
Saving figure activation_functions_plot
```

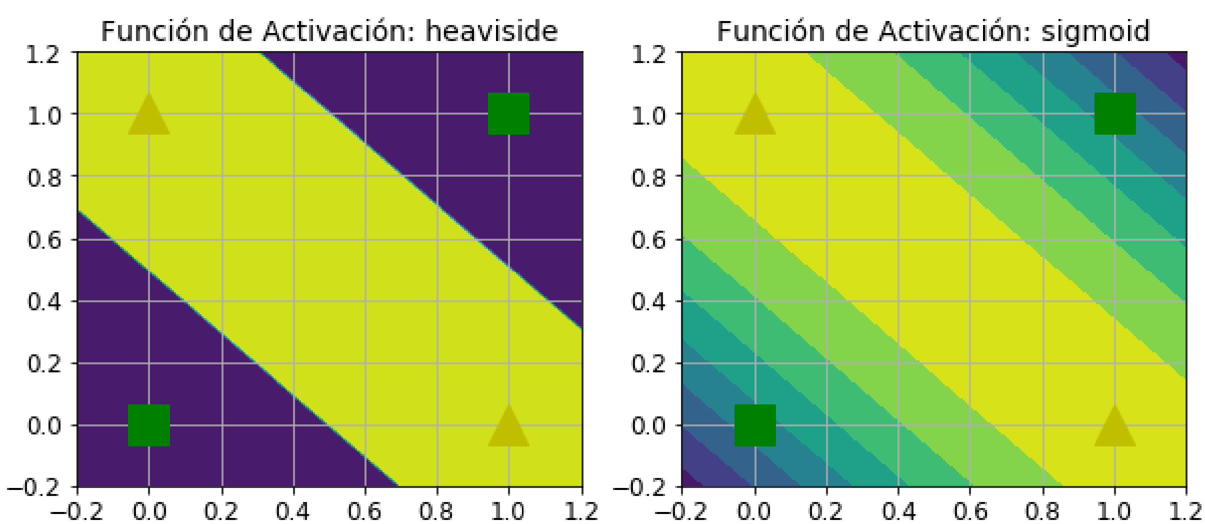


In [123]:

```
def heaviside(z):
    return (z >= 0).astype(z.dtype)

def mlp_xor(x1, x2, activation=heaviside):
    return activation(-activation(x1 + x2 - 1.5) + activation(x1 + x2 - 0.5)
```

In [124]:

```
x1s = np.linspace(-0.2, 1.2, 100)
x2s = np.linspace(-0.2, 1.2, 100)
x1, x2 = np.meshgrid(x1s, x2s)

z1 = mlp_xor(x1, x2, activation=heaviside)
z2 = mlp_xor(x1, x2, activation=sigmoid)

plt.figure(figsize=(10,4))

plt.subplot(121)
plt.contourf(x1, x2, z1)
plt.plot([0, 1], [0, 1], "gs", markersize=20)
plt.plot([0, 1], [1, 0], "y^", markersize=20)
plt.title("Función de Activación: heaviside", fontsize=14)
plt.grid(True)

plt.subplot(122)
plt.contourf(x1, x2, z2)
plt.plot([0, 1], [0, 1], "gs", markersize=20)
plt.plot([0, 1], [1, 0], "y^", markersize=20)
plt.title("Función de Activación: sigmoid", fontsize=14)
plt.grid(True)
```



Construyendo un clasificador de Imágenes

Primero, importemos TensorFlow y Keras.

In [125]:

```
import tensorflow as tf
from tensorflow import keras
```

In [126]:
```
tf.__version__
```

Out[126]: '2.3.1'

In [127]:
```
keras.__version__
```

Out[127]: '2.4.0'

Comencemos cargando el conjunto de datos de moda MNIST. Keras tiene una serie de funciones para cargar conjuntos de datos populares en `keras.datasets` . El conjunto de datos ya está dividido entre un conjunto de entrenamiento y un conjunto de prueba, pero puede ser útil dividir el conjunto de entrenamiento más para tener un conjunto de validación:

In [128]:
```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

El conjunto de entrenamiento contiene 60.000 imágenes en escala de grises, cada una de 28x28 píxeles:

In [129]:
```
X_train_full.shape
```

Out[129]: (60000, 28, 28)

Cada intensidad de píxel se representa como un byte (de 0 a 255):

In [130]:
```
X_train_full.dtype
```

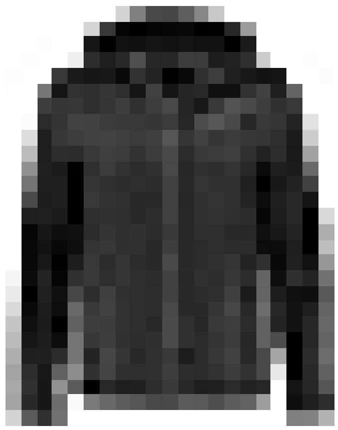
Out[130]: dtype('uint8')

Dividamos el conjunto de entrenamiento completo en un conjunto de validación y un conjunto de entrenamiento (más pequeño). También escalamos las intensidades de píxeles hasta el rango 0-1 y las convertimos en flotantes, dividiéndolas por 255.

In [131]:
```
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```

Puede trazar una imagen usando la función `imshow ()` de Matplotlib, con un `'binario'` mapa de color:

In [132]:
```
plt.imshow(X_train[0], cmap="binary")
plt.axis('off')
plt.show()
```



Las etiquetas son los ID de clase (representados como uint8), de 0 a 9:

In [133]:
```
y_train
```

Out[133]:
```
array([4, 0, 7, ..., 3, 0, 5], dtype=uint8)
```

Aquí están los nombres de clase correspondientes:

In [134]:
```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

Entonces, la primera imagen del conjunto de entrenamiento es un abrigo:

In [135]:
```
class_names[y_train[0]]
```

Out[135]:
```
'Coat'
```

El conjunto de validación contiene 5000 imágenes y el conjunto de prueba contiene 10000 imágenes:

In [136]:
```
X_valid.shape
```

Out[136]:
```
(5000, 28, 28)
```

In [137]:
```
X_test.shape
```

Out[137]:
```
(10000, 28, 28)
```

Echemos un vistazo a una muestra de las imágenes en el conjunto de datos:

In [138]:
```
n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
for row in range(n_rows):
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(X_train[index], cmap="binary", interpolation="nearest")
        plt.axis('off')
        plt.title(class_names[y_train[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
save_fig('fashion_mnist_plot', tight_layout=False)
plt.show()
```

```
Saving figure fashion_mnist_plot
```

In [139]:
```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

In [140]:
```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)
```

In [141]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

In [142]:
```
model.layers
```

Out[142]:
```
[<tensorflow.python.keras.layers.core.Flatten at 0x299f2ae9448>,
 <tensorflow.python.keras.layers.core.Dense at 0x299f2ae95c8>,
 <tensorflow.python.keras.layers.core.Dense at 0x299f2ae9ac8>,
 <tensorflow.python.keras.layers.core.Dense at 0x299f2ad4188>]
```

In [143]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten (Flatten)            (None, 784)               0         
_________________________________________________________________
dense (Dense)                (None, 300)               235500    
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100     
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010      
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [144]:
```
keras.utils.plot_model(model, "my_fashion_mnist_model.png", show_shapes=True
```

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

In [145]:
```
hidden1 = model.layers[1]
hidden1.name
```

Out[145]:
```
'dense'
```

In [146]:
```
model.get_layer(hidden1.name) is hidden1
```

Out[146]:
```
True
```

In [147]:
```
weights, biases = hidden1.get_weights()
```

In [148]:
```python
weights
```

Out[148]:
```
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
          0.03859074, -0.06889391],
        [ 0.00476504, -0.03105379, -0.0586676 , ...,  0.00602964,
         -0.02763776, -0.04165364],
        [-0.06189284, -0.06901957,  0.07102345, ..., -0.04238207,
          0.07121518, -0.07331658],
        ...,
        [-0.03048757,  0.02155137, -0.05400612, ..., -0.00113463,
          0.00228987,  0.05581069],
        [ 0.07061854, -0.06960931,  0.07038955, ..., -0.00384101,
          0.00034875,  0.02878492],
        [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
          0.00272203, -0.06793761]], dtype=float32)
```

In [149]:
```python
weights.shape
```

Out[149]:
```
(784, 300)
```

In [150]:
```python
biases
```

Out[150]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

In [151]:
```python
biases.shape
```

Out[151]:
```
(300,)
```

In [152]:
```python
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Esto es equivalente a:

```python
model.compile(loss=keras.losses.sparse_categorical_crossentropy,
```

```
              optimizer=keras.optimizers.SGD(),
              metrics=[keras.metrics.sparse_categorical_accuracy])
```

In [153]:
```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/30
1719/1719 [==============================] - 6s 4ms/step - loss: 0.7237 - accur
acy: 0.7644 - val_loss: 0.5207 - val_accuracy: 0.8234
Epoch 2/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.4843 - accur
acy: 0.8318 - val_loss: 0.4345 - val_accuracy: 0.8538
Epoch 3/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.4393 - accur
acy: 0.8454 - val_loss: 0.5341 - val_accuracy: 0.7988
Epoch 4/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.4126 - accur
acy: 0.8565 - val_loss: 0.3915 - val_accuracy: 0.8644
Epoch 5/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3940 - accur
acy: 0.8618 - val_loss: 0.3748 - val_accuracy: 0.8690
Epoch 6/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3753 - accur
acy: 0.8677 - val_loss: 0.3707 - val_accuracy: 0.8728
Epoch 7/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3633 - accur
acy: 0.8715 - val_loss: 0.3623 - val_accuracy: 0.8720
Epoch 8/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3519 - accur
acy: 0.8750 - val_loss: 0.3848 - val_accuracy: 0.8624
Epoch 9/30
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3416 - accur
acy: 0.8792 - val_loss: 0.3588 - val_accuracy: 0.8704
Epoch 10/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3324 - accur
acy: 0.8819 - val_loss: 0.3427 - val_accuracy: 0.8780
Epoch 11/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3243 - accur
acy: 0.8835 - val_loss: 0.3433 - val_accuracy: 0.8786
Epoch 12/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3151 - accur
acy: 0.8866 - val_loss: 0.3310 - val_accuracy: 0.8820
Epoch 13/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3083 - accur
acy: 0.8885 - val_loss: 0.3262 - val_accuracy: 0.8888
Epoch 14/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.3024 - accur
acy: 0.8914 - val_loss: 0.3387 - val_accuracy: 0.8774
Epoch 15/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2950 - accur
acy: 0.8939 - val_loss: 0.3205 - val_accuracy: 0.8864
Epoch 16/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2892 - accur
acy: 0.8972 - val_loss: 0.3083 - val_accuracy: 0.8908
Epoch 17/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2841 - accur
acy: 0.8977 - val_loss: 0.3546 - val_accuracy: 0.8740
Epoch 18/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2780 - accur
acy: 0.9000 - val_loss: 0.3138 - val_accuracy: 0.8902
```

```
Epoch 19/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2729 - accur
acy: 0.9021 - val_loss: 0.3130 - val_accuracy: 0.8898
Epoch 20/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2678 - accur
acy: 0.9035 - val_loss: 0.3271 - val_accuracy: 0.8804
Epoch 21/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2626 - accur
acy: 0.9056 - val_loss: 0.3069 - val_accuracy: 0.8918
Epoch 22/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2578 - accur
acy: 0.9072 - val_loss: 0.2971 - val_accuracy: 0.8960
Epoch 23/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2538 - accur
acy: 0.9082 - val_loss: 0.2986 - val_accuracy: 0.8936
Epoch 24/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2485 - accur
acy: 0.9105 - val_loss: 0.3073 - val_accuracy: 0.8882
Epoch 25/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2447 - accur
acy: 0.9121 - val_loss: 0.2970 - val_accuracy: 0.8954
Epoch 26/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2409 - accur
acy: 0.9136 - val_loss: 0.3055 - val_accuracy: 0.8888
Epoch 27/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2366 - accur
acy: 0.9155 - val_loss: 0.3019 - val_accuracy: 0.8952
Epoch 28/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2331 - accur
acy: 0.9167 - val_loss: 0.2993 - val_accuracy: 0.8930
Epoch 29/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2287 - accur
acy: 0.9182 - val_loss: 0.3052 - val_accuracy: 0.8892
Epoch 30/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2255 - accur
acy: 0.9187 - val_loss: 0.3032 - val_accuracy: 0.8930
```

In [154]:

```
history.params
```

Out[154]:

```
{'verbose': 1, 'epochs': 30, 'steps': 1719}
```

In [155]:

```
print(history.epoch)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29]
```

In [156]:

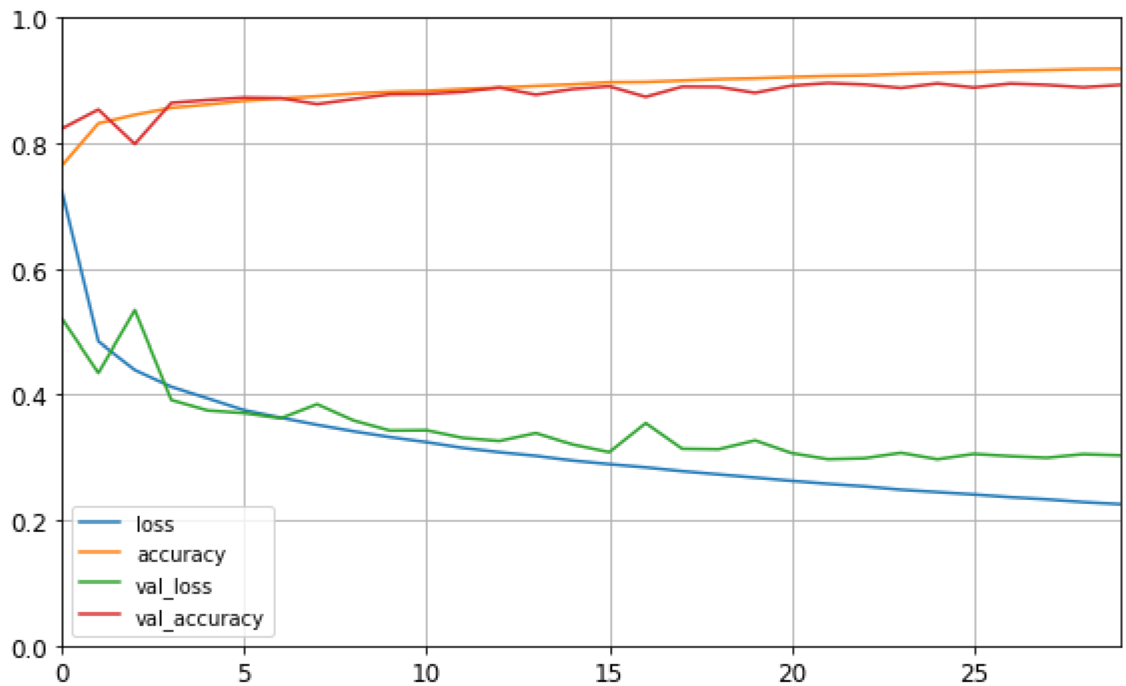
```
history.history.keys()
```

Out[156]:

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [157]:
```
import pandas as pd

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

```
Saving figure keras_learning_curves_plot
```



In [158]:
```
model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3377 - accurac
y: 0.8827
```

Out[158]:
```
[0.33773481845855713, 0.8827000260353088]
```

In [159]:

```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```

```
WARNING:tensorflow:5 out of the last 7 calls to <function Model.make_predict_function.<locals>.predict_function at 0x00000299F2547AF8> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental_relax_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args (https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args) and https://www.tensorflow.org/api_docs/python/tf/function (https://www.tensorflow.org/api_docs/python/tf/function) for  more details.
```

Out[159]:

```
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.02, 0.  , 0.96],
       [0.  , 0.  , 0.99, 0.  , 0.01, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

In [160]:

```
y_pred = model.predict_classes(X_new)
y_pred
```

Out[160]:

```
array([9, 2, 1], dtype=int64)
```

In [161]:

```
np.array(class_names)[y_pred]
```

Out[161]:

```
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

In [162]:

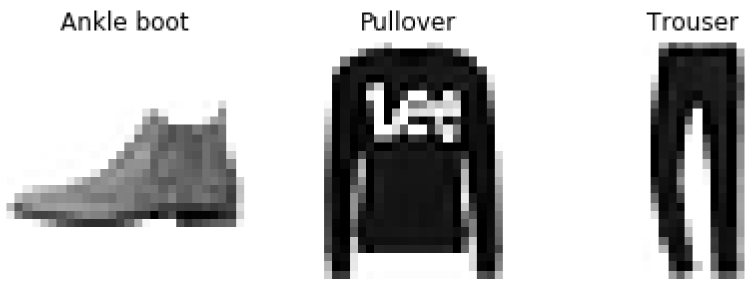
```
y_new = y_test[:3]
y_new
```

Out[162]:

```
array([9, 2, 1], dtype=uint8)
```

In [163]:
```
plt.figure(figsize=(7.2, 2.4))
for index, image in enumerate(X_new):
    plt.subplot(1, 3, index + 1)
    plt.imshow(image, cmap="binary", interpolation="nearest")
    plt.axis('off')
    plt.title(class_names[y_test[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
save_fig('fashion_mnist_images_plot', tight_layout=False)
plt.show()
```

```
Saving figure fashion_mnist_images_plot
```



Regressión MLP

Carguemos, dividamos y escalemos el conjunto de datos de viviendas de California:

In [164]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(housing.data,
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full, y_train_

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

In [165]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

In [166]:

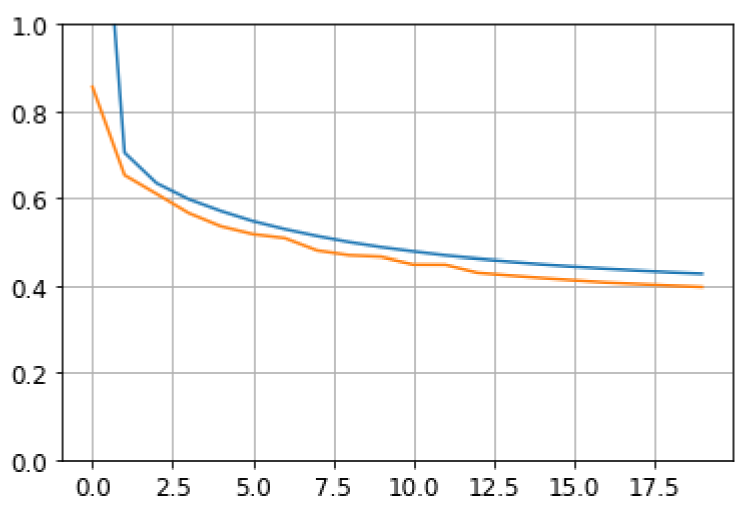
```python
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:])
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1
history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

```
Epoch 1/20
363/363 [==============================] - 1s 3ms/step - loss: 1.6419 - val_los
s: 0.8560
Epoch 2/20
363/363 [==============================] - 1s 2ms/step - loss: 0.7047 - val_los
s: 0.6531
Epoch 3/20
363/363 [==============================] - 1s 2ms/step - loss: 0.6345 - val_los
s: 0.6099
Epoch 4/20
363/363 [==============================] - 1s 2ms/step - loss: 0.5977 - val_los
s: 0.5658
Epoch 5/20
363/363 [==============================] - 1s 2ms/step - loss: 0.5706 - val_los
s: 0.5355
Epoch 6/20
363/363 [==============================] - 1s 2ms/step - loss: 0.5472 - val_los
s: 0.5173
Epoch 7/20
363/363 [==============================] - 1s 2ms/step - loss: 0.5288 - val_los
s: 0.5081
Epoch 8/20
363/363 [==============================] - 1s 2ms/step - loss: 0.5130 - val_los
s: 0.4799
Epoch 9/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4992 - val_los
s: 0.4690
Epoch 10/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4875 - val_los
s: 0.4656
Epoch 11/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4777 - val_los
s: 0.4482
Epoch 12/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4688 - val_los
s: 0.4479
Epoch 13/20
363/363 [==============================] - ETA: 0s - loss: 0.463 - 1s 2ms/step
- loss: 0.4615 - val_loss: 0.4296
Epoch 14/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4547 - val_los
s: 0.4233
Epoch 15/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4488 - val_los
s: 0.4176
Epoch 16/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4435 - val_los
```

```
s: 0.4123
Epoch 17/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4389 - val_los
s: 0.4071
Epoch 18/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4347 - val_los
s: 0.4037
Epoch 19/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4306 - val_los
s: 0.4000
Epoch 20/20
363/363 [==============================] - 1s 2ms/step - loss: 0.4273 - val_los
s: 0.3969
162/162 [==============================] - 0s 1ms/step - loss: 0.4212
WARNING:tensorflow:6 out of the last 9 calls to <function Model.make_predict_fu
nction.<locals>.predict_function at 0x00000299F2540AF8> triggered tf.function r
etracing. Tracing is expensive and the excessive number of tracings could be du
e to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with d
ifferent shapes, (3) passing Python objects instead of tensors. For (1), please
define your @tf.function outside of the loop. For (2), @tf.function has experim
ental_relax_shapes=True option that relaxes argument shapes that can avoid unne
cessary retracing. For (3), please refer to https://www.tensorflow.org/tutorial
s/customization/performance#python_or_tensor_args (https://www.tensorflow.org/t
utorials/customization/performance#python_or_tensor_args) and https://www.tenso
rflow.org/api_docs/python/tf/function (https://www.tensorflow.org/api_docs/pyth
on/tf/function) for  more details.
```

In [167]:

```
plt.plot(pd.DataFrame(history.history))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



In [168]:

```
y_pred
```

Out[168]:

```
array([[0.3885664],
       [1.6792021],
       [3.1022797]], dtype=float32)
```

In [169]:
```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nval/train: {:.2f}".format(logs["val_loss"] / logs["loss"]))
```

In [170]:
```
val_train_ratio_cb = PrintValTrainRatioCallback()
history = model.fit(X_train, y_train, epochs=1,
                    validation_data=(X_valid, y_valid),
                    callbacks=[val_train_ratio_cb])
```

```
  1/363 [..............................] - ETA: 0s - loss: 0.8053WARNING:tensor
flow:Callbacks method `on_train_batch_begin` is slow compared to the batch time
(batch time: 0.0000s vs `on_train_batch_begin` time: 0.0010s). Check your callb
acks.
WARNING:tensorflow:Callbacks method `on_train_batch_end` is slow compared to th
e batch time (batch time: 0.0000s vs `on_train_batch_end` time: 0.0014s). Check
your callbacks.
354/363 [============================>.] - ETA: 0s - loss: 0.4212WARNING:tensor
flow:Callbacks method `on_test_batch_begin` is slow compared to the batch time
(batch time: 0.0013s vs `on_test_batch_begin` time: 0.0029s). Check your callba
cks.

val/train: 0.93
363/363 [==============================] - 1s 2ms/step - loss: 0.4240 - val_los
s: 0.3932
```

In [171]:
```
model.evaluate(X_test, y_test)
```

```
162/162 [==============================] - 0s 2ms/step - loss: 0.4184
```

Out[171]: 0.4183996021747589