Chapter 10 – Introduction to Artificial Neural Networks with Keras

This notebook contains all the sample code and solutions to the exercises in chapter 10.

Setup

This project requires Python 3.7 or above:

In [1]:
```
import sys

assert sys.version_info >= (3, 7)
```

It also requires Scikit-Learn ≥ 1.0.1:

In [2]:
```
from packaging import version
import sklearn

assert version.parse(sklearn.__version__) >= version.parse("1.0.1")
```

And TensorFlow ≥ 2.8:

In [3]:
```
import tensorflow as tf

assert version.parse(tf.__version__) >= version.parse("2.8.0")
```

```
WARNING: All log messages before absl::InitializeLog() is called are writt
en to STDERR
I0000 00:00:1774940204.155672  439194 cudart_stub.cc:31] Could not find cu
da drivers on your machine, GPU will not be used.
I0000 00:00:1774940204.616145  439194 cpu_feature_guard.cc:227] This Tenso
rFlow binary is optimized to use available CPU instructions in performance
-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebui
ld TensorFlow with the appropriate compiler flags.
WARNING: All log messages before absl::InitializeLog() is called are writt
en to STDERR
I0000 00:00:1774940207.042758  439194 cudart_stub.cc:31] Could not find cu
da drivers on your machine, GPU will not be used.
```

As we did in previous chapters, let's define the default font sizes to make the figures prettier:

In [4]:
```
import matplotlib.pyplot as plt

plt.rc('font', size=14)
plt.rc('axes', labelsize=14, titlesize=14)
plt.rc('legend', fontsize=14)
plt.rc('xtick', labelsize=10)
plt.rc('ytick', labelsize=10)
```

And let's create the `images/ann` folder (if it doesn't already exist), and define the `save_fig()` function which is used through this notebook to save the figures in high-res for the book:

In [5]:
```
from pathlib import Path

IMAGES_PATH = Path() / "images" / "ann"
IMAGES_PATH.mkdir(parents=True, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=3
    path = IMAGES_PATH / f"{fig_id}.{fig_extension}"
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

From Biological to Artificial Neurons

The Perceptron

In [6]:
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0)  # Iris setosa

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new)  # predicts True and False for these 2 fl
```

In [7]:
```
y_pred
```

Out[7]:
```
array([ True, False])
```

The `Perceptron` is equivalent to a `SGDClassifier` with `loss="perceptron"`, no regularization, and a constant learning rate equal to 1:

In [8]:
```
# extra code – shows how to build and train a Perceptron

from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(loss="perceptron", penalty=None,
                        learning_rate="constant", eta0=1, random_state=42
sgd_clf.fit(X, y)
assert (sgd_clf.coef_ == per_clf.coef_).all()
assert (sgd_clf.intercept_ == per_clf.intercept_).all()
```

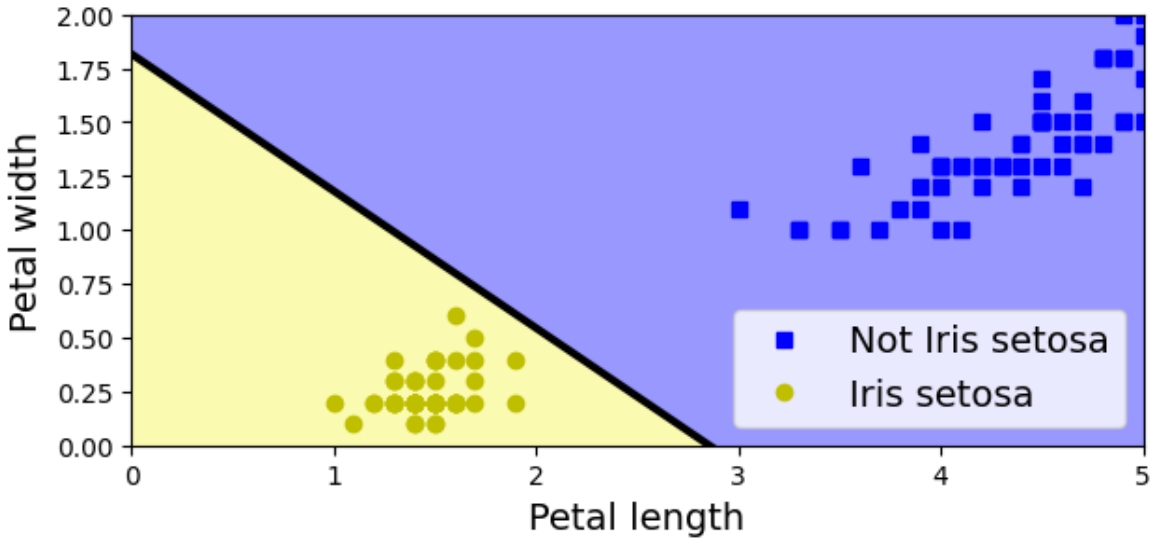
When the Perceptron finds a decision boundary that properly separates the classes, it stops learning. This means that the decision boundary is often quite close to one class:

In [9]:
```
# extra code – plots the decision boundary of a Perceptron on the iris da

import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

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)
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.figure(figsize=(7, 3))
plt.plot(X[y == 0, 0], X[y == 0, 1], "bs", label="Not 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-",
         linewidth=3)
plt.contourf(x0, x1, zz, cmap=custom_cmap)
plt.xlabel("Petal length")
plt.ylabel("Petal width")
plt.legend(loc="lower right")
plt.axis(axes)
plt.show()
```



Activation functions

In [10]:
```
# extra code – this cell generates and saves Figure 10–8

from scipy.special import expit as sigmoid

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

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

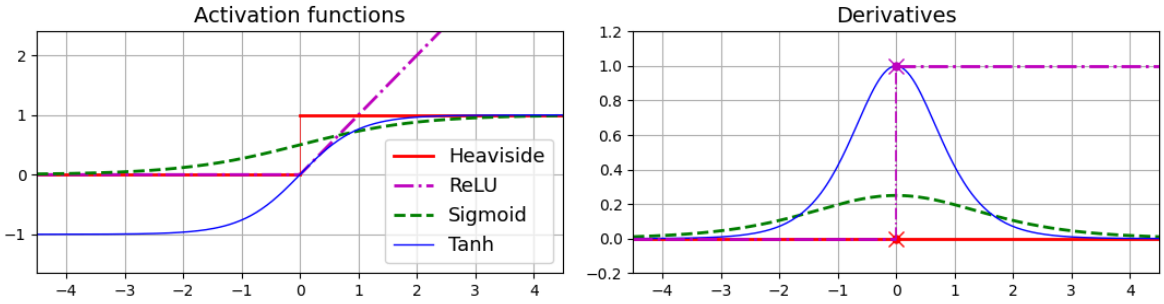
```
max_z = 4.5
z = np.linspace(-max_z, max_z, 200)

plt.figure(figsize=(11, 3.1))

plt.subplot(121)
plt.plot([-max_z, 0], [0, 0], "r-", linewidth=2, label="Heaviside")
plt.plot(z, relu(z), "m-.", linewidth=2, label="ReLU")
plt.plot([0, 0], [0, 1], "r-", linewidth=0.5)
plt.plot([0, max_z], [1, 1], "r-", linewidth=2)
plt.plot(z, sigmoid(z), "g--", linewidth=2, label="Sigmoid")
plt.plot(z, np.tanh(z), "b-", linewidth=1, label="Tanh")
plt.grid(True)
plt.title("Activation functions")
plt.axis([-max_z, max_z, -1.65, 2.4])
plt.gca().set_yticks([-1, 0, 1, 2])
plt.legend(loc="lower right", fontsize=13)

plt.subplot(122)
plt.plot(z, derivative(np.sign, z), "r-", linewidth=2, label="Heaviside")
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=1, label="Tanh")
plt.plot([-max_z, 0], [0, 0], "m-.", linewidth=2)
plt.plot([0, max_z], [1, 1], "m-.", linewidth=2)
plt.plot([0, 0], [0, 1], "m-.", linewidth=1.2)
plt.plot(0, 1, "mo", markersize=5)
plt.plot(0, 1, "mx", markersize=10)
plt.grid(True)
plt.title("Derivatives")
plt.axis([-max_z, max_z, -0.2, 1.2])

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



Implementing MLPs with Keras

Building an Image Classifier Using the Sequential API

Using Keras to load the dataset

Let's start by loading the fashion MNIST dataset. Keras has a number of functions to load popular datasets in `tf.keras.datasets`. The dataset is already split for you between

a training set (60,000 images) and a test set (10,000 images), but it can be useful to split the training set further to have a validation set. We'll use 55,000 images for training, and 5,000 for validation.

In [11]:
```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

The training set contains 60,000 grayscale images, each 28x28 pixels:

In [12]:
```
X_train.shape
```

Out[12]:
```
(55000, 28, 28)
```

Each pixel intensity is represented as a byte (0 to 255):

In [13]:
```
X_train.dtype
```

Out[13]:
```
dtype('uint8')
```

Let's scale the pixel intensities down to the 0-1 range and convert them to floats, by dividing by 255:

In [14]:
```
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

You can plot an image using Matplotlib's `imshow()` function, with a `'binary'` color map:

In [15]:
```
# extra code

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

The labels are the class IDs (represented as uint8), from 0 to 9:

In [16]:
```
y_train
```

Out[16]:
```
array([9, 0, 0, ..., 9, 0, 2], shape=(55000,), dtype=uint8)
```

Here are the corresponding class names:

In [17]:
```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

So the first image in the training set is an ankle boot:

In [18]:
```
class_names[y_train[0]]
```

Out[18]:
```
'Ankle boot'
```

Let's take a look at a sample of the images in the dataset:

In [19]:
```
# extra code – this cell generates and saves Figure 10–10

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]])
plt.subplots_adjust(wspace=0.2, hspace=0.5)
```

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

Creating the model using the Sequential API

In [20]:
```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(input_shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

```
/home/madhavan/python-venv/lib/python3.13/site-packages/keras/src/layers/c
ore/input_layer.py:27: UserWarning: Argument `input_shape` is deprecated.
Use `shape` instead.
  warnings.warn(
E0000 00:00:1774940212.237376  439194 cuda_platform.cc:52] failed call to
cuInit: INTERNAL: CUDA error: Failed call to cuInit: UNKNOWN ERROR (303)
```

In [21]:
```
# extra code – clear the session to reset the name counters
tf.keras.backend.clear_session()
tf.random.set_seed(42)

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

```
/home/madhavan/python-venv/lib/python3.13/site-packages/keras/src/layers/r
eshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_d
im` argument to a layer. When using Sequential models, prefer using an `In
put(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

In [22]:
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	
flatten (Flatten)	(None, 784)	
dense (Dense)	(None, 300)	2
dense_1 (Dense)	(None, 100)	
dense_2 (Dense)	(None, 10)	

Total params: 266,610 (1.02 MB)

Trainable params: 266,610 (1.02 MB)

Non-trainable params: 0 (0.00 B)

In [23]:
```
# extra code – another way to display the model's architecture
tf.keras.utils.plot_model(model, "my_fashion_mnist_model.png", show_shape
```

```
You must install pydot (`pip install pydot`) for `plot_model` to work.
```

In [24]:
```
model.layers
```

Out[24]:
```
[<Flatten name=flatten, built=True>,
 <Dense name=dense, built=True>,
 <Dense name=dense_1, built=True>,
 <Dense name=dense_2, built=True>]
```

In [25]:
```
hidden1 = model.layers[1]
hidden1.name
```

Out[25]:
```
'dense'
```

In [26]:
```
model.get_layer('dense') is hidden1
```

Out[26]:
```
True
```

In [27]:
```
weights, biases = hidden1.get_weights()
weights
```

Out[27]:
```
array([[-0.01503324, -0.01729074, -0.0209767 , ...,  0.01256161,
        -0.01029796, -0.01812954],
       [ 0.04807831, -0.05462622, -0.06968277, ..., -0.07118641,
        -0.03634901, -0.00619955],
       [-0.02758568,  0.0021318 ,  0.01136191, ..., -0.06439991,
        -0.06434148,  0.01817922],
       ...,
       [ 0.02835793, -0.0673513 ,  0.00254125, ..., -0.02448661,
        -0.04257814,  0.03686985],
       [ 0.07074963, -0.07194687, -0.06052938, ...,  0.02105471,
         0.05061468,  0.0740158 ],
       [ 0.05856735, -0.00792463,  0.03314552, ...,  0.04626872,
         0.01818854, -0.0170098 ]], shape=(784, 300), dtype=float32)
```

In [28]:
```
weights.shape
```

Out[28]:
```
(784, 300)
```

In [29]:
```
biases
```

Out[29]:
```
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 [30]:
```
biases.shape
```

Out[30]:
```
(300,)
```

Compiling the model

In [31]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

This is equivalent to:

In [32]:
```
# extra code – this cell is equivalent to the previous cell
model.compile(loss=tf.keras.losses.sparse_categorical_crossentropy,
              optimizer=tf.keras.optimizers.SGD(),
              metrics=[tf.keras.metrics.sparse_categorical_accuracy])
```

In [33]:
```
# extra code – shows how to convert class ids to one-hot vectors
tf.keras.utils.to_categorical([0, 5, 1, 0], num_classes=10)
```

Out[33]:
```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

Note: it's important to set `num_classes` when the number of classes is greater than the maximum class id in the sample.

In [34]:
```
# extra code – shows how to convert one-hot vectors to class ids
np.argmax(
    [[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
     [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
     [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
     [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.]],
    axis=1
)
```

Out[34]:
```
array([0, 5, 1, 0])
```

Training and evaluating the model

In [35]:
```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/30
```

```
W0000 00:00:1774940213.361692  439194 cpu_allocator_impl.cc:82] Allocation
of 172480000 exceeds 10% of free system memory.
```

```
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.7012 - sparse_categor
ical_accuracy: 0.7671 - val_loss: 0.5043 - val_sparse_categorical_accurac
y: 0.8250
Epoch 2/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.4883 - sparse_categor
ical_accuracy: 0.8299 - val_loss: 0.4526 - val_sparse_categorical_accurac
y: 0.8362
Epoch 3/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.4440 - sparse_categor
ical_accuracy: 0.8452 - val_loss: 0.4284 - val_sparse_categorical_accurac
y: 0.8478
Epoch 4/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.4167 - sparse_categor
ical_accuracy: 0.8544 - val_loss: 0.4121 - val_sparse_categorical_accurac
y: 0.8536
Epoch 5/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.3969 - sparse_categor
ical_accuracy: 0.8608 - val_loss: 0.4021 - val_sparse_categorical_accurac
y: 0.8546
Epoch 6/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.3811 - sparse_categor
ical_accuracy: 0.8661 - val_loss: 0.3945 - val_sparse_categorical_accurac
y: 0.8574
Epoch 7/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.3679 - sparse_categor
ical_accuracy: 0.8705 - val_loss: 0.3860 - val_sparse_categorical_accurac
y: 0.8582
Epoch 8/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.3564 - sparse_categor
ical_accuracy: 0.8745 - val_loss: 0.3800 - val_sparse_categorical_accurac
y: 0.8618
Epoch 9/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.3460 - sparse_categor
ical_accuracy: 0.8779 - val_loss: 0.3754 - val_sparse_categorical_accurac
y: 0.8630
Epoch 10/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.3366 - sparse_categor
ical_accuracy: 0.8801 - val_loss: 0.3711 - val_sparse_categorical_accurac
y: 0.8642
Epoch 11/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.3279 - sparse_categor
ical_accuracy: 0.8829 - val_loss: 0.3664 - val_sparse_categorical_accurac
y: 0.8658
Epoch 12/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.3199 - sparse_categor
ical_accuracy: 0.8857 - val_loss: 0.3630 - val_sparse_categorical_accurac
y: 0.8672
Epoch 13/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.3123 - sparse_categor
ical_accuracy: 0.8878 - val_loss: 0.3610 - val_sparse_categorical_accurac
y: 0.8680
Epoch 14/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.3054 - sparse_categor
ical_accuracy: 0.8902 - val_loss: 0.3575 - val_sparse_categorical_accurac
y: 0.8692
Epoch 15/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2988 - sparse_categor
ical_accuracy: 0.8926 - val_loss: 0.3541 - val_sparse_categorical_accurac
y: 0.8698
Epoch 16/30
```

```
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2925 - sparse_categor
ical_accuracy: 0.8953 - val_loss: 0.3528 - val_sparse_categorical_accurac
y: 0.8706
Epoch 17/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2866 - sparse_categor
ical_accuracy: 0.8976 - val_loss: 0.3517 - val_sparse_categorical_accurac
y: 0.8706
Epoch 18/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.2809 - sparse_categor
ical_accuracy: 0.8995 - val_loss: 0.3506 - val_sparse_categorical_accurac
y: 0.8704
Epoch 19/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.2755 - sparse_categor
ical_accuracy: 0.9012 - val_loss: 0.3500 - val_sparse_categorical_accurac
y: 0.8726
Epoch 20/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2703 - sparse_categor
ical_accuracy: 0.9034 - val_loss: 0.3500 - val_sparse_categorical_accurac
y: 0.8734
Epoch 21/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.2653 - sparse_categor
ical_accuracy: 0.9050 - val_loss: 0.3484 - val_sparse_categorical_accurac
y: 0.8732
Epoch 22/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2604 - sparse_categor
ical_accuracy: 0.9071 - val_loss: 0.3483 - val_sparse_categorical_accurac
y: 0.8732
Epoch 23/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2558 - sparse_categor
ical_accuracy: 0.9087 - val_loss: 0.3482 - val_sparse_categorical_accurac
y: 0.8742
Epoch 24/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2513 - sparse_categor
ical_accuracy: 0.9104 - val_loss: 0.3483 - val_sparse_categorical_accurac
y: 0.8740
Epoch 25/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2470 - sparse_categor
ical_accuracy: 0.9121 - val_loss: 0.3479 - val_sparse_categorical_accurac
y: 0.8750
Epoch 26/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2427 - sparse_categor
ical_accuracy: 0.9137 - val_loss: 0.3470 - val_sparse_categorical_accurac
y: 0.8746
Epoch 27/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2386 - sparse_categor
ical_accuracy: 0.9153 - val_loss: 0.3469 - val_sparse_categorical_accurac
y: 0.8748
Epoch 28/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2346 - sparse_categor
ical_accuracy: 0.9170 - val_loss: 0.3470 - val_sparse_categorical_accurac
y: 0.8758
Epoch 29/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2306 - sparse_categor
ical_accuracy: 0.9189 - val_loss: 0.3470 - val_sparse_categorical_accurac
y: 0.8760
Epoch 30/30
1719/1719 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.2266 - sparse_categor
ical_accuracy: 0.9200 - val_loss: 0.3457 - val_sparse_categorical_accurac
y: 0.8760
```

In [36]:
```
history.params
```

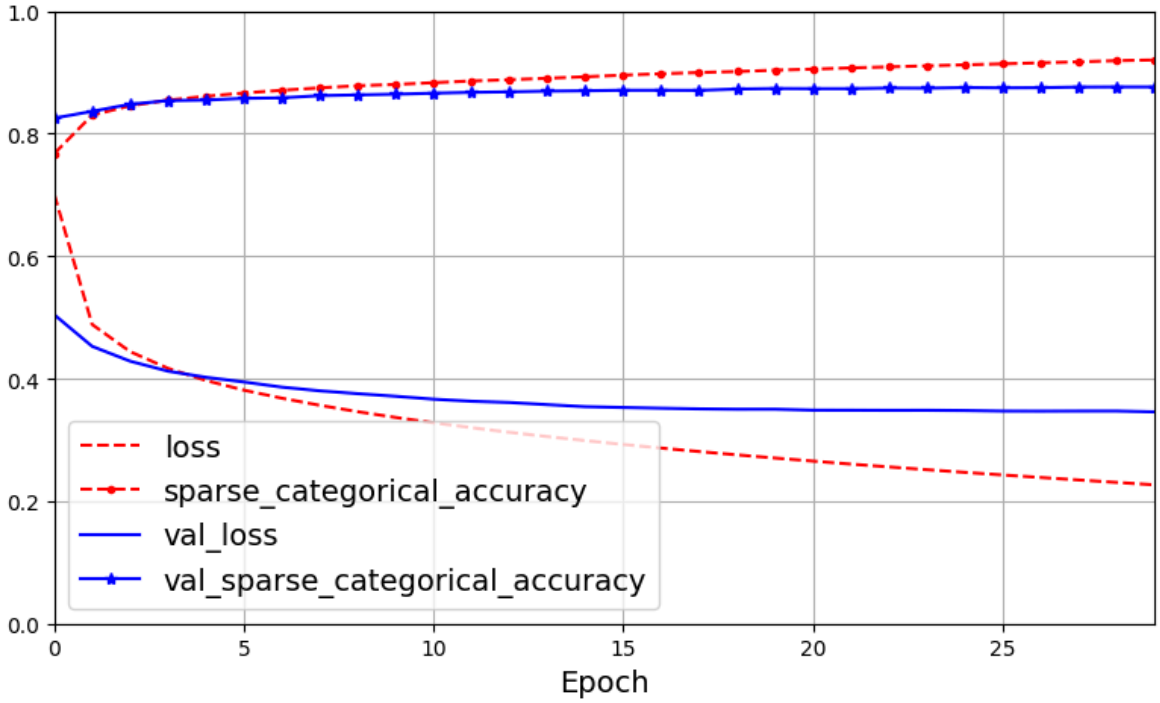
Out[36]:
```
{'verbose': 'auto', 'epochs': 30, 'steps': 1719}
```

In [37]:
```
print(history.epoch)
```

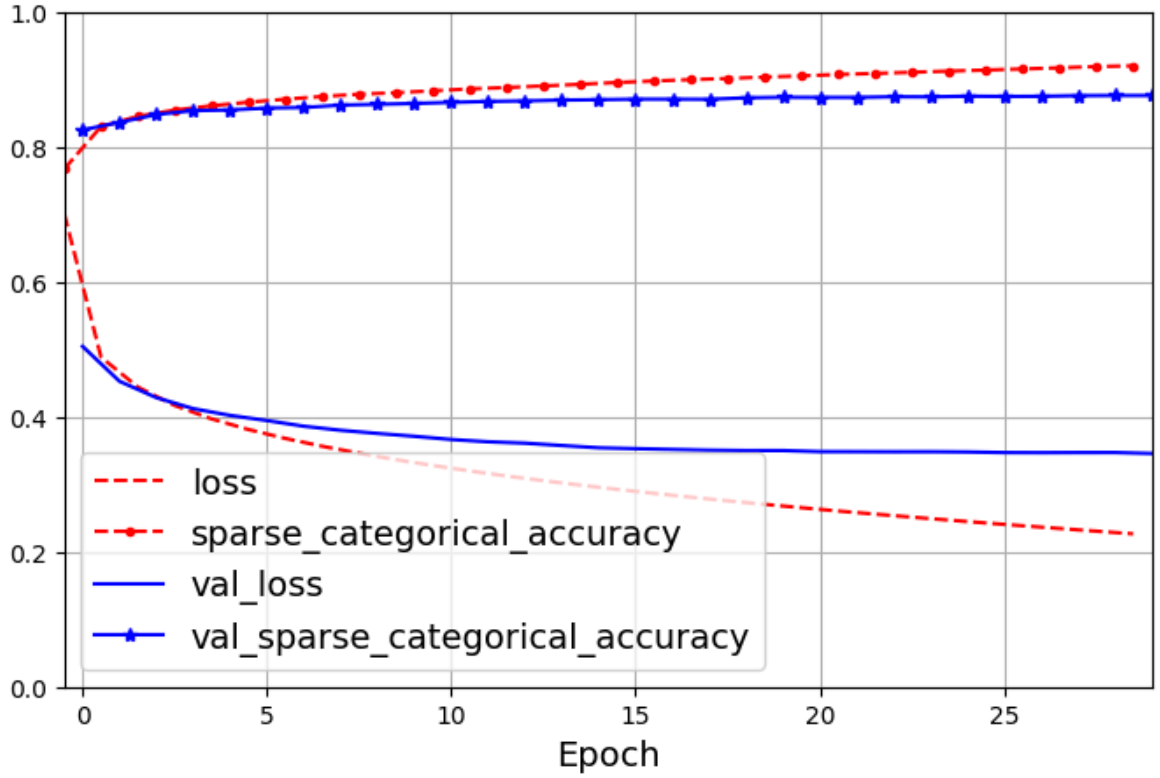
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
```

In [38]:
```
import matplotlib.pyplot as plt
import pandas as pd

pd.DataFrame(history.history).plot(
    figsize=(8, 5), xlim=[0, 29], ylim=[0, 1], grid=True, xlabel="Epoch",
    style=["r--", "r--.", "b-", "b-*"])
plt.legend(loc="lower left")  # extra code
save_fig("keras_learning_curves_plot")  # extra code
plt.show()
```



In [39]:
```
# extra code – shows how to shift the training curve by -1/2 epoch
plt.figure(figsize=(8, 5))
for key, style in zip(history.history, ["r--", "r--.", "b-", "b-*"]):
    epochs = np.array(history.epoch) + (0 if key.startswith("val_") else
    plt.plot(epochs, history.history[key], style, label=key)
plt.xlabel("Epoch")
plt.axis([-0.5, 29, 0., 1])
plt.legend(loc="lower left")
plt.grid()
plt.show()
```



In [40]:
```
model.evaluate(X_test, y_test)
```

```
109/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.3542 - sparse_categoric
al_accuracy: 0.8762
W0000 00:00:1774940320.596303  439194 cpu_allocator_impl.cc:82] Allocation
of 31360000 exceeds 10% of free system memory.
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.3672 - sparse_categoric
al_accuracy: 0.8712
```

Out[40]:
```
[0.36717379093170166, 0.8712000250816345]
```

Using the model to make predictions

In [41]:
```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
```

Out[41]:
```
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.3 , 0.  , 0.01, 0.  , 0.69],
       [0.  , 0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

In [42]:
```
y_pred = y_proba.argmax(axis=-1)
y_pred
```

Out[42]:
```
array([9, 2, 1])
```

In [43]:
```
np.array(class_names)[y_pred]
```

Out[43]:
```
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

In [44]:
```
y_new = y_test[:3]
y_new
```

Out[44]:
```
array([9, 2, 1], dtype=uint8)
```

In [45]:
```
# extra code – this cell generates and saves Figure 10–12
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]])
plt.subplots_adjust(wspace=0.2, hspace=0.5)
save_fig('fashion_mnist_images_plot', tight_layout=False)
plt.show()
```