2.3. Keras and TensorFlow

2.3.1. Keras and TensorFlow libraries

TensorFlow ¶

- **Characteristics**:
  - TensorFlow is a free and open-source software library for machine learning and artificial intelligence that has a particular focus on deep neural networks.
  - TensorFlow was developed by the Google Brain team for internal Google use and publicy released in 2015 under the Apache License 2.0.
  - TensorFlow can be used in a wide variety of programming languages, including Python, JavaScript, C++, and Java.
  - URL: https://www.tensorflow.org/ (https://www.tensorflow.org/)
- **Installation**
  - We can install TensorFlow with Python's pip package manager:

```
pip install tensorflow
```

Keras

- **Characteristics**
  - Keras is a deep learning API written in Python, running on top of TensorFlow.
  - Keras was developed by François Chollet in Google AI in 2015 as a high-level library for different deep learning backends such as Theano (University of Montreal), CNTK (Microsoft), TensorFlow (Google) or mxnet (Apache).
  - Since version 2.4, only TensorFlow is supported.
  - It is planned that version 3.0 will support TensorFlow, PyTorch and
  - URL: https://keras.io (https://keras.io)
- **Installation**
  - To use Keras, will need to have the TensorFlow package installed.
  - Once TensorFlow is installed, just import Keras via:

```
from tensorflow import keras
```

- **Properties**:
  - *Simple* -- but not simplistic. Keras reduces developer cognitive load to free you to focus on the parts of the problem that really matter.
  - *Flexible* -- Keras adopts the principle of progressive disclosure of complexity: simple workflows should be quick and easy, while arbitrarily advanced workflows should be possible via a clear path that builds upon what you've already learned.
  - *Powerful* -- Keras provides industry-strength performance and scalability: it is used by organizations and companies including NASA, YouTube, or Waymo.

Keras models

- URL: https://keras.io/api/models/ (https://keras.io/api/models/)
- There are three ways to create Keras models:

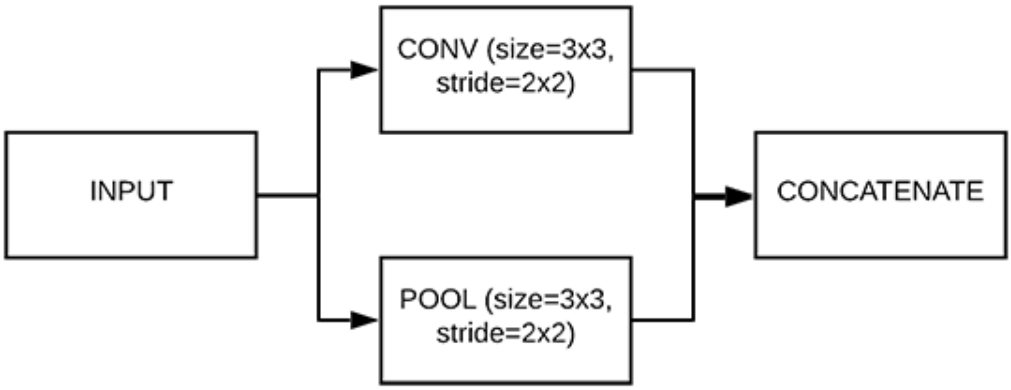
(1) Sequential model

- It is the simplest model, defined as a linear stack of layers, mapping input data to output data.
- It is the most limited model, since it only follows a linear structure of single-input and single-output layers.
- Our first deep learning network will be implemented using the sequential model.

(2) Functional API

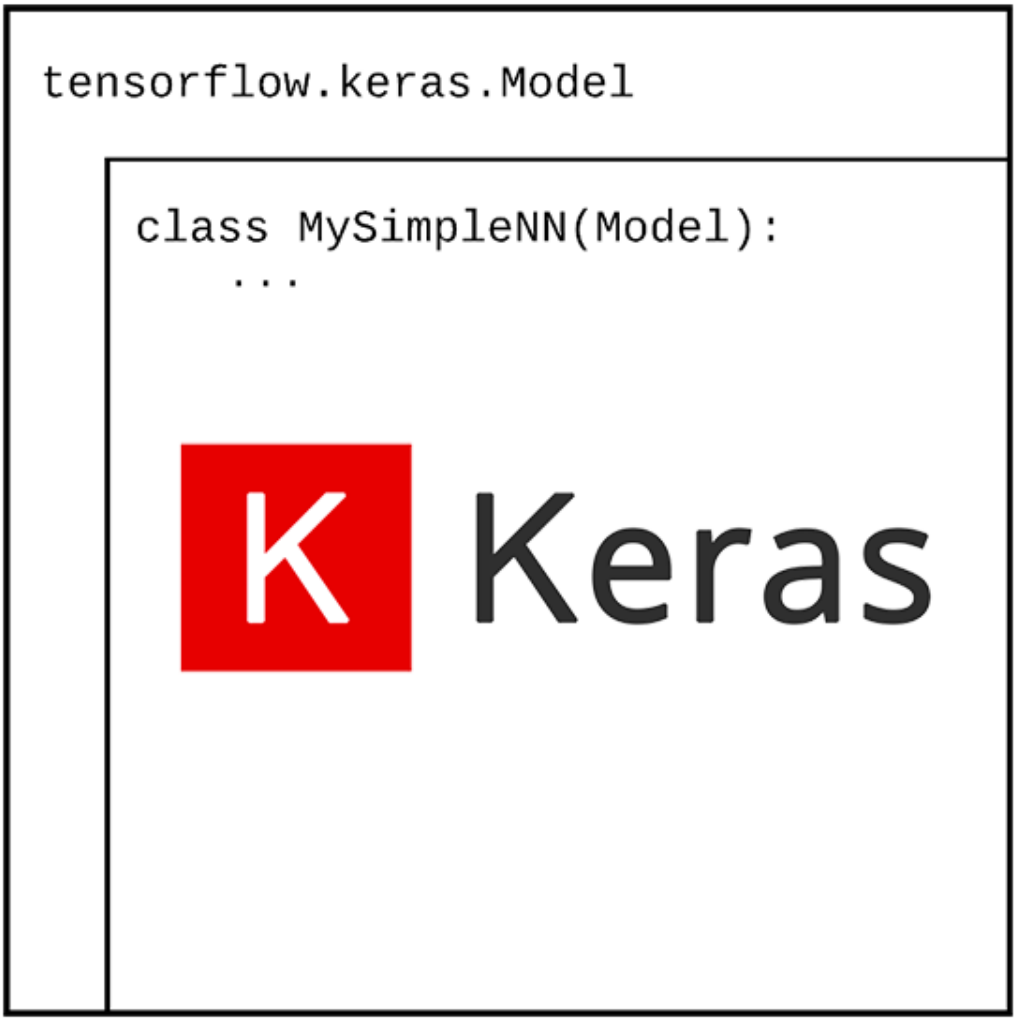
- It is an easy-to-use, fully-featured API that supports arbitrary model architectures.
- It defines more complex models and gives more freedom on how to structure its elements.
- We can, for example, create models with multiple outputs and multiple inputs, define branches in the architecture, design acyclic directed graphs, create shared layers, etc.
- Any sequential model can be easily recreated using a functional model.

- Most of the examples in this course will be created with the functional API.



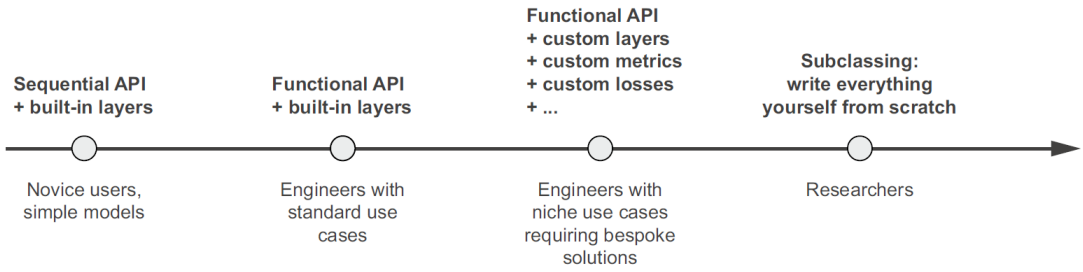
(3) Model subclassing

- Keras uses a `Model` class for representing the different models.
- The subclassing model consists of inheriting from that class and redefining the methods from scratch.
- Is used in complex, out-of-the-box research use cases.



Progressive disclousure

- From simple workflows to more complex ones targeting different users.



Keras layers

URL: [https://keras.io/api/layers/ (https://keras.io/api/layers/)](https://keras.io/api/layers/)

- Layers are the basic building blocks of neural networks in Keras.
- A layer consists of a tensor-in tensor-out computation function (the layer's call method) and some state, held in TensorFlow variables (the layer's weights).
- A Layer instance can be called, much like a function, but unlike a function, layers maintain a state, updated when the layer receives data during training, and stored in `layer.weights`
- Everything in Keras is either a Layer or something that closely interacts with a Layer.
- Some different layers types is Keras:
  - **Core layers**: Main layers for building networks ([https://keras.io/api/layers/core_layers/ (https://keras.io/api/layers/core_layers/)](https://keras.io/api/layers/core_layers/))
    - Input layer: Used to instantiate a Keras tensor.
    - Dense layer: A regular densely-connected NN layer.
    - Activation layer: Applies an activation function to an output.
  - **Layers for managing input**
    - Preprocessing layers: allows developers to build Keras-native input processing pipelines.
    - Normalizaton layers: Layers that normalize its inputs.
    - Reshaping layers: Layers that reshape inputs into a given shape.
  - **Layers for convolutional networks**: CNNs are used mainly to work with images
    - Convolutional layers
    - Pooling layers
  - **Layers for recurrent networks**: RNNs are used mainly to work with sequential data
    - LSTM layers
    - GRU layers
    - Bidirectional layers
  - **Attention layers**: Attention mechanisms are the basis for transformer architectures (the basis of large language models (LLMs) like ChatGPT)

2.3.2. MNIST: image classification

MNIST dataset

- The MNIST (Modified National Institute of Standards and Technology) is a database of handwritten digits.
- It is commonly used to train and test image processing systems in the field of machine learning. One could say that it is the *Hello World* of this field.
- It is composed of 60,000 training images and 10,000 test images.

- Each image has dimensions of 28x28 pixels in grayscale and represents one of the ten possible digits (from 0 to 9).

- The MNIST data are preloaded into Keras in the form of four NumPy arrays: two pair for training and two pair for testing
- Each pair (training or test) has an array to store the image data and an array with the labels corresponding to each image.
- We import the `mnist` package that stores the corresponding data.

In [1]:
```
from tensorflow.keras.datasets import mnist
```

- We load the training data and the test data as a NumPy `ndarray` object.
- NumPy (http://www.numpy.org/ (http://www.numpy.org/)) is the fundamental package for scientific computing with Python.
- It contains, among other things:
  - An object to represent n-dimensional arrays (`ndarray`) efficiently.
  - Fast array-oriented arithmetic operations with flexible broadcasting capabilities.
  - Mathematical functions that allow you to efficiently perform operations on arrays without having to write loops.

In [2]:
```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data
```

In [3]:
```
type(train_images)
```

Out[3]:
```
numpy.ndarray
```

- Let's look at the training data.
- `shape` is an attribute of a `ndarray`.
- It contains a tuple of integers indicating the size of each of the array dimensions.
- The length of the tuple will be the number of array dimensions.

In [4]:
```
train_images.shape
```

Out[4]:
```
(60000, 28, 28)
```

In [5]:
```
train_labels.shape
```

Out[5]:
```
(60000,)
```

In [6]:
```
train_labels
```

Out[6]:
```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

- And here's the test data:

In [7]:
```
test_images.shape
```

Out[7]:
```
(10000, 28, 28)
```

In [8]:
```
test_labels.shape
```

Out[8]:
```
(10000,)
```

In [9]:
```
test_labels
```

Out[9]:
```
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

MNIST: Preparing the data

- We need to prepare the data to what the network expects from them.
- The images are an array of dimensions `(60000, 28, 28)` of type `uint8` with values of `[0, 255]`.
- First, they must be transformed into an array of dimensions `(60000, 28 * 28)`
  - The image is flatten to a simple array of `28 * 28 = 784` values.
  - We use the NumPy `reshape` function that changes the shape of an array without changing its data (https://numpy.org/doc/stable/reference/generated/numpy.reshape.html (https://numpy.org/doc/stable/reference/generated/numpy.reshape.html))

In [10]:
```
train_images = train_images.reshape((60000, 28 * 28))
test_images = test_images.reshape((10000, 28 * 28))
```

- Second, they must be tranformed to a float type `float32` with values between 0 and 1.
  - Each value ranges from 0 to 255 (levels of gray) and should be normalized to values between 0 and 1 (dividing by 255).
  - We use the NumPy `astype` function that copies the array and cast its values to a specified type.
  - When we divide an `ndarray` by a number we divide each element of the array by that number.

In [11]:
```
train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255
```

- We also need to categorically encode the labels using the *one-hot* encoding.
- In this encoding we convert an integer into an array of integers in which all values are set to zero except the value that corresponded to the original integer which is set to one.
- We use the `to_categorical` function that converts integers to a binary class matrix (https://keras.io/api/utils/python_utils/ (https://keras.io/api/utils/python_utils/))

In [12]:
```
train_labels[2]
```

Out[12]:
```
4
```

In [13]:
```
from tensorflow.keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

In [14]:
```
train_labels[2]
```

Out[14]:
```
array([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.], dtype=float32)
```

MNIST: Building the Model and Layers

Sequential model

- We are going to use the Keras sequential model.
- We use the `Sequential` class (https://keras.io/api/models/sequential/ (https://keras.io/api/models/sequential/)) to create a sequential model.

In [15]:
```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential()
```

Input layer

- The first layer to create is an input layer that receives the input (https://keras.io/api/layers/core_layers/input/ (https://keras.io/api/layers/core_layers/input/)).
- The most important parameter in the constructor of `Input` is `shape` , a tuple of integer that indicates the shape of the input.
- For instance, `shape=(784,)` indicates that the expected input will be batches of 784-dimensional vectors (our 28x28 images).
- Elements of this tuple can be `None` representing dimensions where the shape is not known.
- We add this layer to the model using the `add` method.

In [16]:
```
model.add(layers.Input(shape=(784, )))
```

Dense layers

- Dense layers are layers formed by artificial neurons each one is fully connected with the previous and subsequent layers.
- In Keras `Dense` (https://keras.io/api/layers/core_layers/dense/ (https://keras.io/api/layers/core_layers/dense/)) represents a densely-connected neural network layer that implements the operation `output = activation(dot(input, kernel) + bias)` .
- The typical parameters for the `Dense` constructor are:
  - units: the number of neurons in the layer.
  - activation: the activation function used in each neuron.
  - input_shape: when passed Keras will create an input layer to insert before the current layer. This can be treated equivalent ot explicitly defining an `Input` Layer.
- In this case we are going to create a dense layer that receives the 784 inputs and produces 512 outputs activated by the ReLU function.

In [17]:
```
model.add(layers.Dense(512, activation="relu"))
```

Output layer

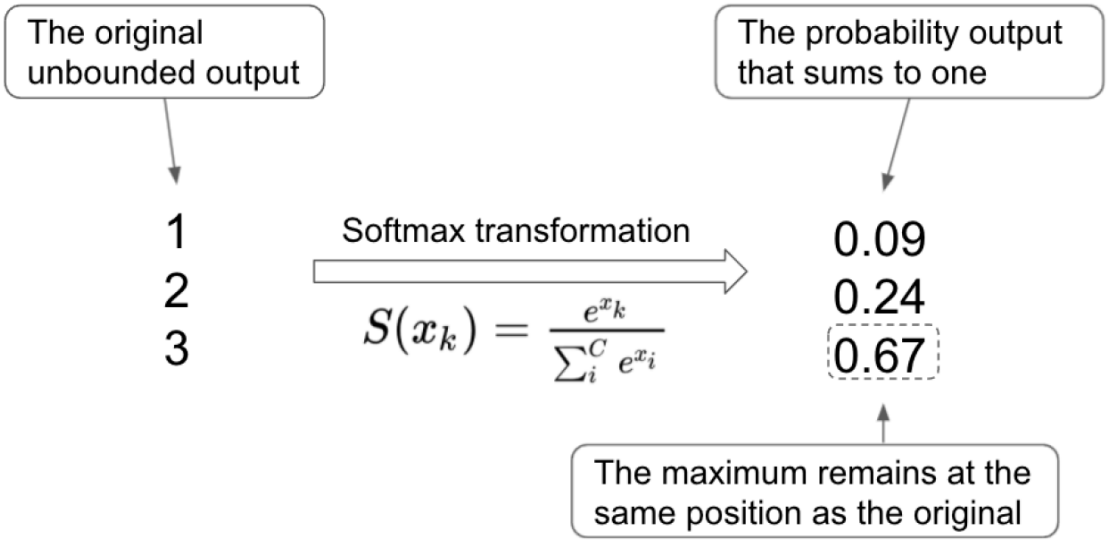
- The output layer is also a dense layer but in this case with 10 neurons, each of which will represent a digit.
- The output value of each neuron of the output layer will indicate the probability that the current digit corresponds to the digit represented by that neuron.
- The total probability must add up to one.
- To obtain these values we use the activation function *softmax*.

In [18]:
```
model.add(layers.Dense(10, activation="softmax"))
```

Softmax activation function

- The softmax function or normalized exponential function converts a vector of K real numbers into a probability distribution of K possible outcomes.
- The softmax function is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.
- The standard softmax function $\sigma : \mathbb{R}^K \to (0, 1)^K$ is defined when $K \geq 1$ by the formula:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \dots, K \text{ and } \mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K.$$



Model summary

- We can observe a summary of our model using the `summary()` function.

In [19]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               401920

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 407050 (1.55 MB)
Trainable params: 407050 (1.55 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

MNIST: Learning process

Defining the learning parameters

- Before training a model, it is necessary to set up the learning process, which is done through the `compile` method.
- We need to pick three more things as part of the compilation step:
  - A **loss function**: How the model will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direction.
  - An **optimizer**: The mechanism through which the model will update itself based on the training data it sees, so as to improve its performance.
  - **Metrics** to monitor during training and testing. Here, we'll only care about accuracy (the fraction of the images that were correctly classified).

In [20]:
```
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
```

Training

- We're now ready to train the model, which in Keras is done via a call to the model's `fit()` method—we fit the model to its training data.
- The main parameters of the `fit()` method are (https://keras.io/api/models/model_training_apis/ (https://keras.io/api/models/model_training_apis/)):
  - The **input data**: the train images.
  - The **target data**: the train labels.
  - The **epochs**: Number of iterations over the entire input and target data.
    - Using the input data several times we increase the accuracy of our model.
    - But we have to take care of not causing overfitting.
  - The **batch_size**: Number of samples per gradient update.
    - With a batch size of 1 we upgrade the network weights after each sample has passed throught the network, this is slow and consume a lot of resources.
    - We can group the input data in batches and only update the weights after entering in the network all the samples of the batch.
    - The learning consume less resources, but since we are propagating backwards the average error of all the batch samples, the quality of the model may degrade and may ultimately be unable to generalize well on data it hasn't seen before.
- Two quantities are displayed during training: the loss of the model over the training data, and the accuracy of the model over the training data.

In [21]:
```
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
Epoch 1/5
469/469 [==============================] - 2s 4ms/step - loss: 0.2655 - a
ccuracy: 0.9237
Epoch 2/5
469/469 [==============================] - 2s 4ms/step - loss: 0.1060 - a
ccuracy: 0.9687
Epoch 3/5
469/469 [==============================] - 2s 4ms/step - loss: 0.0700 - a
ccuracy: 0.9788
Epoch 4/5
469/469 [==============================] - 2s 4ms/step - loss: 0.0512 - a
ccuracy: 0.9844
Epoch 5/5
469/469 [==============================] - 2s 4ms/step - loss: 0.0381 - a
ccuracy: 0.9886
```

Out[21]:
```
<keras.src.callbacks.History at 0x1df97bffb10>
```