# 1.  What's TensorFlow?

- TensorFlow is a Python-based, free, open source machine learning platform, developed primarily by Google.
  - ➢ Much like NumPy, the primary purpose of TensorFlow is to enable engineers and researchers to manipulate mathematical expressions over numerical tensors. But TensorFlow goes far beyond the scope of NumPy in the following ways
    - ✧ It can automatically compute the gradient of any differentiable expression, making it highly suitable for machine learning
    - ✧ It can run not only on CPUs, but also on GPUs and TPUs, highly parallel hardware accelerators
    - ✧ Computation defined in TensorFlow can be easily distributed across many machines
    - ✧ TensorFlow programs can be exported to other runtimes, such as C++, JavaScript (for browser-based applications), or TensorFlow Lite (for applications running on mobile devices or embedded devices), etc. This makes TensorFlow applications easy to deploy in practical settings.

- It's important to keep in mind that TensorFlow is much more than a single library. It's a platform, home to a vast ecosystem of components, some developed by Google and some developed by third parties.
  - ➢ For instance, there's TF-Agents for reinforcement-learning research, TFX for industry-strength machine learning workflow management, TensorFlow Serving for production deployment, and there's the TensorFlow Hub repository of pretrained models.
  - ➢ Together, these components cover a very wide range of use cases, from cutting-edge research to large-scale production applications.
- TensorFlow scales fairly well
  - ➢ For instance, scientists from Oak Ridge National Lab have used it to train a 1.1 exaFLOPS extreme weather forecasting model on 27,000 GPUs of the IBM Summit supercomputer. Likewise, Google has used TensorFlow to develop very compute-intensive deep learning applications, such as the chess-playing and Go-playing agent AlphaZero.
  - ➢ For your own models, if you have the budget, you can realistically hope to scale to around 10 petaFLOPS on a small TPU pod or a large cluster of GPUs rented on Google Cloud or AWS.
    - ✧ That would still be around 1% of the peak compute power of the top supercomputer in 2019!

# 2.  What's Keras?

- Keras is a deep learning API (application programming interface) for Python, built on top of TensorFlow, that provides a convenient way to define and train any kind of deep learning model.
  - ➢ Keras was initially developed for research, with the aim of enabling fast deep learning experimentation

➢ Through TensorFlow, Keras can run on top of different types of hardware – GPU, TPU, or plain CPU – and can be seamlessly scaled to thousands of machines
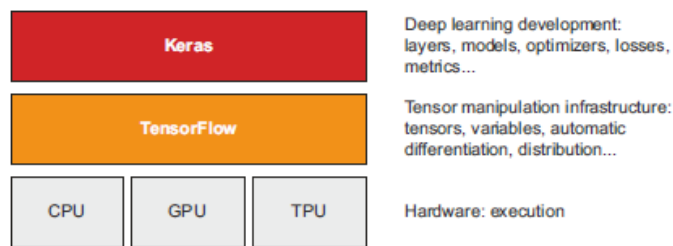


Figure 3.1   Keras and TensorFlow: TensorFlow is a low-level tensor computing platform, and Keras is a high-level deep learning API

➢ Keras is known for prioritizing the developer experience. It's an API for human beings, not machines.
   ✧ It follows best practices for reducing cognitive load: it offers consistent and simple workflows, it minimizes the number of actions required for common use cases, it provides clear and actionable feedback upon user error.
   ✧ This makes Keras easy to learn for a beginner, and highly productive to use for an expert.

➢ Keras has well over a million users as of late 2021, ranging from academic researchers, engineers, and data scientist at both startups and large companies to graduate students and hobbyists.
   ✧ Keras is used at Google, Netflix, Uber, CERN, NASA, Yelp, Instacart, Square, and hundreds of startups working on a wide range of problems across every industry.
   ✧ Your YouTube recommendations originate from Keras models.
   ✧ The Waymo self-driving cars are developed with Keras models.
   ✧ Keras is also a popular framework on Kaggle, the machine learning competition website, where most deep learning competitions have been won using Keras

➢ Because Keras has a large and diverse user base, it doesn't force you to follow a single "true" way of building and training models. Rather, it enables a wide range of different workflows, from the very high level to the very low level, corresponding to different user profiles.
   ✧ For instance, you have an array of ways to build models and an array of ways to train them, each representing a certain trade-off between usability and flexibility.
   ✧ In chapter 5, we'll review in detail a good fraction of this spectrum of workflows.
   ✧ You could be using Keras like you would use Scikit-learn – just calling *fit()* and letting the framework do its thing – or you could be using it like NumPy – taking full control of every little detail

➢ This means that everything you're learning now as you're getting started will still be relevant once you've become an expert.
   ✧ You can get started easily and then gradually dive into workflows where you're writing more and more logic from scratch.
   ✧ You won't have to switch to an entirely different framework as you go from student to researcher, or from data scientist to deep learning engineer.

➢ This philosophy is like that of Python itself!
   ✧ Some languages only offer one way to write programs – for instance, object-oriented programming or functional programming.

- ✧ Meanwhile, Python is a multiparadigm language: it offers an array of possible usage patterns that all work nicely together. This makes Python suitable to a wide range of very different use cases: system administration, data science, machine learning, engineering, web development … or just learning how to program.
  - ➢ Likewise, you can think of Keras as the Python of deep learning: a user-friendly deep leaning language that offers a variety of workflows to different user profiles.

# 3.　Keras and TensorFlow: brief history

- Keras predates TensorFlow by eight months. It was released in March 2015, and TensorFlow was released in November 2015.
  - ➢ If Keras is built on top of TensorFlow, how it could exist before TensorFlow was released?
    - ✧ Keras was originally built on top of Theano, another tensor-manipulation library that provided automatic differentiation and GPU support.
  - ➢ In late 2015, after the release of TensorFlow, Keras was refactored to a multi-backend architecture: it became possible to use Keras with either Theano or TensorFlow, and switching between the two was as easy as changing an environment variable.
  - ➢ By September 2016, TensorFlow had reached a level of technical maturity where it became possible to make it the default backend option for Keras
  - ➢ In 2017, two new additional backend options were added to Keras: CNTK (developed by Microsoft) and MXNet (developed by Amazon).
    - ✧ Nowadays, both Theano and CNTK are out of development, and MXNet is not widely used outside of Amazon.
    - ✧ Keras is back to being a single-backend API – on top of TensorFlow

- Keras and TensorFlow have had a symbiotic relationship for many years.
  - ➢ Throughout 2016 and 2017, Keras became well known as the user-friendly way to develop TensorFlow applications, funneling new users into the TensorFlow ecosystem.
  - ➢ By late 2017, a majority of TensorFlow users were using it through Keras or in combination with Keras.
  - ➢ In 2018, the TensorFlow leadership picked Keras as TensorFlow's official high-level API.
    - ✧ As a result, the Keras API is front and center in TensorFlow 2.0, released in September – an extensive redesign of TensorFlow and Keras that takes into account over four years of user feedback and technical progress.

# 4.　Setting up a deep learning workspace

- A *notebook* is a file generated by the Jupyter Notebook app that you can edit in your browser.

- It mixes the ability to execute Python code with rich text-editing capabilities for annotating what you're doing.
- A notebook also allows you to break up long experiments into smaller pieces that can be executed independently, which makes development interactive and means you don't have to return all of your previous code if something goes wrong late in an experiment.

# 5. First steps with TensorFlow

- Recall that training a neural network revolves around the following concepts
  - First, low-level tensor manipulation – the infrastructure that underlines all modern machine learning. This translates to TensorFlow APIs
    - Tensors, including special tensors that store the network's state (variables)
    - Tensor operations such as addition, relu, matmul
    - Backpropagation, a way to compute the gradient of mathematical expressions (handled in TensorFlow via the GradientTape object)
  - Second, high-level deep learning concepts. This translates to Keras APIs:
    - Layers, which are combined into a model
    - A loss function, which defines the feedback signal used for learning
    - An optimizer, which determines how learning proceeds
    - Metrics to evaluate model performance, such as accuracy
    - A training loop that performs mini-batch stochastic gradient descent

## 5.1. Constant tensors and variables

- Tensors need to be created with some initial value.
  - For instance, you could create all-ones or all-zeros tensors, or tensors of values drawn from a random distribution

```
Listing 3.1   All-ones or all-zeros tensors

>>> import tensorflow as tf
>>> x = tf.ones(shape=(2, 1))          ◁———  Equivalent to
>>> print(x)                                  np.ones(shape=(2, 1))
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)
>>> x = tf.zeros(shape=(2, 1))         ◁———  Equivalent to
>>> print(x)                                  np.zeros(shape=(2, 1))
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)
```

- A significant difference between NumPy arrays and TensorFlow tensors is that TensorFlow tensors aren't assignable: they're constant.
    - ➢ For instance, in NumPy, you can do the following

    - ➢ Try to do the same thing in TensorFlow, and you will get an error: "EagerTensor object does not support item assignment."

- To train a model, we'll need to update its state, which is a set of tensors. If tensors aren't assignable, how do we do it? That's where *variables* come in. *tf.Variable* is the class meant to manage modifiable state in TensorFlow.
    - ➢ To create a variable, you need to provide some initial value, such as a random tensor.

    - ➢ The state of a variable can be modified via its *assign* method

    - ➢ It also works for a subset of the coefficients

➢ Similarly, *assign_add()* and *assign_sub()* are efficient equivalents of += and -=

```
Listing 3.8   Using assign_add()

>>> v.assign_add(tf.ones((3, 1)))
array([[2.],
       [2.],
       [2.]], dtype=float32)>
```

## 5.2.  Tensor operations: Doing math in TensorFlow

- Just like NumPy, TensorFlow offers a large collection of tensor operations to express mathematical formulas

```
Listing 3.9   A few basic math operations

a = tf.ones((2, 2))          Take the square.
b = tf.square(a)
c = tf.sqrt(a)                 Take the square root.
d = b + c                        Add two tensors (element-wise).
e = tf.matmul(a, b)
e *= d                            Take the product of two tensors
                                      (as discussed in chapter 2).
Multiply two tensors
    (element-wise).
```

➢ Importantly, each of the preceding operations gets executed on the fly: at any point, you can print what the current result is, just like in NumPy.
➢ We call this **eager execution**.

## 5.3.  Second look at GradientTape API

- So far, TensorFlow seems to look a lot like NumPy. Here's something NumPy can't do: retrieve the gradient of any differentiable expression with respect to any of its inputs
  ➢ Just open a *GradientTape* scope, apply some computation to one or several input tensors, and retrieve the gradient of the result with respect to the inputs.

```
Listing 3.10   Using the GradientTape

input_var = tf.Variable(initial_value=3.)
with tf.GradientTape() as tape:
    result = tf.square(input_var)
gradient = tape.gradient(result, input_var)
```

  ➢ So far, you've only seen the case where the input tensors in *tape.gradient()* were TensorFlow variables. It's actually possible for these inputs to be any arbitrary tensor.
    ✧ However, only *trainable variables* are tracked by default. With a constant tensor, you'd have to manually mark it being tracked by calling *tape.watch()* on it.

```
Listing 3.11   Using GradientTape with constant tensor inputs

input_const = tf.constant(3.)
with tf.GradientTape() as tape:
    tape.watch(input_const)
    result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

- Why is this necessary? Because it would be too expensive to preemptively store the information required to compute the gradient of anything with respect to anything.
  - To avoid wasting resources, the tape needs to know what to watch.
  - Trainable variables are watched by default because computing the gradient of a loss with regard to a list of trainable variables is the most common use of the gradient tape.

- The gradient tape is a powerful utility, even capable of computing *second-order gradients*, that is to say, the gradient of a gradient.
  - if you measure the position of a falling apple along a vertical axis over time and find that it verifies position(time) = 4.9 * time **2, what is its acceleration? Let's use two nested gradient tapes to find out

**Listing 3.12   Using nested gradient tapes to compute second-order gradients**

```
time = tf.Variable(0.)
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
        position =  4.9 * time ** 2
    speed = inner_tape.gradient(position, time)
acceleration = outer_tape.gradient(speed, time)
```

We use the outer tape to compute the gradient of the gradient from the inner tape. Naturally, the answer is 4.9 * 2 = 9.8.

## 5.4.  An end-to-end example: a linear classifier in pure TensorFlow

- In a machine learning job interview, you may be asked to implement a linear classifier from scratch in TensorFlow: a very simple task that serves as a filter between candidates who have some minimal machine learning background and those who don't

- Firstly, let's use some linearly separable synthetic data: two classes of points in a 2D plane.
  - We'll generate each class of points by drawing their coordinates from a random distribution with a specific covariance matrix and a specific mean
  - The covariance matrix describes the shape of the point cloud, and the mean describes its position in the plane
  - We'll reuse the same covariance matrix for both point clouds, but we'll use two different mean values

**Listing 3.13   Generating two classes of random points in a 2D plane**

```
num_samples_per_class = 1000
negative_samples = np.random.multivariate_normal(
    mean=[0, 3],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
positive_samples = np.random.multivariate_normal(
    mean=[3, 0],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
```

Generate the first class of points: 1000 random 2D points. cov=[[1, 0.5],[0.5, 1]] corresponds to an oval-like point cloud oriented from bottom left to top right.

Generate the other class of points with a different mean and the same covariance matrix.

  - Now, we have two arrays with shape (1000, 2). Let's stack them into a single array with shape (2000,2)

**Listing 3.14   Stacking the two classes into an array with shape (2000, 2)**

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

  - Now, let's generate the corresponding target labels, an array of zeros and ones of shape (2000, 1), where *targets[i, 0]* is 0 if *inputs[i]* belong to class 0
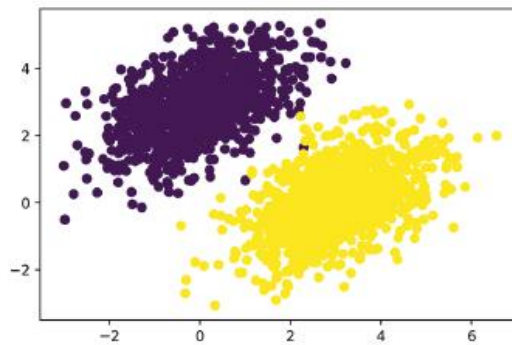
➢ Now, let's plot the data with Matplotlib

Figure 3.6   Our synthetic data: two classes of random points in the 2D plane

- Now, let's create a linear classifier that can learn to separate these two blobs.
  - ➢ A linear classifier is an affine transformation, $prediction = W * input + b$, trained to minimize the square of the difference between predictions and the targets.
  - ➢ We first create variables, W and b, initialized with random values and with zeros, respectively.

  - ➢ Here's our forward pass function

    - ✧ Note that our linear classifier operates on 2D inputs, W is really just two scalar coefficients, w1, and w2 and b is a single scalar coefficient.
    - ✧ Thus, for a given input point [x, y], its prediction value is w1 * x + w2 * y + b
  - ➢ We can define the loss function

  - ➢ Next is the training step, which receives some training data and updates the weights W and b so as to minimize the loss on the data.

- For simplicity, we'll do *batch training* instead of *mini-batch training*.
  - ➤ We'll run each training step (gradient computation and weight update) for all the data, rather than iterate over the data in small batches.
    - ✧ This means each training step will take much longer to run, but each gradient update will be much more effective at reducing the loss on the training data, since it will encompass information from all training samples instead of a portion of random samples.
    - ✧ Subsequently, we will need many fewer steps of training, and we should use a larger learning rate than we would typically use for mini-batch training (i.e. 0.1)

  - ➤ After 40 steps, the training loss seems to have stabilized around 0.025. We can plot how the linear model classifies the training data points.
    - ✧ Because our targets are zeros and ones, a given input point will be classified as "0" if its prediction value is below 0.5, and as "1" if it is above 0.5

```
predictions = model(inputs)
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
plt.show()
```
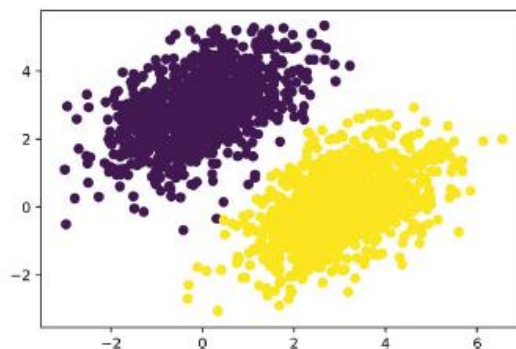


Figure 3.7  Our model's predictions on the training inputs: pretty similar to the training targets

  - ➤ Notice that class 0 is defined as w1 * x + w2 * y < 0.5 and class 1 is defined as w1 * x + w2 * y > 0.5
  - ➤ We are really looking at the equation of a line that is w1 * x + w2 * y = 0.5
  - ➤ Rearranging the equation gives y = -w1/w2 * x + (0.5-b) / 2
  - ➤ The plot is as follows

Generate 100 regularly spaced
numbers between −1 and 4, which
we will use to plot our line.

This is our line's
equation.

Plot our line ("-r"
means "plot it as
a red line").

```
x = np.linspace(-1, 4, 100)
y = - W[0] /  W[1] * x + (0.5 - b) / W[1]
plt.plot(x, y, "-r")
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
```

Plot our model's predictions on the same plot.
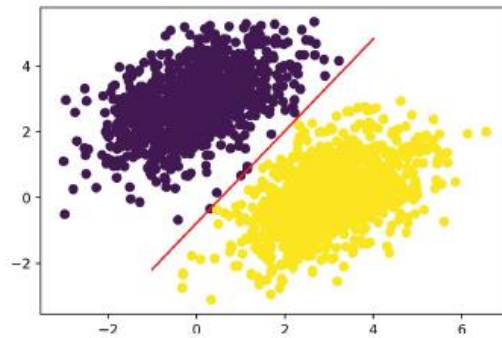
Figure 3.8   Our model,
visualized as a line

- This is really what a linear classifier is all about: finding the parameters of a line (or, in higher-dimensional spaces, a hyperplane) neatly separating two classes of data.

# 6.   Anatomy of NN: core Keras APIs

- At this point, you know the basics of TensorFlow, and you can use it to implement a toy model from scratch, such as the batch linear classifier in the previous section, or the toy neural network at the end of chapter 2. That's a solid foundation to build upon.

- It's now time to move on to a more productive, more robust path to deep learning: the Keras API

## 6.1.   Layers: the building blocks of deep learning

- The fundamental data structure in neural networks is the *layer*.
  - ➢ A layer is a data processing module that takes as input one or more tensors and that outputs one or more tensors.
  - ➢ Some layers are stateless, but more frequently layers have a state: the layer's *weights*, one or several tensors learned with stochastic gradient descent, which together contain the network's *knowledge*.

- Different types of layers are appropriate for different tensor formats and different types of data processing.

➤ Simple vector data, stored in rank-2 tensors of shape (samples, features), is often processed by ***densely connected*** layers, also called *fully connected* or *dense* layers

➤ Sequence data, stored in rank-3 tensors of shape (samples, timesteps, features), is typically processed by ***recurrent*** layers, such as an *LSTM* layer, or 1D convolution layers.

➤ Image data, stored in rank-4 tensors, is usually processed by 2D convolution layers

- The **base layer class** in Keras
    ➤ A simple API should have a single abstraction around which everything is centered. In Keras, that's the *Layer* class. Everything in Keras is either a *Layer* or something that closely interacts with a *Layer*.
        ✧ A *Layer* is an object that encapsulates some state (weights) and some computation (a forward pass).
            ▫ The weights are typically defined in a *build()* (although they could also be created in the constructor, *_init_()*),
            ▫ The computation is defined in the *call()* method
    ➤ In the previous chapter, we implemented a NaiveDense class that contained two weights W and b and applied the computation $output = activation(dot(input, W) + b)$
        ✧ This is what the same layer would look like in Keras

**Listing 3.22   A Dense layer implemented as a Layer subclass**

```
from tensorflow import keras                       All Keras layers inherit
                                                    from the base Layer class.
class SimpleDense(keras.layers.Layer):

    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation           Weight creation
                                               takes place in the
    def build(self, input_shape):              build() method.
        input_dim = input_shape[-1]
        self.W = self.add_weight(shape=(input_dim, self.units),
                                 initializer="random_normal")
        self.b = self.add_weight(shape=(self.units,),
                                 initializer="zeros")
    def call(self, inputs):                    add_weight() is a shortcut
        y = tf.matmul(inputs, self.W) + self.b method for creating weights.
        if self.activation is not None:        It is also possible to create
            y = self.activation(y)             standalone variables and assign
        return y                               them as layer attributes, like self.W =
                                               tf.Variable(tf.random.uniform(w_shape)).
```

We define the forward pass computation in the call() method.

        ✧ Note that in the next section, we'll cover in detail the purpose of the *build()* and *call()* methods
    ➤ Once instantiated, a layer like this can be used just like a function, taking as input a TensorFlow tensor:

```
>>> my_dense = SimpleDense(units=32, activation=tf.nn.relu)   Instantiate our
>>> input_tensor = tf.ones(shape=(2, 784))                    layer, defined
>>> output_tensor = my_dense(input_tensor)        Create      previously.
>>> print(output_tensor.shape)                    some test
(2, 32))                                           inputs.
                        Call the layer on
                        the inputs, just
                        like a function.
```

- Automatic **shape inference**: building layers on the fly
    ➤ Layers can be stacked together only if they are *compatible*. The notion here refers specifically to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape.

➢ Consider

```
from tensorflow.keras import layers
layer = layers.Dense(32, activation="relu")
```
A dense layer with
32 output units

✦ This layer will return a tensor where the first dimension has been transformed to be 32, it can only be connected to a downstream layer that expects 32-dimensional vectors as its input.

➢ When using Keras, you don't have to worry about size compatibility most of the time, because the layers you add to your models are dynamically built to match the shape of the incoming layers. For instance, suppose you write the following

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(32)
])
```

✦ The layers didn't receive any information about the shape of their inputs – instead, they automatically inferred their input shape as being the shape of the first inputs they see.

➢ In the NaiveDense that we fined earlier, we had to pass the layer's input size explicitly to the constructor in order to be able to create its weights.

✦ That's not ideal, because it would lead to models that look like this

```
model = NaiveSequential([
    NaiveDense(input_size=784, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=64, activation="relu"),
    NaiveDense(input_size=64, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=10, activation="softmax")
])
```

✦ It would be worse if the rules used by a layer to produce its output shape are complex.

▫ For instance, what if our layer returned outputs of shape ( $batch, input\_size *$ $2 \, if \, input\_size \% 2 == 0$ )?

➢ If we were to reimplement our NaiveDense layer as a Keras layer capable of automatic shape inference, it would look like the previous SimpleDense layer, with its *build()* and *call()* methods.

✦ In SimpleDense, we no longer create weights in the constructor like in the NaiveDense example; instead, we create them in a dedicated state-creation method, *build()*, which receives the first input shape seen by the layer as an argument. The *build()* method is called automatically the first time the layer is called (via its __call__() method). In fact, that's why we defined the computation in a separate call() method rather than in the __call__() method of the base layer schematically looks like this:

```
def __call__(self, inputs):
    if not self.built:
        self.build(inputs.shape)
        self.built = True
    return self.call(inputs)
```

➢ With automatic shape inference, our previous example becomes simple and neat:

```
model = keras.Sequential([
    SimpleDense(32, activation="relu"),
    SimpleDense(64, activation="relu"),

    SimpleDense(32, activation="relu"),
    SimpleDense(10, activation="softmax")
])
```

➢ Note that automatic shape inference is not the only thing that the Layer class's __call__() method handles. It takes care of many more things, in particular routing between *eager* and *graph* execution (chapter 7), and input masking (chapter 11).

❖ For now, just remember: when implementing your own layers, put the forward pass in the *call()* method.

## 6.2. From layers to models

- A deep learning model is a graph of layers.
  - ➢ In Keras, that's the *Model* class. Until now, you've only seen *Sequential* models (a subset of *Model*), which are simple stacks of layers, mapping a single input to a single output. As your move forward, you'll be exposed to a much broader variety of network topologies. These are some common ones
    - ❖ Two-branch networks
    - ❖ Multihead networks
    - ❖ Residual networks
  - ➢ Network topology can get quite involved. For instance, the following figure shows the topology of the graph of layers of a Transformer, a common architecture designed to process text data



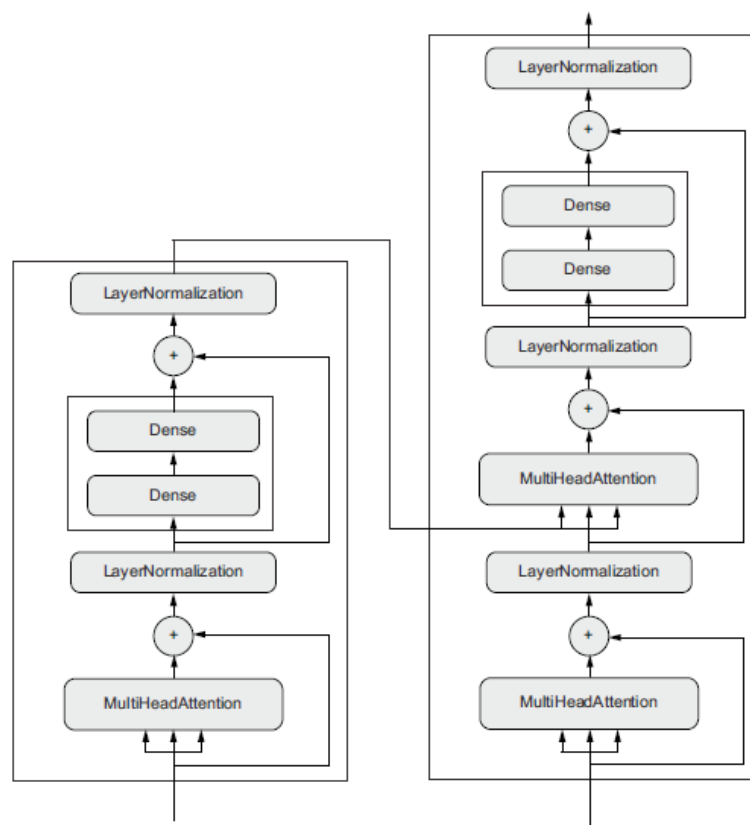**Figure 3.9   The Transformer architecture (covered in chapter 11). There's a lot going on here. Throughout the next few chapters, you'll climb your way up to understanding it.**

- There are generally two ways of building such models in Keras (covered in chapter 7)
  - ➢ Directly subclass the *Model* class,
  - ➢ Use the Functional API, which lets you do more with less code

- The topology of a model **defines a *hypothesis space*.**

- ➢ Recall that we described machine learning as searching for useful representations of some input data, within a predefined *space of possibilities*, using guidance from a feedback signal.
  - ➢ By choosing a network topology, you constrain your space of possibilities (hypothesis space) to a specific series of tensor operations, mapping input data to output data.
  - ➢ What you'll then be searching for is a good set of values for the weight tensors involved in these tensor operations.

- To learn from data, you have to make assumptions about it. These assumptions define what can be learned. As such, the structure of your hypothesis space – the architecture of your model – is extremely important.
  - ➢ It encodes the assumptions you make about your problem, the prior knowledge that the model starts with.
  - ➢ For instance, if you're working on a two-class classification problem with a model made of a single Dense layer with no activation (a pure affine transformation), you are assuming that your two classes are linearly separable.

- Picking the right network architecture is more an art than science, and although there are some best practices and principles you can rely on, only practice can help you become a proper neural-network architect.
- The next few chapters will both teach you explicit principles for building neural networks and help you develop intuition as to what works or doesn't work for specific problems.
- You'll build a solid intuition about
  - ➢ what type of model architectures work for different kinds of problems,
  - ➢ how to build these networks in practice,
  - ➢ how to pick the right learning configuration, and
  - ➢ how to tweak a model until it yields the results you want to see

## 6.3. The "compile" step: configuring the learning process

- Once the model architecture is defined, you still have to choose three more things
  - ➢ Loss function: the quantity that will be minimized during training. It represents a measure of success for the task at hand.
  - ➢ Optimizer: determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD)
  - ➢ Metrics: the measure of success you want to monitor during training and validation, such as classification accuracy.
    - ✧ Unlike the loss, training will not optimize directly for these metrics. As such, metrics don't need to be differentiable

- Once you've picked your loss, optimizer, and metrics, you can use the build-in *compile( )* and *fit( )* methods to start training your model.
  - ➢ Alternatively, you could also write your own custom training loops – covered in chapter 7

➢ The *compile( )* method configures the training process by taking the arguments *optimizer, loss, metrics*:

```
model = keras.Sequential([keras.layers.Dense(1)])     ← Define a linear classifier.
model.compile(optimizer="rmsprop",                     ← Specify the optimizer
              loss="mean_squared_error",                 by name: RMSprop
              metrics=["accuracy"])                      (it's case-insensitive).
                                                       ← Specify the loss
                                                         by name: mean
                                                         squared error.
           ← Specify a list of metrics: in
             this case, only accuracy.
```

➢ Notice that we passed the optimizer, loss, and metrics as strings. These strings are actually shortcuts that get converted to Python objects.

❖ For instance, "rmsprop" becomes $keras.optimizers.RMSprop()$.

➢ Importantly, it's also possible to specify these arguments as object instances, like this

```
model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])
```

❖ This is useful if you want to pass your own custom losses or metrics, or if you want to further configure the objects you're using – for instance, by passing a learning_rate argument to the optimizer

```
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),
              loss=my_custom_loss,
              metrics=[my_custom_metric_1, my_custom_metric_2])
```

- In chapter 7, we'll cover how to create custom losses and metrics.
  ➢ In general, you won't have to create your own losses, metrics, or optimizers from scratch, because Keras offers a wide range of built-in options that is likely to include what you need
  ➢ Optimizers
    ❖ SGD (with or without momentum)
    ❖ RMSprop
    ❖ Adam
    ❖ Adagrad
    ❖ …
  ➢ Losses
    ❖ CategoricalCrossentropy
    ❖ SparseCategoricalCrossentropy
    ❖ BinaryCrossentropy
    ❖ MeanSquaredError
    ❖ KLDivergence
    ❖ CosineSimilarity
    ❖ …
  ➢ Metrics
    ❖ CategoricalAccuracy
    ❖ SparseCategoricalAccuracy
    ❖ BinaryAccuracy
    ❖ AUC
    ❖ Precision
    ❖ Recall
    ❖ …

- You'll see concrete applications of many of these options

## 6.4. Picking a loss function

- Choosing the right loss function for the right problem is extremely important: your network will take any shortcut it can to minimize the loss, so if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted.
  - ➤ Imagine a stupid, omnipotent AI trained via SGD with this poorly chosen objective function: "maximizing the average well-being of all humans alive"
    - ✧ To make its job easier, this AI might choose to kill all humans except a few and focus on the well-being of the remaining ones – because average well-being isn't affected by how many humans are left
    - ✧ That might not be what you intended!
  - ➤ Just remember that all neural networks you build will be just as ruthless in lowering their loss function – so choose the objective wisely, or you'll have to face unintended side effects.

- Fortunately, when it comes to common problems such as classification, regression, and sequence prediction, there are simple guidelines you can follow to choose the correct loss.
  - ➤ For instance, you'll use binary crossentropy for a two-class classification problem, categorical crossentropy for a multi-class classification problem, and so on.
  - ➤ Only when you're working on truly new research problems will you have to develop your own loss functions.

## 6.5. Understanding the fit( ) method

- After *compile( )* comes *fit( )*. This method implements the training loop itself. It has some key arguments
  - ➤ The *data* (inputs and targets) to train on.
    - ✧ It will typically be passed either in the form of NumPy arrays or a TensorFlow *Dataset* object. You will learn more about the object in the next chapters
  - ➤ The number of *epochs* to train for: how many times the training loop should iterate over the data passed
  - ➤ The batch size to use within each epoch of mini-batch gradient descent: the number of training examples considered to compute the gradients for one weight update step.



Listing 3.23   Calling `fit()` with NumPy data

```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128
)
```

The input examples, as a NumPy array

The corresponding training targets, as a NumPy array

The training loop will iterate over the data in batches of 128 examples.

The training loop will iterate over the data 5 times.

> ➢ The call to *fit( )* returns a *History* object. This object contains a *history* field, which is a dictionary mapping keys such as "loss" or specific metric names to the list of their per-epoch values

```
>>> history.history
{"binary_accuracy": [0.855, 0.9565, 0.9555, 0.95, 0.951],
 "loss": [0.6573270302042366,
          0.07434618508815766,
          0.07687718723714351,
          0.07412414988875389,
          0.07617757616937161]}
```

# 6.6.  Monitoring loss and metrics on validation data

- The goal of machine learning is not to obtain models that perform well on the training data, which is easy – just follow the gradient. The goal is to obtain models that perform well in general, and particularly on data points that the model has never encountered before.
  - ➢ Note that just because a model performs well on its training data doesn't mean it will perform well on data it has never seen
  - ➢ For instance, it's possible that your model could end up merely *memorizing* a mapping between your training samples and their targets, which would be useless for the task of predicting targets for data the model has never seen before.
    - ✧ More defaulted points are covered in chapter 5

- To keep an eye on how the model does on new data, it's standard practiced to reserve a subset of the training data as *validation data*: you won't be training the model on this data, but you will use it to compute a loss value and metrics value.
  - ➢ You do this by using the *validation_data* argument in *fit( )*.
  - ➢ Like the training data, the validation data would be passed as NumPy arrays or as a TensorFlow *Dataset* object.

**Listing 3.24   Using the `validation_data` argument**

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])

indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs))
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]
model.fit(
    training_inputs,        │ Training data, used to update
    training_targets,       │ the weights of the model
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets)
)
```

To avoid having samples from only one class in the validation data, shuffle the inputs and targets using a random indices permutation.

Reserve 30% of the training inputs and targets for validation (we'll exclude these samples from training and reserve them to compute the validation loss and metrics).

Validation data, used only to monitor the validation loss and metrics

- The value of the loss on the validation data is called the "validation loss", to distinguish it from the "training loss"

➢ Note that it's essential to keep the training data and validation data strictly separate: the purpose of validation is to monitor whether what the model is learning is actually useful on new data.
  ✧ If any of the validation data has been seen by the model during training, your validation loss and metrics will be flawed.

- Note that if you want to compute the validation loss and metrics after the training is complete, you can call the *evaluate( )* method

```
loss_and_metrics = model.evaluate(val_inputs, val_targets, batch_size=128)
```

➢ The *evaluate( )* will iterate in batches (of size specified) over the data passed and return a list of scalars, where the first entry is the validation loss and the following entries are the validation metrics. If the model has no metrics, only the validation loss is returned (rather than a list)

## 6.7. Inference: Using a model after training

- Once you've trained your model, you're going to want to use it to make predictions on new data. This is called *inference*. To do this, a naïve approach would simply be to __call__( ) the model:

```
predictions = model(new_inputs)
```
**Takes a NumPy array or TensorFlow tensor and returns a TensorFlow tensor**

➢ However, this will process all inputs passed at once, which may not be feasible if you're looking at a lot of data (in particular, it may require more memory than your GPU has)

- A better way to do inference is to use the *predict( )* method. It will iterate over the data in small batches and return a NumPy array of predictions.
  ➢ And unlike __call__( ), it can process TensorFlow *Dataset* objects.

```
predictions = model.predict(new_inputs, batch_size=128)
```
**Takes a NumPy array or a Dataset and returns a NumPy array**

  ➢ For instance, if we use *predict( )* on some of our validation data with the linear model we trained earlier, we get scalar scores that correspond to the model's prediction for each input example

```
>>> predictions = model.predict(val_inputs, batch_size=128)
>>> print(predictions[:10])
[[0.3590725 ]
 [0.82706255]
 [0.74428225]
 [0.682058  ]
 [0.7312616 ]
 [0.6059811 ]
 [0.78046083]
 [0.025846  ]
 [0.16594526]
 [0.72068727]]
```

# Summary

- TensorFlow is an industry-strength numerical computing framework that can run on CPU, GPU, or TPU.
  - It can automatically compute the gradient of any differentiable expression, it can be distributed to many devices, and it can export programs to various external runtimes – even JavaScript.

- Keras is the standard API for doing deep learning with TensorFlow. It's what we'll use throughout this book.

- Key TensorFlow objects include tensors, variables, tensor operations, and the gradient tape.

- The central class of Keras is the *Layer*. A *layer* encapsulates some weights and some computation. Layers are assembled into *models*.

- Before you start training a model, you need to pick an *optimizer*, a *loss*, and some *metrics*, which you specify via $model.compile()$ method.

- To train a model, you can use the *fit( )* method, which runs mini-batch gradient descent for you. You can also use it to monitor your loss and metrics on *validation data*, a set of inputs that the model doesn't see during training.

- Once your model is trained, you use the $model.predict()$ method to generate predictions on new inputs.

# Appendix: runtime

- Runtime is a stage of the programming lifecycle. It is the time that a program is running alongside all the external instructions needed for proper execution.
  - Some of these external instructions are called *runtime systems* or *runtime environments* and come as integral parts of the programming language.

- A runtime system creates a layer over the operating system that contains other programs that handle the tasks needed to get the main program running.
  - These other programs handle tasks such as allocating memory for the main program and scheduling it.

- When an application is at the runtime stage, the executable file of the program is loaded into RAM, along with any files that the program references.

➢ These may include code that the user did not write but that works in the background to make the program run. That code is sent to the computer's processor to be turned into machine code. It then makes the hardware run the program