Lecture_3_TensorFlow_and_Keras

October 18, 2023

1 Introduction to TensorFlow and Keras

1.1 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 tensors**.

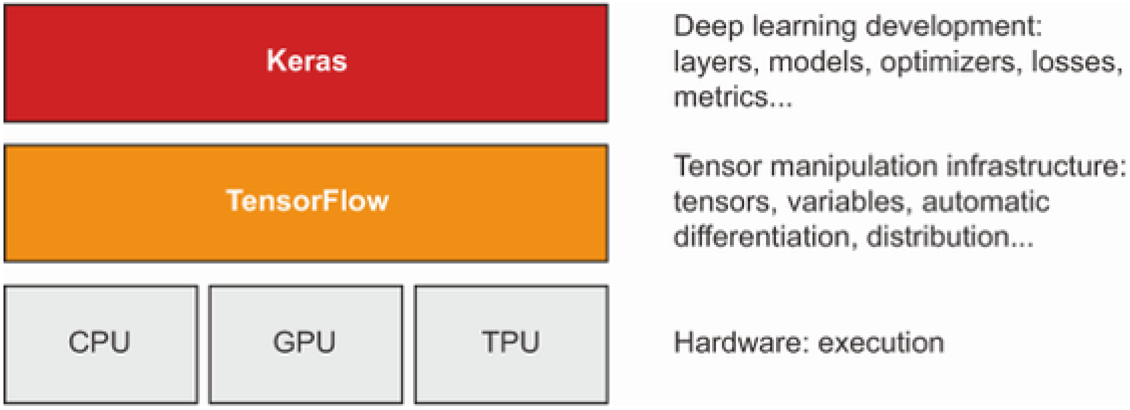
TensorFlow goes far beyond the scope of NumPy in the following ways: - It can automatically compute the gradient of any differentiable expression, making it highly suitable for ML. - It can run not only on CPUs, but also on GPUs and TPUs, highly parallel hardware accelerators. - Computation defined in 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.

TensorFlow scales pretty 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.

1.2 What's Keras?

Keras is a deep learning API for Python, built on top of TensorFlow, that provides a convenient way to define and train any kind of deep learning model.

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


Keras
Deep learning development: layers, models, optimizers, losses, metrics...
TensorFlow
Tensor manipulation infrastructure: tensors, variables, automatic differentiation, distribution...
CPU
GPU
TPU
Hardware: execution

Keras prioritizes the developer experience. 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's easy to learn for a beginner, and highly productive for an expert.

Keras 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 a very high level to a very low level, corresponding to different user profiles. For instance, you have multiple ways to build models and multiple ways to train them, each representing a certain trade-off between usability and flexibility.

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 DL engineer.

1.3 Setting up a DL workspace

Before developing DL applications, you need to set up your environment. It's highly recommended that you run DL code on a modern GPU (e.g. NVIDIA) rather than your computer's CPU. Some applications – in particular, image processing with convolutional networks – will be extremely slow on CPU, even a fast multicore CPU.

To do deep learning on a GPU, you have three options: - Buy and install a physical GPU on your workstation. - Use GPU instances on Google Cloud or AWS EC2. - Use the free GPU runtime from Colaboratory, a hosted notebook service offered by Google.

Colaboratory is the easiest way to start – it requires no hardware purchase and no software installation. Just open a tab in your browser and start coding.

However, the free version of Colaboratory is suitable for smaller projects and is time limited to 12 hours.

1.4 First steps with TensorFlow

Training a neural network revolves around the following concepts: 1. Low-level tensor manipulation - the infrastructure that underlies 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)

2. 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

1.4.1 Constant tensors and variables

[1]:

```
import tensorflow as tf
x = tf.constant([1, 2, 3, 4, 5, 6])
# If you want to specify type: x = tf.constant([1, 2, 3, 4, 5, 6],␣
 ↪dtype='float64')
# of x = tf.constant([1, 2, 3, 4, 5, 6], dtype=tf.dtypes.float32)
print(x)
x = tf.constant(x, shape=[2, 3])   # creates new tensor based on existing
print(x)
x = tf.reshape(x, shape=[3, 2])    # reshapes an existing tensor
print(x)
```

```
tf.Tensor([1 2 3 4 5 6], shape=(6,), dtype=int32)
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[1 2]
 [3 4]
 [5 6]], shape=(3, 2), dtype=int32)
```

All-ones or all-zeros tensors

[2]:

```
x = tf.ones(shape=(2, 1))     # By default, dtype=tf.dtypes.float32
print(x)
x = tf.zeros(shape=(2, 1))
print(x)
```

```
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)
```

Random tensors

[3]:

```
x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
print(x)
```

```
tf.Tensor(
[[ 0.92460215]
 [-0.7883522 ]
 [-0.11875589]], shape=(3, 1), dtype=float32)
```

[4]:

```
x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
print(x)
```

```
tf.Tensor(
[[0.04497004]
```

```
 [0.8335694 ]
 [0.6970067 ]], shape=(3, 1), dtype=float32)
```

NumPy arrays are assignable, TensorFlow tensors not!

A significant difference between NumPy arrays and TensorFlow tensors is that **TensorFlow tensors aren't assignable**: they're constant.

```
[5]: import numpy as np
     xn = np.ones(shape=(2, 2))
     xn[0, 0] = 0

     # If we try the same thing in TensorFlow, we will get an error: "EagerTensor␣
      ↪object does not support item assignment."
     # xt = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
     # xt[0, 0] = 0
```

Creating a TensorFlow variable

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.

`tf.Variable( initial_value=None, trainable=None, validate_shape=True, caching_device=None, name=None, variable_def=None, dtype=None, import_scope=None, constraint=None, synchronization=tf.VariableSynchronization.AUTO, aggregation=tf.compat.v1.VariableAggregation.NONE, shape=None )`

To create a variable, you need to provide some initial value, for example, a constant or a random tensor.

```
[6]: v_int = tf.Variable(initial_value=34)      # Optional name for the variable.␣
      ↪Defaults to 'Variable'
     v_float = tf.Variable(34., name='float_value')      # without parameter name␣
      ↪initial_value
     v_array = tf.Variable([[1.], [2.], [3.]], name='array')
     v_rand = tf.Variable(tf.random.normal(shape=(3, 1)))
     print(v_int)
     print(v_float)
     print(v_array)
     print(v_rand)
```

```
<tf.Variable 'Variable:0' shape=() dtype=int32, numpy=34>
<tf.Variable 'float_value:0' shape=() dtype=float32, numpy=34.0>
<tf.Variable 'array:0' shape=(3, 1) dtype=float32, numpy=
array([[1.],
       [2.],
       [3.]], dtype=float32)>
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[-0.21184683],
       [ 0.44106543],
       [ 1.0783477 ]], dtype=float32)>
```

[7]:

```
v_rand_numpy = v_rand.numpy()  # TensorFlow provides method numpy() to convert␣
 ↪a tensor to a numpy array
print(v_rand_numpy)
```

```
[[-0.21184683]
 [ 0.44106543]
 [ 1.0783477 ]]
```

Assigning a value to a TensorFlow variable

The state of a variable can be modified via its **`assign`** method, as follows

[8]:

```
v = tf.Variable(tf.random.normal(shape=(3,1)))
print(v)
v.assign(tf.constant([[1.], [2.], [3.]]))
print(v)
print("Numpy array: " + str(v.numpy()))
```

```
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[ 0.11889053],
       [-0.11158869],
       [-0.4178847 ]], dtype=float32)>
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[1.],
       [2.],
       [3.]], dtype=float32)>
Numpy array: [[1.]
 [2.]
 [3.]]
```

Assigning a value to a subset of a TensorFlow variable

[9]:

```
v[0,0].assign(3.)
# v[0,0] = 3. is not supported
```

[9]:

```
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[3.],
       [2.],
       [3.]], dtype=float32)>
```

[10]:

```
v2 = tf.Variable(tf.random.normal(shape=(3,3)))
v2[:,0].assign(tf.constant([1., 2., 3.]))    # try with int values and with 2D␣
 ↪tensor tf.constant([[1.], [2.], [3.]])
print(v2)
```

```
<tf.Variable 'Variable:0' shape=(3, 3) dtype=float32, numpy=
array([[ 1.        ,  1.5281631, -1.7391351],
       [ 2.        ,  0.3401864,  0.9879204],
       [ 3.        ,  1.3467306, -1.3396515]], dtype=float32)>
```

Using `assign_add` and `assign_sub`

[11]:
```
v.assign_add(tf.ones((3, 1)))
print(v)
v.assign_sub(2 * tf.ones((3, 1)))
print(v)
```

```
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[4.],
       [3.],
       [4.]], dtype=float32)>
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[2.],
       [1.],
       [2.]], dtype=float32)>
```

Tensor operations: Doing math in TensorFlow

[12]:
```
a = tf.ones((2, 3))
b = tf.square(a)       # Takes the element-wise square
c = tf.sqrt(a)         # Takes the element-wise square root
d = b + c              # Element-wise addition of two tensors
e = tf.matmul(d, tf.transpose(b))    # Product of two tensors (matrix product)
c *= d                 # Element-wise product of two tensors
```

The gradient tape in TensorFlow TensorFlow seems to look a lot like NumPy. But here's something NumPy cannot do: **calculate the gradient of any differentiable expression with respect to its inputs**.

The API through which we can leverage TensorFlow's powerful automatic differentiation capabilities is **GradientTape**. It's a Python scope that will "record" the tensor operations that run inside it, in the form of a computation graph. This graph can then be used to retrieve *the gradient of any output with respect to any variable or set of variables* (instances of the `tf.Variable` class).

A **`tf.Variable`** is a specific kind of tensor meant to hold mutable state—for instance, the weights of a neural network are always `tf.Variable` instances.

[13]:
```
x = tf.Variable(4.)
with tf.GradientTape() as tape:
    y = 2.7 * tf.square(x) + 3.1
grad_of_y_wrt_x = tape.gradient(y, x)
print(grad_of_y_wrt_x)
```

```
tf.Tensor(21.6, shape=(), dtype=float32)
```

By default, the resources held by a GradientTape are released as soon as **`GradientTape.gradient()`** method is called. To compute multiple gradients over the same computation, create a **persistent**

gradient tape. This allows multiple calls to the `gradient()` method, i.e., we can calculate multiple gradients with one GradientTape.

In case of persistent gradient tape, the resources are released when the tape object is garbage collected. We can manually delete persistent gradient tape with the `del` command.

[14]:
```
x = tf.Variable(4.)
with tf.GradientTape(persistent=True) as tape:
    y1 = 2.7 * tf.square(x) + 3.1
    y2 = tf.sqrt(x)
grad_of_y1_wrt_x = tape.gradient(y1, x)
grad_of_y2_wrt_x = tape.gradient(y2, x)
del tape                                  # We manually delete persistent␣
  ↪gradient tape
print(grad_of_y1_wrt_x)
print(grad_of_y2_wrt_x)
```

```
tf.Tensor(21.6, shape=(), dtype=float32)
tf.Tensor(0.25, shape=(), dtype=float32)
```

[15]:
```
# The GradientTape works with tensor operations:
x = tf.Variable(tf.random.uniform((2, 2)))
with tf.GradientTape() as tape:
    y = 2.7 * tf.square(x) + 3.1
grad_of_y_wrt_x = tape.gradient(y, x)
print(grad_of_y_wrt_x)
```

```
tf.Tensor(
[[3.0193236 3.165596 ]
 [0.4912691 4.1473656]], shape=(2, 2), dtype=float32)
```

[16]:
```
# It also works with lists of variables:
W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
# This is most commonly used to retrieve the gradients of the loss of a model␣
  ↪with respect to its weights:
# gradients = tape.gradient(loss, weights)
print(grad_of_y_wrt_W_and_b)
```

```
[<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[0.877681 , 0.877681 ],
       [0.9665097, 0.9665097]], dtype=float32)>, <tf.Tensor: shape=(2,),
dtype=float32, numpy=array([2., 2.], dtype=float32)>]
```

Using nested gradient tapes to compute second-order gradients

[17]:

```python
time = tf.Variable(2.)
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)
print(speed, acceleration)
```

```
tf.Tensor(19.6, shape=(), dtype=float32) tf.Tensor(9.8, shape=(), dtype=float32)
```

Using `GradientTape` with constant tensor inputs

So far, we've only used TensorFlow variables as input tensors in **`tape.gradient()`**. The input can be **any arbitrary tensor**, but only trainable variables are tracked by default. With a constant tensor, you'd have to *manually mark it as being tracked* by calling **`tape.watch()`** on it.

[18]:

```python
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)
print(gradient)
```

```
tf.Tensor(6.0, shape=(), dtype=float32)
```

1.5 An end-to-end example: A linear classifier in pure TensorFlow

Generating two classes of random points in a 2D plane

First, let's generate linearly separable synthetic data to work with: 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. Intuitively, the covariance matrix describes the shape of the point cloud, and the mean describes its position in the plane.

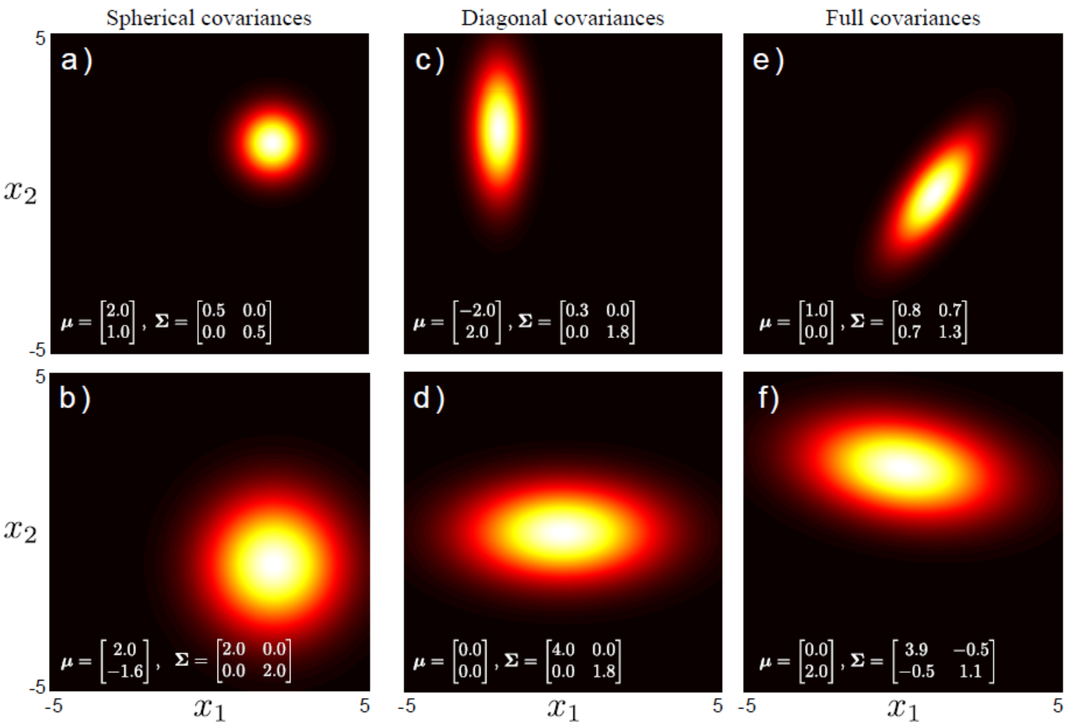
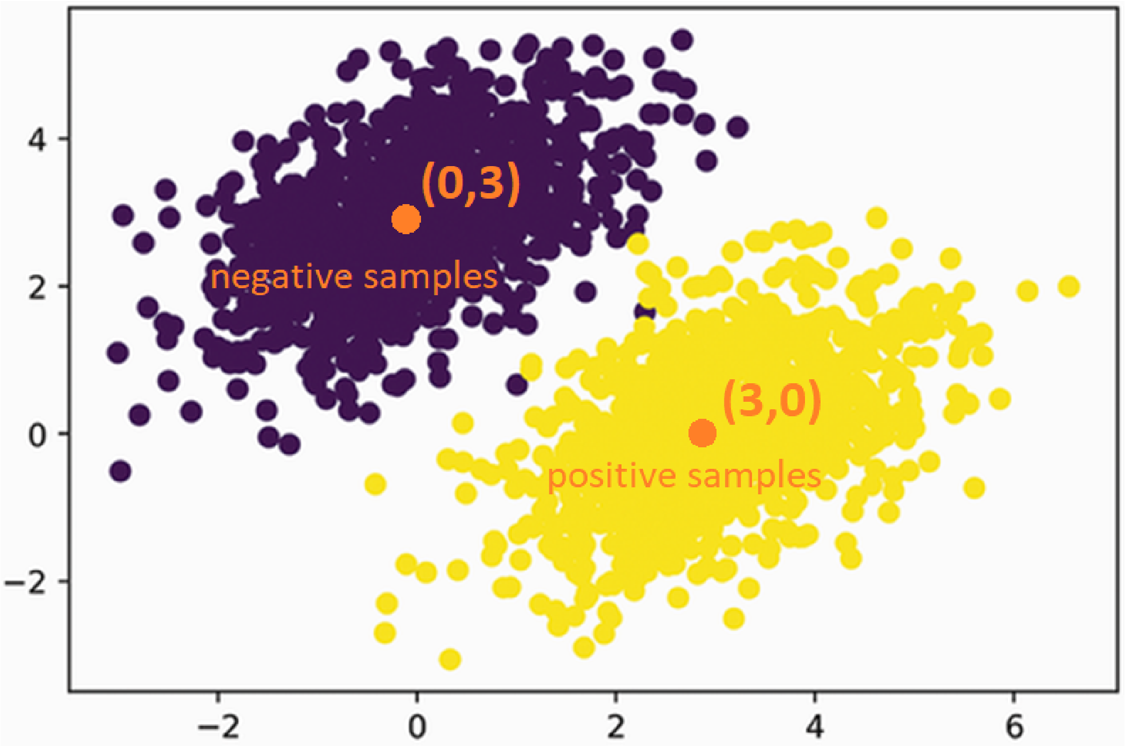


Image source: **Computer vision: models, learning and inference**, Simon J.D. Prince

We'll use the same covariance matrix for both point clouds, but we'll use two different mean values — the point clouds will have the same shape, but different positions.



[19]:
```
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]],     # Same covariance matrix
    size=num_samples_per_class)
```

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

In the preceding code, negative_samples and positive_samples are both arrays with shape (1000, 2). Let's stack them into a single array with shape (2000, 2).

[20]:
```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

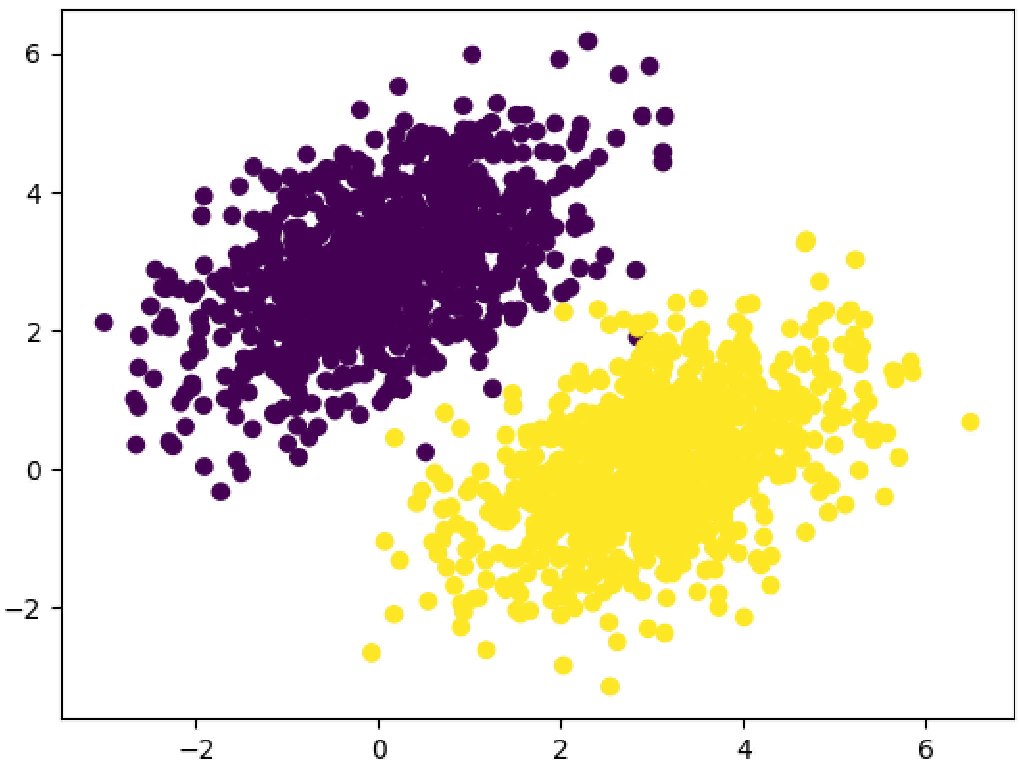
Generating the corresponding targets (0 and 1)

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]` belongs to class 0 (and inversely).

[21]:
```
targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype="float32"),
                     np.ones((num_samples_per_class, 1), dtype="float32")))
```

Plotting the two point classes

[22]:
```
import matplotlib.pyplot as plt
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])
plt.show()
```



Creating the linear classifier variables

Now let's create a linear classifier that can learn to separate these two blobs. A linear classifier is an affine transformation (`prediction = input • W + b`) trained to minimize the square of the difference between predictions and the targets.

Let's create our variables, `W` and `b`, initialized with random values and with zeros, respectively.

[23]:
```
input_dim = 2      # The inputs are 2D points
output_dim = 1     # The prediction is a single number (0 or 1) per point
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
```

The forward pass function

[24]:
```
def model(inputs):
    return tf.matmul(inputs, W) + b
```

Because our linear classifier operates on 2D inputs, `W` is two scalar coefficients, `w1` and `w2`: `W = [[w1], [w2]]`, whereas `b` is a single scalar coefficient. As such, for a given input point `[x, y]`, the prediction value equals:

`[x, y] * [[w1], [w2]] + b = w1 * x + w2 * y + b`

The mean squared error loss function

[25]:
```python
def square_loss(targets, predictions):
    per_sample_losses = tf.square(targets - predictions)
    return tf.reduce_mean(per_sample_losses)
```

`per_sample_losses` is a tensor with the same shape as targets and predictions, containing per sample losses.

`tf.reduce_mean` averages per sample losses.

The training step function

[26]:
```python
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(targets, predictions)
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss
```

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. On one hand, this means that each training step will take much longer to run, since we'll compute the forward pass and the gradients for 2,000 samples at once. On the other hand, 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, say, only 128 random samples. As a result, 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 (we'll use `learning_rate = 0.1`).

The batch training loop

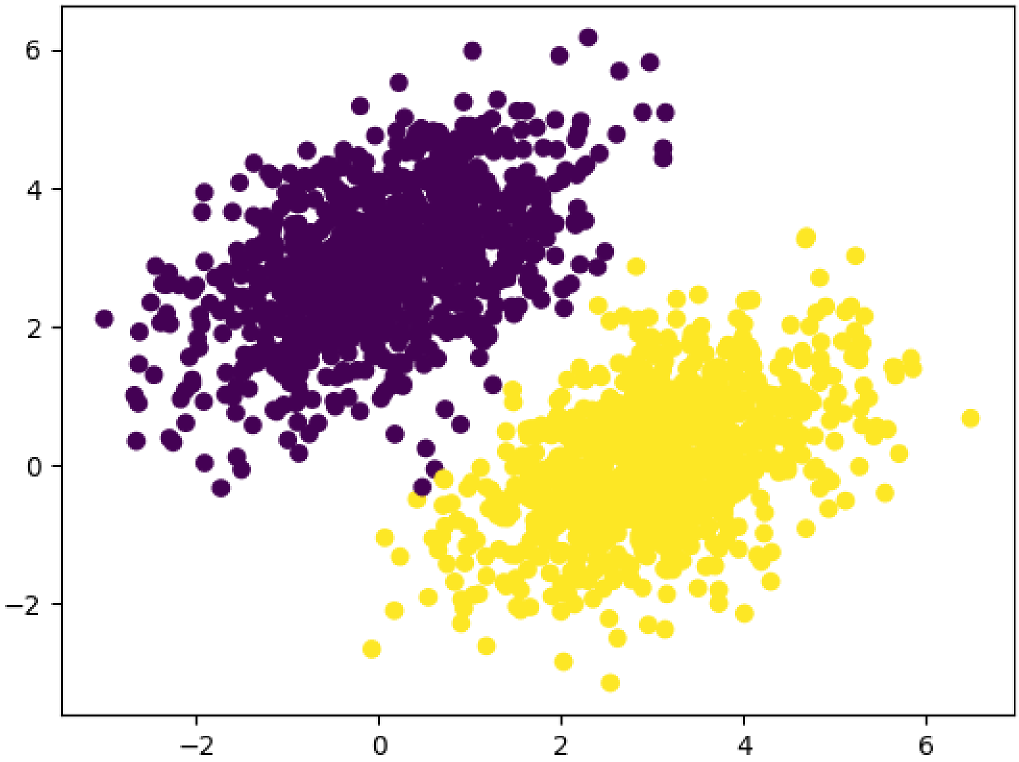
[27]:
```python
for step in range(40):
    loss = training_step(inputs, targets)
    print(f"Loss at step {step}: {loss:.4f}")
```

```
Loss at step 0: 5.6187
Loss at step 1: 0.5572
Loss at step 2: 0.1977
Loss at step 3: 0.1404
Loss at step 4: 0.1249
Loss at step 5: 0.1157
Loss at step 6: 0.1080
Loss at step 7: 0.1011
Loss at step 8: 0.0948
Loss at step 9: 0.0890
Loss at step 10: 0.0837
Loss at step 11: 0.0788
Loss at step 12: 0.0744
Loss at step 13: 0.0703
Loss at step 14: 0.0666
Loss at step 15: 0.0632
Loss at step 16: 0.0600
Loss at step 17: 0.0572
Loss at step 18: 0.0545
Loss at step 19: 0.0521
Loss at step 20: 0.0499
Loss at step 21: 0.0479
Loss at step 22: 0.0461
Loss at step 23: 0.0444
Loss at step 24: 0.0429
Loss at step 25: 0.0414
Loss at step 26: 0.0401
Loss at step 27: 0.0390
Loss at step 28: 0.0379
Loss at step 29: 0.0369
Loss at step 30: 0.0360
Loss at step 31: 0.0351
Loss at step 32: 0.0344
Loss at step 33: 0.0337
Loss at step 34: 0.0330
Loss at step 35: 0.0325
Loss at step 36: 0.0319
Loss at step 37: 0.0314
Loss at step 38: 0.0310
Loss at step 39: 0.0306
```

[28]:

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

# 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
```



Recall that the prediction value for a given point `[x, y]` is simply

`prediction = [x, y] * [[w1], [w2]] + b = w1 * x + w2 * y + b`

Thus, class 0 is defined as `w1 * x + w2 * y + b < 0.5`, and class 1 is defined as `w1 * x + w2 * y + b > 0.5`.

The equation of a line in the 2D plane which separates the classes is

`w1 * x + w2 * y + b = 0.5`

Above the line is class 1, and below the line is class 0. Let's rewrite the equation in the format `y = a * x + b`:

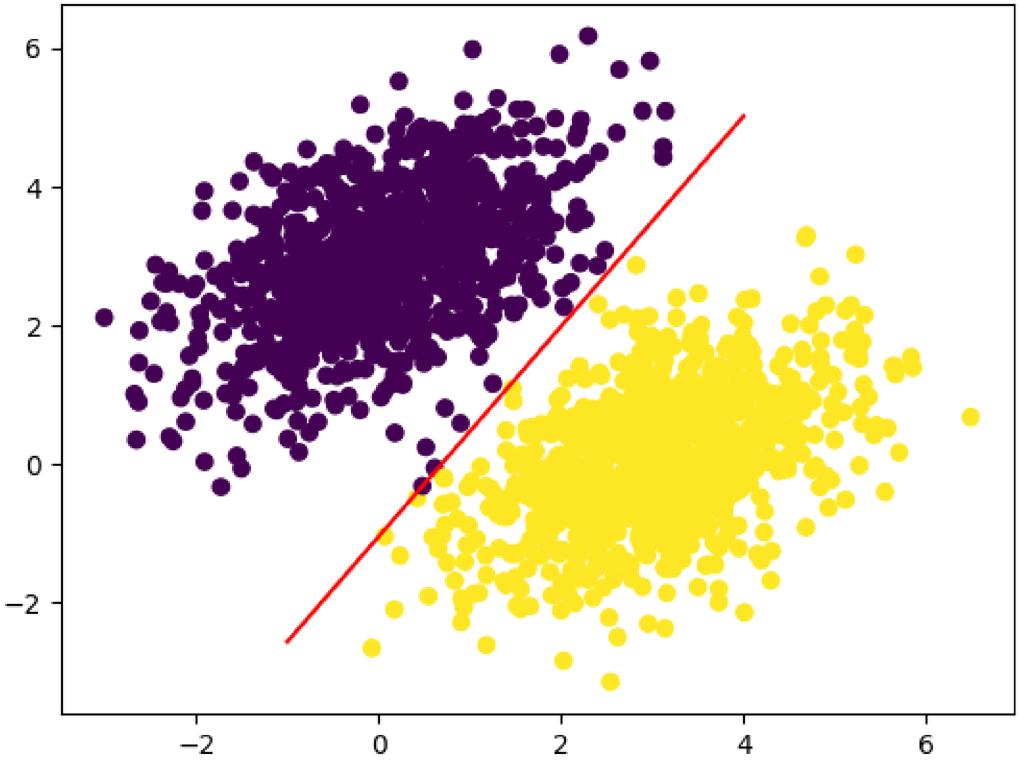
`y = - w1 / w2 * x + (0.5 - b) / w2`

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.

Let's plot this line.

[29]:
```
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)
```

```
[29]: <matplotlib.collections.PathCollection at 0x1dea65ecf40>
```

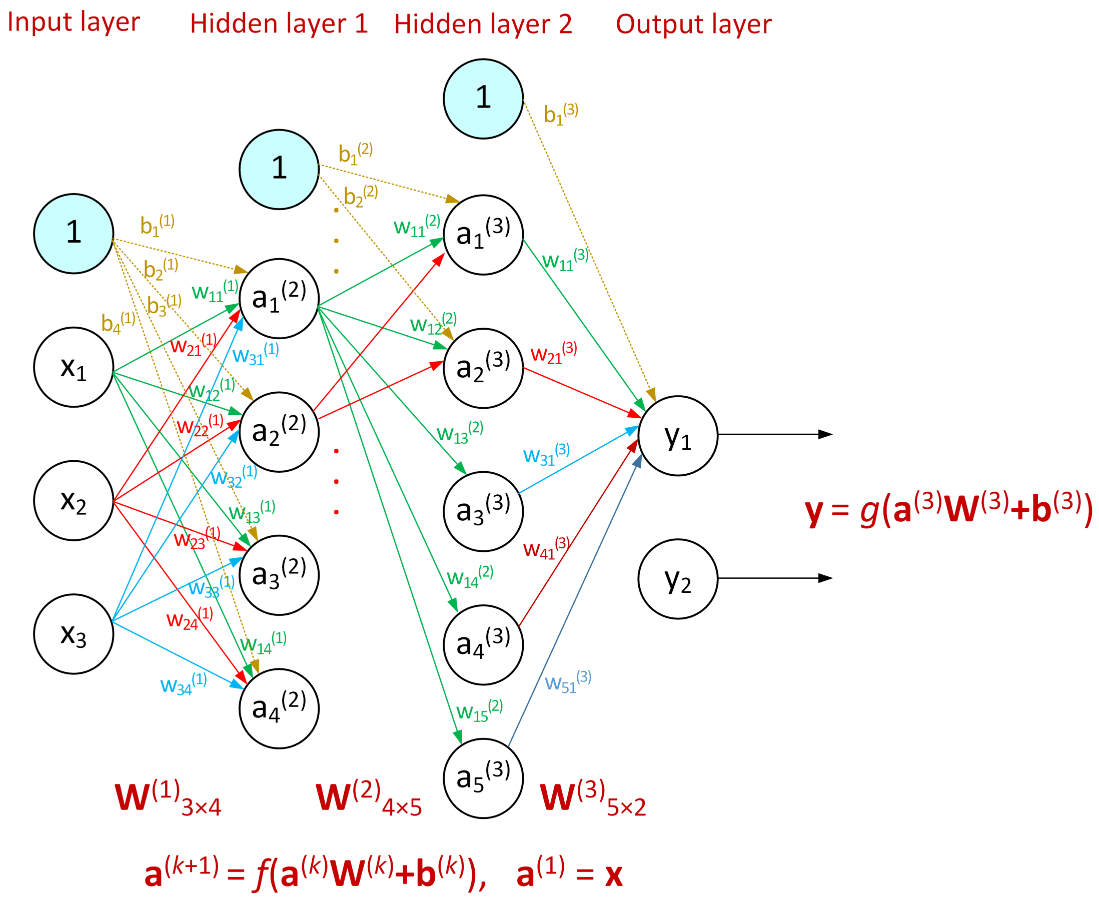


1.6 Anatomy of a neural network: Understanding core Keras APIs

1.6.1 Layers: The building blocks of deep learning

The fundamental data structure in neural networks is the **layer**, 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. For instance, 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 (the **`Dense`** class in Keras). 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 (**`Conv1D`**). Image data, stored in rank-4 tensors, is usually processed by 2D convolution layers (**`Conv2D`**).


Input layer
Hidden layer 1
Hidden layer 2
Output layer
1
1
1
x_1
x_2
x_3
$a_1^{(2)}$
$a_2^{(2)}$
$a_3^{(2)}$
$a_4^{(2)}$
$a_1^{(3)}$
$a_2^{(3)}$
$a_3^{(3)}$
$a_4^{(3)}$
$a_5^{(3)}$
y_1
y_2
$b_1^{(1)}$ $b_2^{(1)}$ $b_3^{(1)}$ $b_4^{(1)}$
$w_{11}^{(1)}$ $w_{21}^{(1)}$ $w_{31}^{(1)}$ $w_{12}^{(1)}$ $w_{22}^{(1)}$ $w_{32}^{(1)}$ $w_{13}^{(1)}$ $w_{23}^{(1)}$ $w_{33}^{(1)}$ $w_{14}^{(1)}$ $w_{24}^{(1)}$ $w_{34}^{(1)}$
$b_1^{(2)}$ $b_2^{(2)}$
$w_{11}^{(2)}$ $w_{12}^{(2)}$ $w_{13}^{(2)}$ $w_{14}^{(2)}$ $w_{15}^{(2)}$
$b_1^{(3)}$
$w_{11}^{(3)}$ $w_{21}^{(3)}$ $w_{31}^{(3)}$ $w_{41}^{(3)}$ $w_{51}^{(3)}$
$\mathbf{y} = g(\mathbf{a}^{(3)}\mathbf{W}^{(3)}+\mathbf{b}^{(3)})$
$\mathbf{W}^{(1)}_{3\times4}$
$\mathbf{W}^{(2)}_{4\times5}$
$\mathbf{W}^{(3)}_{5\times2}$
$\mathbf{a}^{(k+1)} = f(\mathbf{a}^{(k)}\mathbf{W}^{(k)}+\mathbf{b}^{(k)}),\quad \mathbf{a}^{(1)} = \mathbf{x}$

The base Layer class in Keras In Keras, the `Layer` class represents a single abstraction around which everything is centered. Everything in Keras is either a `Layer` or something that closely interacts with a `Layer`.

A `Layer` is a callable object that encapsulates some **state** (weights) and some **computation** (a forward pass). The weights are typically defined in the **`build()`** and the computation is defined in the **`call()`** method. Alternatively, the weights could also be created in the constructor **`__init__()`**.

Let's create a **`SimpleDense`** class that contains two weights **`W`** and **`b`** and implements computation:

`output = activation(dot(input, W) + b)`

A Dense layer implemented as a Layer subclass

[30]:
```
from tensorflow import keras

class SimpleDense(keras.layers.Layer): # All Keras layers inherit from the base Layer class

    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation

    def build(self, input_shape):     # In the build() method, we create layer␣
↪weights W and b
        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")
        # add_weight() adds a new variable to the layer. It is also possible to␣
↪create standalone variables
        # and assign them as layer attributes, like self.W = tf.Variable(tf.
↪random.uniform(w_shape)).

    def call(self, inputs):     # We define the forward pass computation in the␣
↪call() method.
        y = tf.matmul(inputs, self.W) + self.b
        if self.activation is not None:
            y = self.activation(y)
        return y
```

[31]:
```
# Once instantiated, a layer can be used just like a function, taking as input␣
↪a TensorFlow tensor
my_dense = SimpleDense(units=32, activation=tf.nn.relu)   # Create an object of␣
↪class SimpleDense (instantiate a layer)
input_tensor = tf.ones(shape=(2, 784))      # Create some test input (784 = 28 *␣
↪28, image flatten into an array)
output_tensor = my_dense(input_tensor)      # Call the layer on the inputs, just␣
↪like a function
print(output_tensor.shape)
```

```
(2, 32)
```

Automatic shape inference: Building layers on the fly Like with LEGO bricks, we can "clip" together layers that are compatible. The notion of layer compatibility 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 the following example:

[32]:
```
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 takes 32-dimensional input vectors.

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 layer**. For instance, suppose you write the following:

[33]:
```
from tensorflow.keras import models
model = models.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(64)
])
```

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.

With automatic shape inference, creating a network with multiple fully-connected layers, i.e. our **model**, is simple and neat:

[34]:
```
model = keras.Sequential([
    SimpleDense(32, activation="relu"),
    SimpleDense(64, activation="relu"),
    SimpleDense(32, activation="relu"),
    SimpleDense(10, activation="softmax")
])
```

Best practice: deferring weight creation until the shape of the inputs is known In many cases, you may not know in advance the size of your inputs, and you would like to **lazily** create weights when that value becomes known, some time after instantiating the layer. In the Keras API, creating layer weights in the `build(self, inputs_shape)` method of our layer is recommended.

The `__call__()` method of our layer will automatically call `build()` the first time the layer is called. At instantiation, we don't know on what inputs this is going to get called

`my_dense = SimpleDense(units=32, activation=tf.nn.relu)`

The layer's weights are created dynamically the first time the layer is called

`output_tensor = my_dense(input_tensor)`

Implementing `build()` separately as shown above nicely separates creating weights only once from using weights in every call.

1.6.2 The "compile" step: Configuring the learning process

Once the model architecture (number of layers, number of neurons per layer) is defined, we have to choose three more things: - **Loss function** (objective 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 measures of success we want to monitor during training and validation, such as classification accuracy. Unlike the loss, training will not optimize directly for these metrics. Therefore, *metrics don't need to be differentiable.*

Once we've picked our loss, optimizer, and metrics, we can use the built-in `compile()` and `fit()` methods to start training our model.

The `compile()` method configures the training process — we've already used it in our first neural network for MNIST classification.

Let's create a simple network (only input and output layers) to classify points in a 2D plane.

[35]:
```
model = keras.Sequential([keras.layers.Dense(1)])   # Define a linear␣
 ↪classifier, no hidden layers
model.compile(optimizer="rmsprop",
              loss="mean_squared_error",
              metrics=["accuracy"])                 # Specify a list of metrics:
 ↪ in this case, only accuracy
```

The strings "rmsprop", "mean_squared_error" and "accuracy" are shortcuts that are converted to Python objects. For instance, "rmsprop" becomes **`keras.optimizers.RMSprop()`**. It's also possible to specify these arguments as object instances:

[36]:
```
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),   # we␣
 ↪can pass arguments
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])
```

Optimizers: - **SGD** (with or without momentum) - **RMSprop** - **Adam** - **Adagrad**

Losses: - **CategoricalCrossentropy** - many-class classification - **SparseCategoricalCrossentropy** - many-class classification with integer labels - **BinaryCrossentropy** - two-class classification - **MeanSquaredError** - **KLDivergence** - **CosineSimilarity**

Metrics: - **CategoricalAccuracy** - **SparseCategoricalAccuracy** - **BinaryAccuracy** - **AUC** - **Precision** - **Recall**

1.6.3 Understanding the fit() method

After `compile()` comes `fit()`. The `fit()` method *implements the training loop*. Its key arguments are: - 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. - 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.

[37]:
```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128
)
```

```
Epoch 1/5
16/16 [==============================] - 0s 2ms/step - loss: 1.4696 -
binary_accuracy: 0.0275
Epoch 2/5
16/16 [==============================] - 0s 2ms/step - loss: 1.4558 -
binary_accuracy: 0.0275
Epoch 3/5
16/16 [==============================] - 0s 1ms/step - loss: 1.4439 -
binary_accuracy: 0.0275
```

```
Epoch 4/5
16/16 [==============================] - 0s 1ms/step - loss: 1.4323 -
binary_accuracy: 0.0280
Epoch 5/5
16/16 [==============================] - 0s 1ms/step - loss: 1.4208 -
binary_accuracy: 0.0280
```

[38]:
```
history.history
```

[38]:
```
{'loss': [1.469582438468933,
  1.4557862281799316,
  1.4439142942428589,
  1.432308554649353,
  1.4207680225372314],
 'binary_accuracy': [0.027499999850988388,
  0.027499999850988388,
  0.027499999850988388,
  0.0280000086426735,
  0.0280000086426735]}
```

1.6.4 Monitoring loss and metrics on validation data

The goal of ML is not to obtain models that perform well on the training data, which is easy — all you have to do is 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.

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.

To keep an eye on how the model does on new data, it's standard practice 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 could be passed as NumPy arrays or as a TensorFlow Dataset object.

Using the `validation_data` argument

[39]:
```
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()])

# To avoid having samples from only one class in the validation data, shuffle␣
 ↪the inputs and targets
# using a random indices permutation
indices_permutation = np.random.permutation(len(inputs))   #
shuffled_inputs = inputs[indices_permutation]
```

```
shuffled_targets = targets[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)
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_targets,
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets)  # Validation data, used only to␣
 ↪monitor the validation loss and metrics
)
```

```
Epoch 1/5
88/88 [==============================] - 0s 3ms/step - loss: 0.1360 -
binary_accuracy: 0.9600 - val_loss: 0.0426 - val_binary_accuracy: 0.9933
Epoch 2/5
88/88 [==============================] - 0s 2ms/step - loss: 0.0719 -
binary_accuracy: 0.9550 - val_loss: 0.1263 - val_binary_accuracy: 0.9750
Epoch 3/5
88/88 [==============================] - 0s 2ms/step - loss: 0.0774 -
binary_accuracy: 0.9579 - val_loss: 0.0634 - val_binary_accuracy: 0.9750
Epoch 4/5
88/88 [==============================] - 0s 2ms/step - loss: 0.0691 -
binary_accuracy: 0.9600 - val_loss: 0.0354 - val_binary_accuracy: 0.9950
Epoch 5/5
88/88 [==============================] - 0s 2ms/step - loss: 0.0742 -
binary_accuracy: 0.9621 - val_loss: 0.1175 - val_binary_accuracy: 0.9267
```

```
[39]: <keras.callbacks.History at 0x1deb0c1ef70>
```

The value of the loss on the validation data is called the **validation loss**, to distinguish it from the **training loss**. Tt'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.

The **`evaluate()`** method is used to compute the validation loss and metrics after the training is complete:

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

`evaluate()` will iterate in **batches** (of size `batch_size`) 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).

1.6.5 Inference: Using a model after training

Once we've trained the model, we can use it to make *predictions on new data*. This is called **inference**. To do this, a naive approach would simply be to `__call__()` the model:

```
predictions = model(new_inputs)
```

However, this will process all inputs in new_inputs 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 also process TensorFlow `Dataset` objects.

[40]:

```
predictions = model.predict(val_inputs, batch_size=128)
print(predictions[:10])
```

```
5/5 [==============================] - 0s 997us/step
[[1.13072   ]
 [1.3268945 ]
 [1.3697844 ]
 [0.02611524]
 [1.017559  ]
 [0.366431  ]
 [0.18915352]
 [0.05625391]
 [1.2945915 ]
 [1.2938304 ]]
```