



In tensors of type `tf.string`, the string length is not part of the tensor's shape. In other words, strings are considered as atomic values. However, in a Unicode string tensor (i.e., an `int32` tensor), the length of the string *is* part of the tensor's shape.

The `tf.strings` package contains several functions to manipulate string tensors, such as `length()` to count the number of bytes in a byte string (or the number of code points if you set `unit="UTF8_CHAR"`), `unicode_encode()` to convert a Unicode string tensor (i.e., `int32` tensor) to a byte string tensor, and `unicode_decode()` to do the reverse:

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: id=386, shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: id=393, shape=(4,), dtype=int32,
      numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

You can also manipulate tensors containing multiple strings:

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: id=299, shape=(4,), dtype=int32,
      numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
tf.RaggedTensor(values=tf.Tensor(
[  67  97 102 233  67 111 102 102 101 101  99  97
 102 102 232 21654 21857], shape=(17,), dtype=int32),
      row_splits=tf.Tensor([ 0  4 10 15 17], shape=(5,), dtype=int64))
>>> print(r)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101],
      [99, 97, 102, 102, 232], [21654, 21857]]>
```

Notice that the decoded strings are stored in a `RaggedTensor`. What is that?

Ragged Tensors

A ragged tensor is a special kind of tensor that represents a list of arrays of different sizes. More generally, it is a tensor with one or more *ragged dimensions*, meaning dimensions whose slices may have different lengths. In the ragged tensor `r`, the second dimension is a ragged dimension. In all ragged tensors, the first dimension is always a regular dimension (also called a *uniform dimension*).

All the elements of the ragged tensor `r` are regular tensors. For example, let's look at the second element of the ragged tensor:

```
>>> print(r[1])
tf.Tensor([ 67 111 102 102 101 101], shape=(6,), dtype=int32)
```

The `tf.ragged` package contains several functions to create and manipulate ragged tensors. Let's create a second ragged tensor using `tf.ragged.constant()` and concatenate it with the first ragged tensor, along axis 0:

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> print(tf.concat([r, r2], axis=0))
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

The result is not too surprising: the tensors in `r2` were appended after the tensors in `r` along axis 0. But what if we concatenate `r` and another ragged tensor along axis 1?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102, 102, 101, 101,
71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

This time, notice that the i^{th} tensor in `r` and the i^{th} tensor in `r3` were concatenated. Now that's more unusual, since all of these tensors can have different lengths.

If you call the `to_tensor()` method, it gets converted to a regular tensor, padding shorter tensors with zeros to get tensors of equal lengths (you can change the default value by setting the `default_value` argument):

```
>>> r.to_tensor()
<tf.Tensor: id=1056, shape=(4, 6), dtype=int32, numpy=
array([[ 67,  97, 102, 233,   0,   0],
       [ 67, 111, 102, 102, 101, 101],
       [ 99,  97, 102, 102, 232,   0],
       [21654, 21857,   0,   0,   0,   0]], dtype=int32)>
```

Many TF operations support ragged tensors. For the full list, see the documentation of the `tf.RaggedTensor` class.

Sparse Tensors

TensorFlow can also efficiently represent *sparse tensors* (i.e., tensors containing mostly zeros). Just create a `tf.SparseTensor`, specifying the indices and values of the nonzero elements and the tensor's shape. The indices must be listed in “reading order” (from left to right, and top to bottom). If you are unsure, just use `tf.sparse.reorder()`. You can convert a sparse tensor to a dense tensor (i.e., a regular tensor) using `tf.sparse.to_dense()`:

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
                        values=[1., 2., 3.],
                        dense_shape=[3, 4])
>>> tf.sparse.to_dense(s)
<tf.Tensor: id=1074, shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

Note that sparse tensors do not support as many operations as dense tensors. For example, you can multiply a sparse tensor by any scalar value, and you get a new sparse tensor, but you cannot add a scalar value to a sparse tensor, as this would not return a sparse tensor:

```
>>> s * 3.14
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x13205d470>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

Tensor Arrays

A `tf.TensorArray` represents a list of tensors. This can be handy in dynamic models containing loops, to accumulate results and later compute some statistics. You can read or write tensors at any location in the array:

```
array = tf.TensorArray(dtype=tf.float32, size=3)
array = array.write(0, tf.constant([1., 2.]))
array = array.write(1, tf.constant([3., 10.]))
array = array.write(2, tf.constant([5., 7.]))
tensor1 = array.read(1) # => returns (and pops!) tf.constant([3., 10.] )
```

Notice that reading an item pops it from the array, replacing it with a tensor of the same shape, full of zeros.



When you write to the array, you must assign the output back to the array, as shown in this code example. If you don't, although your code will work fine in eager mode, it will break in graph mode (these modes were presented in [Chapter 12](#)).

When creating a `TensorArray`, you must provide its `size`, except in graph mode. Alternatively, you can leave the `size` unset and instead set `dynamic_size=True`, but this will hinder performance, so if you know the `size` in advance, you should set it. You must also specify the `dtype`, and all elements must have the same shape as the first one written to the array.

You can stack all the items into a regular tensor by calling the `stack()` method:

```
>>> array.stack()
<tf.Tensor: id=2110875, shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

Sets

TensorFlow supports sets of integers or strings (but not floats). It represents them using regular tensors. For example, the set {1, 5, 9} is just represented as the tensor `[[1, 5, 9]]`. Note that the tensor must have at least two dimensions, and the sets must be in the last dimension. For example, `[[1, 5, 9], [2, 5, 11]]` is a tensor holding two independent sets: {1, 5, 9} and {2, 5, 11}. If some sets are shorter than others, you must pad them with a padding value (0 by default, but you can use any other value you prefer).

The `tf.sets` package contains several functions to manipulate sets. For example, let's create two sets and compute their union (the result is a sparse tensor, so we call `to_dense()` to display it):

```
>>> a = tf.constant([[1, 5, 9]])
>>> b = tf.constant([[5, 6, 9, 11]])
>>> u = tf.sets.union(a, b)
>>> u
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

You can also compute the union of multiple pairs of sets simultaneously:

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0, 0]])
>>> u = tf.sets.union(a, b)
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                                [ 0, 10, 13,  0,  0]], dtype=int32)>
```

If you prefer to use a different padding value, you must set `default_value` when calling `to_dense()`:

```
>>> tf.sparse.to_dense(u, default_value=-1)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                                [ 0, 10, 13, -1, -1]], dtype=int32)>
```



The default `default_value` is 0, so when dealing with string sets, you must set the `default_value` (e.g., to an empty string).

Other functions available in `tf.sets` include `difference()`, `intersection()`, and `size()`, which are self-explanatory. If you want to check whether or not a set contains some given values, you can compute the intersection of that set and the values. If you want to add some values to a set, you can compute the union of the set and the values.

Queues

A queue is a data structure to which you can push data records, and later pull them out. TensorFlow implements several types of queues in the `tf.queue` package. They used to be very important when implementing efficient data loading and preprocessing pipelines, but the `tf.data` API has essentially rendered them useless (except perhaps in some rare cases) because it is much simpler to use and provides all the tools you need to build efficient pipelines. For the sake of completeness, though, let's take a quick look at them.

The simplest kind of queue is the first-in, first-out (FIFO) queue. To build it, you need to specify the maximum number of records it can contain. Moreover, each record is a tuple of tensors, so you must specify the type of each tensor, and optionally their shapes. For example, the following code example creates a FIFO queue with maximum three records, each containing a tuple with a 32-bit integer and a string. Then it pushes two records to it, looks at the size (which is 2 at this point), and pulls a record out:

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])
>>> q.enqueue([10, b"windy"])
>>> q.enqueue([15, b"sunny"])
>>> q.size()
<tf.Tensor: id=62, shape=(), dtype=int32, numpy=2>
>>> q.dequeue()
[<tf.Tensor: id=6, shape=(), dtype=int32, numpy=10>,
 <tf.Tensor: id=7, shape=(), dtype=string, numpy=b'windy'>]
```

It is also possible to enqueue and dequeue multiple records at once (the latter requires specifying the shapes when creating the queue):

```
>>> q.enqueue_many([[13, 16], [b'cloudy', b'rainy']])
>>> q.dequeue_many(3)
[<tf.Tensor: [...] numpy=array([15, 13, 16], dtype=int32)>,
 <tf.Tensor: [...] numpy=array([b'sunny', b'cloudy', b'rainy'], dtype=object)>]
```

Other queue types include:

PaddingFIFOQueue

Same as `FIFOQueue`, but its `dequeue_many()` method supports dequeuing multiple records of different shapes. It automatically pads the shortest records to ensure all the records in the batch have the same shape.

PriorityQueue

A queue that dequeues records in a prioritized order. The priority must be a 64-bit integer included as the first element of each record. Surprisingly, records with a lower priority will be dequeued first. Records with the same priority will be dequeued in FIFO order.

RandomShuffleQueue

A queue whose records are dequeued in random order. This was useful to implement a shuffle buffer before `tf.data` existed.

If a queue is already full and you try to enqueue another record, the `enqueue*()` method will freeze until a record is dequeued by another thread. Similarly, if a queue is empty and you try to dequeue a record, the `dequeue*()` method will freeze until records are pushed to the queue by another thread.

TensorFlow Graphs

In this appendix, we will explore the graphs generated by TF Functions (see [Chapter 12](#)).

TF Functions and Concrete Functions

TF Functions are polymorphic, meaning they support inputs of different types (and shapes). For example, consider the following `tf_cube()` function:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

Every time you call a TF Function with a new combination of input types or shapes, it generates a new *concrete function*, with its own graph specialized for this particular combination. Such a combination of argument types and shapes is called an *input signature*. If you call the TF Function with an input signature it has already seen before, it will reuse the concrete function it generated earlier. For example, if you call `tf_cube(tf.constant(3.0))`, the TF Function will reuse the same concrete function it used for `tf_cube(tf.constant(2.0))` (for float32 scalar tensors). But it will generate a new concrete function if you call `tf_cube(tf.constant([2.0]))` or `tf_cube(tf.constant([3.0]))` (for float32 tensors of shape [1]), and yet another for `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (for float32 tensors of shape [2, 2]). You can get the concrete function for a particular combination of inputs by calling the TF Function's `get_concrete_function()` method. It can then be called like a regular function, but it will only support one input signature (in this example, float32 scalar tensors):


```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<tensorflow.python.eager.function.ConcreteFunction at 0x155c29240>
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: id=19068249, shape=(), dtype=float32, numpy=8.0>
```

Figure G-1 shows the `tf_cube()` TF Function, after we called `tf_cube(2)` and `tf_cube(tf.constant(2.0))`: two concrete functions were generated, one for each signature, each with its own optimized *function graph* (FuncGraph), and its own *function definition* (FunctionDef). A function definition points to the parts of the graph that correspond to the function's inputs and outputs. In each FuncGraph, the nodes (ovals) represent operations (e.g., power, constants, or placeholders for arguments like `x`), while the edges (the solid arrows between the operations) represent the tensors that will flow through the graph. The concrete function on the left is specialized for `x = 2`, so TensorFlow managed to simplify it to just output 8 all the time (note that the function definition does not even have an input). The concrete function on the right is specialized for float32 scalar tensors, and it could not be simplified. If we call `tf_cube(tf.constant(5.0))`, the second concrete function will be called, the placeholder operation for `x` will output 5.0, then the power operation will compute $5.0 ** 3$, so the output will be 125.0.

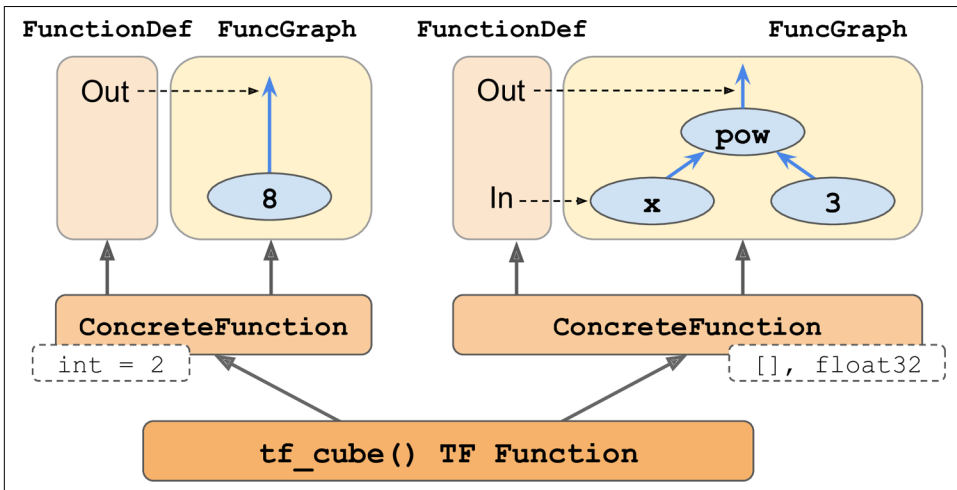


Figure G-1. The `tf_cube()` TF Function, with its `ConcreteFunctions` and their `FunctionGraphs`

The tensors in these graphs are *symbolic tensors*, meaning they don't have an actual value, just a data type, a shape, and a name. They represent the future tensors that will flow through the graph once an actual value is fed to the placeholder `x` and the graph is executed. Symbolic tensors make it possible to specify ahead of time how to

connect operations, and they also allow TensorFlow to recursively infer the data types and shapes of all tensors, given the data types and shapes of their inputs.

Now let's continue to peek under the hood, and see how to access function definitions and function graphs and how to explore a graph's operations and tensors.

Exploring Function Definitions and Graphs

You can access a concrete function's computation graph using the `graph` attribute, and get the list of its operations by calling the graph's `get_operations()` method:

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x14db5ef98>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

In this example, the first operation represents the input argument `x` (it is called a *placeholder*), the second “operation” represents the constant 3, the third operation represents the power operation (`**`), and the final operation represents the output of this function (it is an identity operation, meaning it will do nothing more than copy the output of the addition operation¹). Each operation has a list of input and output tensors that you can easily access using the operation's `inputs` and `outputs` attributes. For example, let's get the list of inputs and outputs of the power operation:

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

This computation graph is represented in [Figure G-2](#).

¹ You can safely ignore it—it is only here for technical reasons, to ensure that TF Functions don't leak internal structures.

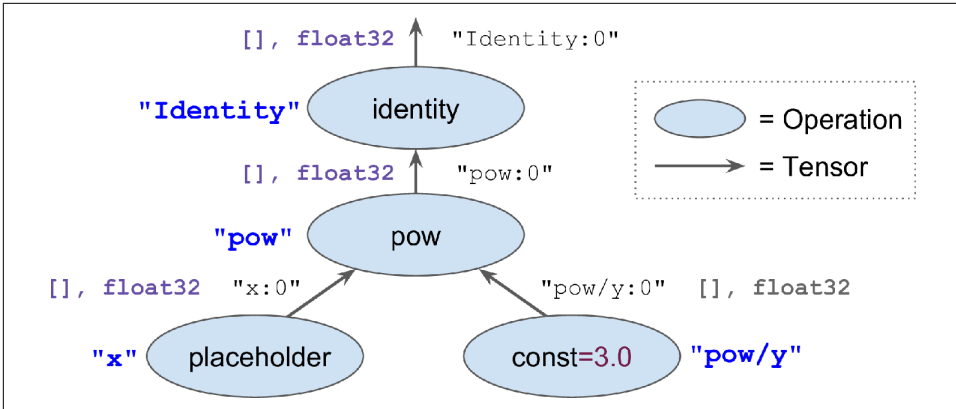


Figure G-2. Example of a computation graph

Note that each operation has a name. It defaults to the name of the operation (e.g., "pow"), but you can define it manually when calling the operation (e.g., `tf.pow(x, 3, name="other_name")`). If a name already exists, TensorFlow automatically adds a unique index (e.g., "pow_1", "pow_2", etc.). Each tensor also has a unique name: it is always the name of the operation that outputs this tensor, plus :0 if it is the operation's first output, or :1 if it is the second output, and so on. You can fetch an operation or a tensor by name using the graph's `get_operation_by_name()` or `get_tensor_by_name()` methods:

```
>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>
```

The concrete function also contains the function definition (represented as a protocol buffer²), which includes the function's signature. This signature allows the concrete function to know which placeholders to feed with the input values, and which tensors to return:

```
>>> concrete_function.function_def.signature
name: "__inference_cube_19068241"
input_arg {
  name: "x"
  type: DT_FLOAT
}
output_arg {
  name: "identity"
  type: DT_FLOAT
}
```

² A popular binary format discussed in [Chapter 13](#).

Now let's look more closely at tracing.

A Closer Look at Tracing

Let's tweak the `tf_cube()` function to print its input:

```
@tf.function
def tf_cube(x):
    print("x =", x)
    return x ** 3
```

Now let's call it:

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: id=19068290, shape=(), dtype=float32, numpy=8.0>
```

The result looks good, but look at what was printed: `x` is a symbolic tensor! It has a shape and a data type, but no value. Plus it has a name ("`x:0`"). This is because the `print()` function is not a TensorFlow operation, so it will only run when the Python function is traced, which happens in graph mode, with arguments replaced with symbolic tensors (same type and shape, but no value). Since the `print()` function was not captured into the graph, the next times we call `tf_cube()` with float32 scalar tensors, nothing is printed:

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

But if we call `tf_cube()` with a tensor of a different type or shape, or with a new Python value, the function will be traced again, so the `print()` function will be called:

```
>>> result = tf_cube(2) # new Python value: trace!
x = 2
>>> result = tf_cube(3) # new Python value: trace!
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]]) # New shape: trace!
x = Tensor("x:0", shape=(1, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]]) # New shape: trace!
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]]) # Same shape: no trace
```



If your function has Python side effects (e.g., it saves some logs to disk), be aware that this code will only run when the function is traced (i.e., every time the TF Function is called with a new input signature). It best to assume that the function may be traced (or not) any time the TF Function is called.

In some cases, you may want to restrict a TF Function to a specific input signature. For example, suppose you know that you will only ever call a TF Function with batches of 28×28 -pixel images, but the batches will have very different sizes. You may not want TensorFlow to generate a different concrete function for each batch size, or count on it to figure out on its own when to use None. In this case, you can specify the input signature like this:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2] # drop half the rows and columns
```

This TF Function will accept any float32 tensor of shape $[*, 28, 28]$, and it will reuse the same concrete function every time:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
preprocessed_images = shrink(img_batch_1) # Works fine. Traces the function.
preprocessed_images = shrink(img_batch_2) # Works fine. Same concrete function.
```

However, if you try to call this TF Function with a Python value, or a tensor of an unexpected data type or shape, you will get an exception:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])
preprocessed_images = shrink(img_batch_3) # ValueError! Unexpected signature.
```

Using AutoGraph to Capture Control Flow

If your function contains a simple for loop, what do you expect will happen? For example, let's write a function that will add 10 to its input, by just adding 1 10 times:

```
@tf.function
def add_10(x):
    for i in range(10):
        x += 1
    return x
```

It works fine, but when we look at its graph, we find that it does not contain a loop: it just contains 10 addition operations!

```
>>> add_10(tf.constant(0))
<tf.Tensor: id=19280066, shape=(), dtype=int32, numpy=10>
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'add' type=Add>, [...],
 <tf.Operation 'add_1' type=Add>, [...],
 <tf.Operation 'add_2' type=Add>, [...],
 [...],
 <tf.Operation 'add_9' type=Add>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

This actually makes sense: when the function got traced, the loop ran 10 times, so the `x += 1` operation was run 10 times, and since it was in graph mode, it recorded this operation 10 times in the graph. You can think of this for loop as a “static” loop that gets unrolled when the graph is created.

If you want the graph to contain a “dynamic” loop instead (i.e., one that runs when the graph is executed), you can create one manually using the `tf.nn.loop()` operation, but it is not very intuitive (see the “Using AutoGraph to Capture Control Flow” section of the Chapter 12 notebook for an example). Instead, it is much simpler to use TensorFlow’s *AutoGraph* feature, discussed in [Chapter 12](#). AutoGraph is actually activated by default (if you ever need to turn it off, you can pass `autograph=False` to `tf.function()`). So if it is on, why didn’t it capture the for loop in the `add_10()` function? Well, it only captures for loops that iterate over `tf.range()`, not `range()`. This is to give you the choice:

- If you use `range()`, the for loop will be static, meaning it will only be executed when the function is traced. The loop will be “unrolled” into a set of operations for each iteration, as we saw.
- If you use `tf.range()`, the loop will be dynamic, meaning that it will be included in the graph itself (but it will not run during tracing).

Let’s look at the graph that gets generated if you just replace `range()` with `tf.range()` in the `add_10()` function:

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'range' type=Range>, [...],
 <tf.Operation 'while' type=While>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

As you can see, the graph now contains a While loop operation, as if you had called the `tf.nn.loop()` function.

Handling Variables and Other Resources in TF Functions

In TensorFlow, variables and other stateful objects, such as queues or datasets, are called *resources*. TF Functions treat them with special care: any operation that reads or updates a resource is considered stateful, and TF Functions ensure that stateful operations are executed in the order they appear (as opposed to stateless operations, which may be run in parallel, so their order of execution is not guaranteed). Moreover, when you pass a resource as an argument to a TF Function, it gets passed by reference, so the function may modify it. For example:

```

counter = tf.Variable(0)

@tf.function
def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter) # counter is now equal to 1
increment(counter) # counter is now equal to 2

```

If you peek at the function definition, the first argument is marked as a resource:

```

>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE

```

It is also possible to use a `tf.Variable` defined outside of the function, without explicitly passing it as an argument:

```

counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)

```

The TF Function will treat this as an implicit first argument, so it will actually end up with the same signature (except for the name of the argument). However, using global variables can quickly become messy, so you should generally wrap variables (and other resources) inside classes. The good news is `@tf.function` works fine with methods too:

```

class Counter:
    def __init__(self):
        self.counter = tf.Variable(0)

    @tf.function
    def increment(self, c=1):
        return self.counter.assign_add(c)

```



Do not use `=`, `+=`, `-=`, or any other Python assignment operator with TF variables. Instead, you must use the `assign()`, `assign_add()`, or `assign_sub()` methods. If you try to use a Python assignment operator, you will get an exception when you call the method.

A good example of this object-oriented approach is, of course, `tf.keras`. Let's see how to use TF Functions with `tf.keras`.

Using TF Functions with tf.keras (or Not)

By default, any custom function, layer, or model you use with tf.keras will automatically be converted to a TF Function; you do not need to do anything at all! However, in some cases you may want to deactivate this automatic conversion—for example, if your custom code cannot be turned into a TF Function, or if you just want to debug your code, which is much easier in eager mode. To do this, you can simply pass `dynamic=True` when creating the model or any of its layers:

```
model = MyModel(dynamic=True)
```

If your custom model or layer will always be dynamic, you can instead call the base class's constructor with `dynamic=True`:

```
class MyLayer(keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

Alternatively, you can pass `run_eagerly=True` when calling the `compile()` method:

```
model.compile(loss=my_mse, optimizer="nadam", metrics=[my_mae],
              run_eagerly=True)
```

Now you know how TF Functions handle polymorphism (with multiple concrete functions), how graphs are automatically generated using AutoGraph and tracing, what graphs look like, how to explore their symbolic operations and tensors, how to handle variables and resources, and how to use TF Functions with tf.keras.

Symbols

1cycle scheduling, 361
1D convolutional layers, 520

A

A/B experiments, 667
accelerated K-Means, 244
accuracy
 defined, 89
 example of, 2
 measuring using cross-validation, 89
action advantage, 620
action step, 656
actions
 evaluating, 619
 exploiting versus exploring, 618
activation functions
 exponential linear unit (ELU), 336-338
 hyperbolic tangent (tanh), 291
 Logistic (sigmoid), 143, 293, 302, 332
 nonsaturating, 335
 Rectified Linear Unit function (ReLU), 292-293
 Scaled Exponential Linear Unit (SELU), 334, 337-338, 368
 softmax, 294, 299, 470, 482, 488, 543
 softplus, 293
active constraint, 762
active learning, 255
Actor-Critic algorithms, 625, 662
AdaBoost, 200
AdaGrad, 354
Adam and Nadam optimization, 356
Adaptive Boosting, 200
adaptive instance normalization (AdaIN), 604
adaptive learning rate, 355
adaptive moment estimation, 356
additive attention, 550
Advantage Actor-Critic (A2C), 663
adversarial learning, 495, 568
affine transformations, 604
affinity, 237
affinity propagation, 259
agents, 14
agglomerative clustering, 258
AI Platform, 680
Akaike information criterion (AIC), 267
AlexNet, 464
algorithms
 Actor-Critic algorithms, 625, 662
 Advantage Actor-Critic (A2C), 663
 AllReduce algorithm, 705
 Asynchronous Advantage Actor-Critic (A3C), 662
 BIRCH algorithm, 259
 CART training algorithm, 177, 179
 clustering algorithms, 10
 Dueling DQN algorithm, 641
 dynamic placer algorithm, 697
 Expectation-Maximization (EM) algorithm, 262
 for anomaly detection, 274
 genetic algorithms, 612
 greedy algorithms, 180
 hierarchical clustering algorithms, 10
 importance of data over, 24
 Isolation Forest algorithm, 274
 isomap algorithm, 233

- K-Means algorithm, 238
- Lloyd-Forgy algorithm, 238
- Mean-Shift algorithm, 259
- off-policy algorithms, 632
- on-policy algorithms, 632
- one-class SVM algorithm, 275
- Proximal Policy Optimization (PPO), 663
- Randomized PCA algorithm, 225
- REINFORCE algorithms, 620
- Soft Actor-Critic algorithm, 663
- supervised learning, 8
- unsupervised learning, 9
- Value Iteration algorithm, 627
- visualization algorithms, 11
- AllReduce algorithm, 705
- alpha channels, 250
- anchor boxes, 490
- anomaly detection
 - additional algorithms for, 274
 - examples of, 12
 - goal of, 236
 - using clustering, 237
 - using Gaussian Mixtures, 266
- Approximate Q-Learning, 633
- area under the curve (AUC), 98
- argmax operator, 149
- artificial neural networks (ANNs)
 - Boltzmann machines, 775
 - fine-tuning hyperparameters for, 320-327
 - from biological to artificial neurons, 280-295
 - Hopfield networks, 773
 - implementing MLPs with Keras, 295-320
 - overview of, 279
 - restricted Boltzmann machines (RBMs), 776
 - self-organizing maps (SOMs), 780
- artificial neurons, 283
- association rule learning, 12
- associative memory networks, 773
- Asynchronous Advantage Actor-Critic (A3C), 662
- asynchronous updates, 707
- Atari preprocessing, 645
- attention mechanisms
 - defined, 526
 - explainability and, 553
 - overview of, 549
 - Transformer architecture, 554
 - visual attention, 552
- attributes, 8
- autoencoders
 - convolutional, 579
 - denoising, 581
 - efficient data representations, 569
 - generative, 586
 - versus Generative Adversarial Networks (GANs), 568
 - overview of, 567
 - parts of, 569
 - PCA with undercomplete linear autoencoders, 570
 - probabilistic, 586
 - recurrent, 580
 - sparse, 582
 - stacked, 572-575
 - undercomplete, 570
 - unsupervised pretraining using stacked, 576-579
 - variational, 586-591
- AutoGraphs, 407
- automatic differentiation (autodiff), 290, 399, 765-772
- AutoML, 323
- autonomous driving systems, 497
- autoregressive integrated moving average (ARIMA) models, 506
- average absolute deviation, 41
- average pooling layer, 459
- Average Precision (AP), 491

B

- backpropagation, 289-292
- backpropagation through time (BPTT), 502
- bag of words, 438
- bagging and pasting
 - out-of-bag evaluation, 195
 - overview of, 192
 - in Scikit-Learn, 194
- Bahdanau attention, 550
- bandwidth saturation, 708
- basic cells, 500
- Batch Gradient Descent, 121
- batch learning, 15
- Batch Normalization (BN), 339
- batch size, 325
- batched action step, 657
- batched time step, 657
- batched trajectory, 657

- Bayesian Gaussian Mixture models, 270
- Bayesian inference, 586
- Bayesian information criterion (BIC), 267
- beam search, 547
- beam width, 547
- Bellman Optimality Equation, 627
- Better Life Index, 19
- bias neurons, 285
- bias terms, 112
- bias/variance trade-off, 134
- bidirectional recurrent layers, 546
- bidirectional RNNs, 546
- binary classifiers, 88
- binary trees, 177
- biological neural networks (BNN), 282
- biological neurons, 280
- BIRCH algorithm, 259
- black box models, 178
- black box stochastic variational inference (BBSVI), 273
- blenders, 208
- Boltzmann machines, 775
- boosting
 - AdaBoost, 200
 - Gradient Boosting, 203
 - overview of, 199
- bottleneck layers, 467
- boundary transitions, 660
- bounding box priors, 490
- break the symmetry, 291
- Byte-Pair Encoding, 536

C

- calculus, 112
- California Housing Prices dataset, 36
- callbacks, 315
- canary testing, 684
- CART training algorithm, 177, 179
- catastrophic forgetting, 637
- categorical distribution, 261
- categorical features
 - encoding using embeddings, 433
 - encoding using one-hot vectors, 431
- causal models, 510
- centroids, 238
- chain rule, 290
- chaining transformations, 415
- character RNNs (Char-RNNs)
 - building and training, 530

- chopping sequential datasets, 528
- generating Shakespearean text, 531
- overview of, 526
- splitting sequential datasets, 527
- stateful RNNs and, 532
- training dataset creation, 527
- using, 531
- chatbots, 525
- chi-squared test, 182
- Classification and Regression Tree (CART), 177, 179
- classification problems
 - AdaBoost classifiers, 200
 - binary classifiers, 88
 - classification and localization, 483
 - classification MLPs, 294
 - error analysis, 102
 - example of, 8
 - Extra-Trees classifier, 198
 - hard margin classification, 154
 - image classifiers using Sequential APIs, 297-307
 - large margin classification, 153
 - linear SVM classification, 153
 - MNIST dataset, 85
 - multiclass classification, 100
 - multilabel classification, 106
 - multioutput classification, 107
 - multitask classification, 311
 - nonlinear SVM classification, 157-162
 - performance measures, 88-100
 - soft margin classification, 154
 - voting classifiers, 189
- closed-form solution, 114
- cluster specification, 711
- clustering algorithms
 - additional algorithms, 258
 - applications for, 10, 237
 - DBSCAN, 255
 - goal of, 236
 - for image segmentation, 238, 249
 - K-Means, 238-249
 - overview of, 236
 - for preprocessing, 251
 - for semi-supervised learning, 253
- code examples, obtaining and using, xxi
- codings, 567
- Colab Runtime, 693
- Colaboratory (Colab), 693

- collect policy, 649
- color channels, 451
- color segmentation, 249
- column vectors, 113
- comments and questions, xxiii, 718
- complementary slackness, 762
- components, 38
- compression, 224
- computation graphs, 376
- Compute Unified Device Architecture library (CUDA), 690
- concatenative attention, 550
- concrete functions, 791
- conditional probability, 547
- confusion matrix, 90
- connectionism, 280
- constrained optimization, 166
- Contrastive Divergence, 777
- convergence, 118
- convex function, 120
- convolution kernels, 450
- convolutional autoencoders, 579
- convolutional layer
 - filters, 450
 - memory requirements, 456
 - overview of, 448
 - stacking multiple feature maps, 451
 - TensorFlow implementation, 453
- Convolutional Neural Networks (CNNs)
 - architecture of visual cortex, 446
 - classification and localization, 483
 - CNN architectures, 460-478
 - convolutional layer, 448-456
 - object detection, 485-492
 - overview of, 445
 - pooling layer, 456
 - pretrained models for transfer learning, 481
 - pretrained models from Keras, 479
 - ResNet-34 using Keras, 478
 - semantic segmentation, 492
- core instances, 255
- corpus development, 24
- correlation coefficient, 58
- cost functions
 - cross-entropy loss (log loss), 149
 - hinge loss, 155, 173
 - mean absolute error (MAE), 41, 293
 - mean squared error, 120, 293, 308, 384, 570, 573, 583, 636
- role of, 20
- credit assignment problem, 619
- cross-entropy loss (log loss), 149, 295
- cross-validation, 31, 73, 89
- CUDA Deep Neural Network library (cuDNN), 690
- curiosity-based exploration, 664
- curse of dimensionality, 214
- custom models
 - about, 375
 - activation functions, initializers, regularizers, and constraints, 387
 - computing gradients using Autodiff, 399, 765-772
 - layers, 391
 - loss functions, 384
 - losses and metrics, 397
 - metrics, 388
 - models, 394
 - saving and loading, 385
 - training loops, 402
- customer segmentation, 237

D

- data (see also data preparation; data visualization; training data)
 - analyzing through clustering, 237
 - California Housing Prices dataset, 36
 - chopping sequential datasets, 528
 - compressing, 224
 - data mismatch, 32
 - decompressing, 224
 - downloading, 46
 - efficient data representations, 569
 - Fashion MNIST dataset, 297, 574, 590
 - flat datasets, 529
 - geographical data, 56
 - Google News 7B corpus, 541
 - helper function creation, 420
 - importance of over algorithms, 24
 - Internet Movie Database, 534
 - iris dataset, 145
 - loading and preprocessing with TensorFlow, 413-442
 - MNIST dataset, 85
 - nested datasets, 529
 - noisy data, 19
 - prefetching, 421
 - preprocessing, 251, 419, 430-439

- reconstruction error, 224
- reducing dimensionality of, 222
- shuffling, 416
- skewed datasets, 89
- sources for, 35
- splitting sequential datasets, 527
- training dataset creation, 527
- training sparse models, 359
- using datasets with tf.Keras, 423
- Data API (TensorFlow)
 - chaining transformations, 415
 - helper function creation, 420
 - overview of, 414
 - prefetching data, 421
 - preprocessing data, 419
 - shuffling data, 416
 - using datasets with tf.keras, 423
- data augmentation, 464
- data parallelism, 701, 704
- data preparation
 - benefits of functions for, 62
 - custom transformers, 68
 - data cleaning, 63
 - feature scaling, 69
 - handling text and categorical attributes, 65
 - transformation pipelines, 70
- data snooping bias, 51
- data visualization
 - attribute combinations, 61
 - computing correlations, 58
 - dimensionality reduction, 213
 - geographical data, 56
 - test, training, and exploration sets, 56
 - using TensorBoard for, 317
 - visualizing Fashion MNIST Dataset, 574
 - visualizing reconstructions, 574
- datasets, defined, 414
- DataViz (see data visualization)
- DBSCAN (density-based spatial clustering of
 - applications with noise), 255
- decision boundaries, 145
- decision function, 93
- Decision Stumps, 203
- Decision Trees
 - benefits of, 175
 - CART training algorithm, 179
 - computational complexity, 180
 - estimating class probabilities, 178
 - evaluating, 73
 - Gini impurity versus entropy, 180
 - instability drawbacks, 185
 - making predictions, 176
 - regression tasks, 183
 - regularization hyperparameters, 181
 - training and visualizing, 175
- decoders, 501, 569
- decompression, 224
- deep autoencoders, 572
- deep belief networks (DBNs), 13, 777
- deep computer vision (see Convolutional Neural Networks (CNNs))
- deep convolutional GANs, 598
- Deep Learning VM Images, 692
- deep neural networks (DNNs)
 - avoiding overfitting, 364-371
 - default configuration, 371
 - defined, xv, 289
 - faster optimizers, 351-364
 - overview of, 331
 - reusing pretrained layers, 345-351
 - vanishing/exploding gradients problems, 332-345
- Deep Neuroevolution, 323
- Deep Q-Learning
 - Double DQN, 640
 - Dueling DQN, 641
 - fixed Q-Value targets, 639
 - implementing, 634
 - overview of, 633
 - prioritized experience replay, 640
 - variants of, 639
- deep Q-networks (DQNs), 633, 650, 650
- denoising autoencoders, 581
- dense layer, 285
- dense vectors, 556
- density estimation, 236, 264
- depth concat layer, 467
- depth radius, 466
- depthwise separable convolution, 474
- deque, 635
- development sets (dev sets), 31
- differencing, 506
- dimensionality reduction
 - additional techniques, 232
 - approaches for, 215-218
 - using clustering, 237
 - curse of dimensionality, 214
 - goal of, 12

- LLE (Locally Linear Embedding), 230
 - overview of, 213
- PCA (Principal Component Analysis), 219-230
- discount factors, 619
- discriminators, 568
- Distribution Strategies API, 668, 709
- dot product, 551
- Double DQN, 640
- Double Dueling DQN, 642
- DQN agents, 652
- dropout, 365
- dual numbers, 768
- dual problem, 168, 761
- duck typing, 68
- Dueling DQN algorithm, 641
- dummy attributes, 67
- dying ReLUs problem, 335
- dynamic models, 313
- dynamic placer algorithm, 697
- Dynamic Programming, 628

E

- eager execution/eager mode, 408
- early stopping, 141
- Elastic Net, 140
- ELU (exponential linear unit), 336-338
- embedded devices, 685
- Embedded Reber grammars, 566
- embedding, 68, 413, 433
- embedding matrix, 435
- encoders, 501, 569
- Encoder–Decoder model, 501, 542-548
- end-of-sequence (EoS) token, 542, 556
- energy function, 774
- Ensemble Learning
 - bagging and pasting, 192-196
 - benefits of, 74
 - best uses of, 191
 - boosting, 199-208
 - defined, 189
 - examples of, 189
 - Random Forests, 189, 197
 - random patches and random subspaces, 196
 - stacking, 208
 - voting classifiers, 189
- Ensemble methods, 189
- ensembles, 189
- entailment, 564

- entropy impurity measure, 180
- epochs, 125, 290
- equalized learning rates, 603
- equivariance, 458
- error analysis, 102
- estimators, 64
- Euclidean norm, 41
- event files, 317
- evidence lower bound (ELBO), 272
- example project
 - data downloading, 42-55, 756
 - data preparation, 62-72, 757
 - data visualization, 56-62, 756
 - framing the problem, 37, 755
 - launching, monitoring, and maintaining, 80, 760
 - Machine Learning project checklist, 37, 755
 - model fine-tuning, 75-80, 759
 - model selection and training, 72, 758
 - overview of, 35
 - project goals, 37
 - real-world data for, 35
 - selecting performance measure, 39
 - verifying assumptions, 42
- Exclusive OR (XOR) classification problem, 288
- exercise solutions, 719-753
- expectation step, 262
- Expectation-Maximization (EM) algorithm, 262
- experience replay, 597
- explainability, 553
- explained variance ratio, 222
- exploding gradients problem, 332
- exploration policy, 630, 632
- exploration sets, 56
- exponential linear unit (ELU), 336-338
- exponential scheduling, 360
- Extra-Trees classifier, 198
- Extremely Randomized Trees ensemble, 198

F

- F1 score, 92
- fake quantization, 687
- false positive rate (FPR), 97
- fan-in/fan-out numbers, 333
- Fashion MNIST dataset, 297, 574, 590
- Fast-MCD (minimum covariance determinant), 274

- feature engineering, 27
- feature extraction, 12, 27
- feature maps, 228, 450
- feature scaling, 69
- feature selection, 27
- feature space, 226
- feature vector, 113
- features, 8
- feedforward neural networks (FNNs), 289
- filters, 450
- final trained models, 20
- finite difference approximation, 766
- First In, First Out (FIFO) queues, 383
- first-order partial derivatives (Jacobians), 358
- fitness functions, 20
- fixed Q-Value targets, 639
- flat datasets, 529
- folds, 73, 89
- forecasting, 503
- forget gate, 516
- forward pass, 290
- forward-mode autodiff, 767
- fraud detection, 237
- Full Gradient Descent, 122
- fully connected layer, 285
- fully convolutional networks (FCNs), 487
- fully-specified model architecture, 20
- function definitions, 792
- function graphs, 792
- Functional API, 308-313

G

- gate controllers, 516
- Gated Recurrent Unit (GRU) cell, 518
- Gaussian mixture model (GMM)
 - additional algorithms for anomaly and novelty detection, 274
 - anomaly detection using, 266
 - Bayesian Gaussian Mixture models, 270
 - graphical model of, 260
 - overview of, 260
 - selecting cluster number, 267
 - variants, 260
- Gaussian Radial Basis Function (RBF), 159
- generalization error, 30
- generalized Lagrangian, 762
- Generative Adversarial Networks (GANs)
 - versus autoencoders, 568
 - deep convolutional GANs (DCGANs), 598

- difficulties of training, 596
- overview of, 592
- progressive growing of, 601
- StyleGANs, 604
- uses for, 567
- generative autoencoders, 586
- generative models, 263, 567, 775 (see also autoencoders; Generative Adversarial Networks (GANs))
- generative network, 569
- generators, 568
- genetic algorithms, 612
- Gini impurity measure, 180
- global average pooling layer, 460
- global minimum, 119
- Glorot and He initialization, 333
- Google Cloud Platform (GCP)
 - prediction service creation, 677-681
 - prediction service use, 682-685
- Google Cloud Storage (GCS), 679
- Google News 7B corpus, 541
- GoogLeNet, 466
- GPUs (graphics processing units)
 - adding to single machines, 689
 - Colaboratory (Colab), 693
 - GPU-equipped virtual machines, 692
 - managing GPU RAM, 694
 - parallel execution across multiple devices, 699
 - placing operations and variables on devices, 697
 - selecting, 690
 - speeding computations with, 689
- Gradient Boosted Regression Trees (GBRT), 203
- Gradient Boosting, 203
- gradient clipping, 345
- Gradient Descent (GD)
 - Batch Gradient Descent, 121
 - Mini-batch Gradient Descent, 127
 - overview of, 111, 118
 - Stochastic Gradient Descent, 124
- Gradient Tree Boosting, 203
- graph mode, 408
- greedy algorithms, 180
- greedy layer-wise pretraining, 349
- greedy layer-wise training, 578

H

- hard clustering, 240
- hard margin classification, 154
- hard voting classifiers, 190
- harmonic mean, 92
- HDF5 format, 314
- He initialization, 333
- Heaviside step function, 285
- Hebb's rule, 286
- Hebbian learning, 286
- helper functions, 420
- hidden layers
 - in MLPs, 289
 - neurons per hidden layer, 324
 - number of, 323
- hidden units, 775
- hierarchical clustering algorithms, 10
- Hierarchical DBSCAN (HDBSCAN), 258
- high-dimensional training sets, 213
- hinge loss function, 155, 173
- Hinton, Geoffrey, xv
- histograms, 50
- hold outs, 31
- holdout validation, 31
- Hopfield networks, 773
- Huber loss, 293, 384
- Hyperas, 322
- Hyperband, 323
- hyperbolic tangent function (tanh), 291
- Hyperopt, 322
- hyperparameters
 - defined, 29
 - fine-tuning for neural networks, 320-327
 - hyperparameter tuning, 31, 75
 - learning rate, 118
 - Python libraries for optimization, 322
 - regularization hyperparameters, 181
- hyperplanes, 165
- hypothesis boosting, 199

I

- identity matrix, 137
- image classification
 - multitask classification, 311
 - using Sequential API, 297-307
- image generation, 495
- image segmentation, 238, 249
- importance sampling (IS), 640
- impurity, 177, 180

- imputation, 503
- incremental learning, 16
- Incremental PCA (IPCA), 225
- independent and identically distributed (IID), 126
- inequality constraints, 762
- inertia, 243
- inference, 23
- information theory, 180
- initialization
 - centroid initialization methods, 243
 - Glorot and He initialization, 333
 - LeCun initialization, 334
 - random initialization, 118
 - Xavier initialization, 333
- inliers, 266
- input and output sequences, 501
- input gate, 516
- input layers, 289
- input neurons, 285
- input signatures, 791
- instability, 185
- instance segmentation, 249, 495
- instance-based learning, 17, 22
- inter-op thread pool, 699
- intercept terms, 112
- Internet Movie Database, 534
- intra-op thread pool, 699
- invariance, 457
- inverse transformation, 225
- iris dataset, 145
- isolated environments, 43
- Isolation Forest algorithm, 274
- isomap algorithm, 233

J

- JupyterLab, 692
- just-in-time (JIT) compiler, 376

K

- K-fold cross-validation, 73, 89
- K-Means
 - accelerated and mini-batch, 244
 - centroid initialization methods, 243
 - hard and soft clustering, 240
 - image segmentation, 249
 - K-Means algorithm, 241
 - limits of, 248
 - optimal cluster number, 245

- overview of, 238
- preprocessing with, 251
- proposed improvement to, 243
- scaling input features, 249
- for semi-supervised learning, 253
- k-Nearest Neighbors regression, 22
- Karush–Kuhn–Tucker (KKT) multipliers, 762
- keep probability, 367
- Keras
 - benefits of, xvi
 - complex architectures, 314
 - gradient clipping in, 345
 - implementing Batch Normalization with, 341
 - implementing dropout using, 367
 - implementing MLPs with, 295-320
 - implementing ResNet-34 with, 478
 - keras.callbacks package, 316
 - loading datasets with, 297
 - low-level API, 381
 - multibackend Keras, 295
 - preprocessing layers, 437
 - saving and restoring models in, 314
 - stacked autoencoders using, 572
 - transfer learning with, 347
 - using code examples from keras.io, 300
 - using pretrained models from, 479
- Keras Tuner, 322
- Kernel PCA (kPCA), 226-230
- kernel trick, 158, 228
- kernelized SVM, 169
- kernels, 170, 226, 377
- kopt library, 322
- Kullback–Leibler divergence, 150

L

- label propagation, 254
- labels, 8, 39, 239
- Lagrange multipliers, 761
- landmarks, 159
- language models, 563 (see also natural language processing (NLP))
- large margin classification, 153
- Lasso Regression, 137
- latent loss, 587
- latent representations, 567
- latent variables, 262
- law of large numbers, 191
- Layer Normalization, 512

- layers
 - 1D convolutional layer, 520
 - adaptive instance normalization (AdaIN), 604
 - bidirectional recurrent layer, 546
 - convolutional layer, 448-456
 - dense (fully connected) layer, 285
 - hidden layer, 289
 - input layer, 289
 - Masked Multi-Head Attention layer, 556
 - minibatch standard deviation layer, 603
 - Multi-Head Attention layer, 556, 559
 - output layer, 289
 - pooling layer, 456
 - recurrent, 498-502
 - reusing pretrained, 345-351
 - Scaled Dot-Product Attention layer, 559
- leaf nodes, 176
- leaky ReLU function, 335
- learning curves, 130-134
- learning rate, 16, 118, 325, 603
- learning rate scheduling, 359
- learning schedules, 125, 360
- LeCun initialization, 334
- LeNet-5, 463
- Levenshtein distance, 161
- liblinear library, 162
- libsvm library, 162
- likelihood function, 267
- linear algebra, 112
- linear autoencoders, 570
- Linear Discriminant Analysis (LDA), 233
- linear models, 19
- Linear Regression model
 - approaches to training, 111, 113
 - computational complexity, 117
 - Normal Equation, 114
 - overview of, 112
- linear SVM classification, 153
- lists of lists, using SequenceExample Protobuf, 429
- LLE (Locally Linear Embedding), 230
- Lloyd-Forgy algorithm, 238
- local minimum, 119
- Local Outlier Factor (LOF), 274
- local response normalization, 465
- localization, 483
- log loss, 144
- log-odds, 144

- logical computations, 283
- logical GPU devices, 695
- Logistic (sigmoid) function, 143, 293-294, 302, 332
- Logistic Regression
 - classification with, 8
 - decision boundaries, 145
 - estimating probabilities, 143
 - overview of, 142
 - Softmax Regression, 148
 - training and cost function, 144
- logit, 144
- Logit Regression (see Logistic Regression)
- long sequences
 - overview of, 511
 - short-term memory problems, 514-523
 - unstable gradients problem, 512
- Long Short-Term Memory (LSTM) cell, 514
- loss functions (see cost functions)
- Luong attention, 551

M

- Machine Learning (ML)
 - additional resources, xix
 - applications for, xv, 5
 - approach to learning, xvi
 - benefits of, 2
 - challenges of, 23-30
 - defined, 1
 - history of, xv
 - locating papers on, 378
 - notations for, 40, 164
 - overview of, 30
 - prerequisites to learning, xvii
 - testing and validating, 30-33
 - topics covered, xvii
 - types of, 7-23
- Machine Learning project checklist, 37, 755
- majority-vote classifiers, 190
- majority-vote predictions, 187
- Manhattan norm, 41
- manifold assumption, 218
- manifold hypothesis, 218
- Manifold Learning, 218
- manual differentiation, 765
- margin violations, 155
- Markov chains, 625
- Markov Decision Processes (MDP), 625-629
- Mask R-CNN, 495
- mask tensors, 539
- masked language model (MLM), 564
- Masked Multi-Head Attention layer, 556
- masking, 538
- max pooling layer, 457
- max-norm regularization, 370
- maximization step, 262
- maximum a-posteriori (MAP) estimation, 269
- maximum likelihood estimate (MLE), 269
- mean absolute error (MAE), 41
- mean Average Precision (mAP), 491
- mean coding, 586
- mean field variational inference, 273
- Mean-Shift algorithm, 259
- measure of similarity, 18
- memory bandwidth, 422
- memory cells, 500
- Mercer's conditions, 171
- Mercer's theorem, 171
- meta learners, 208
- metagraphs, 671
- metrics
 - accuracy, 388
 - area under the curve (AUC), 98
 - confusion matrix, 90, 90
 - F1 score, 92
 - mean absolute error (MAE), 41, 293
 - mean average precision, 491
 - mean squared error, 183, 505
 - precision, 91-97
 - recall, 91-97
 - RMSE, 39
 - ROC curve, 97
- Microsoft Cognitive Toolkit (CNTK), 295
- min-max scaling, 69
- Mini-batch Gradient Descent, 127
- mini-batch K-Means, 244
- mini-batches, 15, 127
- minibatch discrimination, 597
- minibatch standard deviation layer, 603
- mirrored strategy, 704
- mixing regularization, 606
- ML Engine, 680
- MNIST dataset, 85
- mobile devices, 685
- mode collapse, 597
- model parallelism, 701
- model parameters, 20
- model selection, 19, 31, 72

- model-based learning, 18
- models (see also custom models)
 - causal models, 510
 - complex using Functional API, 308-313
 - custom with TensorFlow, 384-405
 - defined, 20
 - dynamic using Subclassing API, 313
 - fine-tuning, 75-80
 - parametric versus nonparametric, 181
 - pretrained models for transfer learning, 481
 - pretrained models from Keras, 479
 - saving and restoring, 314
 - sequence-to-sequence models, 510
 - training, 20, 72 (see also training models)
 - training across multiple devices, 701-717
 - training sparse models, 359
 - using callbacks, 315
 - using TensorBoard for visualization, 317
 - white versus black box, 178
- modules, 540
- momentum optimization, 351
- momentum vector, 352
- Monte Carlo (MC) dropout, 368
- Multi-Head Attention layer, 556, 559
- multibackend Keras, 295
- multiclass classification, 100
- Multidimensional Scaling (MDS), 232
- multilabel classification, 106
- Multilayer Perceptrons (MLPs)
 - backpropagation and, 289-292
 - classification MLPs, 294
 - regression MLPs, 292
- multinomial classifiers, 100
- Multinomial Logistic Regression, 148
- multioutput classification, 107
- multiple outputs, 311
- multiple regression problems, 39
- multiplicative attention, 551
- multitask classification, 311
- multivariate regression problems, 39
- multivariate time series, 503

N

- naive forecasting, 505
- Nash equilibrium, 596
- natural language processing (NLP)
 - attention mechanisms, 549-563
 - CNNs for, 445
 - Encoder-Decoder network for, 542-548
 - generating text using character RNNs, 526-534
 - overview of, 525
 - recent innovations in, 563
 - RNNs for, 497
 - sentiment analysis, 534-542
 - uses for, 351
- nested datasets, 529
- Nesterov Accelerated Gradient (NAG), 353
- Nesterov momentum optimization, 353
- neural machine translation (NMT), 542-563
 - (see also natural language processing (NLP))
- neurons
 - bias neurons, 285
 - fan-in/fan-out numbers, 333
 - from biological to artificial, 280-295
 - input neurons, 285
 - logical computations with, 283
 - per hidden layer, 324
 - recurrent neurons, 498-502
 - stochastic neurons, 775
- Newton's difference quotient, 766
- next sentence prediction (NSP), 565
- No Free Lunch (NFL) theorem, 33
- noisy data, 19
- non-max suppression, 486
- nonlinear dimensionality reduction (NLDR), 230
- nonlinear SVM classification, 157-162
- nonparametric models, 181
- nonsaturating activation functions, 335
- nonsequential neural networks, 308
- Normal Equation, 114
- normalization, 69, 339, 603
- normalized exponential, 148
- novelty detection, 12, 267, 274
- NP-Complete problem, 180
- null hypothesis, 182
- NumPy
 - array_split() function, 226
 - dense arrays, 67
 - installing, 42
 - inv() function, 115
 - memmap class, 226
 - randint() function, 107
 - serializing large arrays, 75
 - svd() function, 221
 - using TensorFlow like, 379-384

NVIDIA Collective Communications Library (NCCL), 710
Nvidia GPU cards, 690

O

object detection
 fully convolutional networks (FCNs), 487
 overview of, 485
 You Only Look Once (YOLO), 489
objectness output, 486
observed variables, 262
observers, 654
off-policy algorithms, 632
offline learning, 15
on-policy algorithms, 632
one-class SVM algorithm, 275
one-hot encoding, 67
one-hot vectors, 431
one-versus-all (OvA) strategy, 100
one-versus-one (OvO) strategy, 100
one-versus-the-rest (OvR) strategy, 100
online learning, 15, 88
online model, 639
online SVMs, 172
OpenAI Gym, 613-617
Optical Character Recognition (OCR), 1
optimal state value, 627
optimizers
 AdaGrad, 354
 Adam and Nadam optimization, 356
 creating faster, 351
 first- and second-order partial derivatives, 358
 learning rate scheduling, 359
 momentum optimization, 351
 Nesterov Accelerated Gradient (NAG), 353
 RMSProp, 355
 Stochastic Gradient Descent (SGD), 88, 124
original space, 226
out-of-core learning, 16
out-of-sample error, 30
out-of-vocabulary (oov) buckets, 432
outlier detection, 237, 266
output gate, 516
output layers, 289
overcomplete autoencoders, 580, 580
overfitting
 avoiding through regularization, 364-371
 defined, 27

limiting risk of, 457

P

p (posterior) distribution, 272
p (prior) distribution, 271
p-value, 182
parameter efficiency, 323
parameter matrix, 148
parameter servers, 705
parameter space, 121
parameter vector, 113
parametric leaky ReLU (PReLU), 335
parametric models, 181
partial derivatives, 121
pasting (see bagging and pasting)
pattern matching, 569
PCA (Principal Component Analysis)
 anomaly and novelty detection using, 274
 choosing dimension number, 223
 for compression, 224
 explained variance ratio, 222
 incremental, 225
 Kernel PCA (kPCA), 226-230
 overview of, 219
 preserving variance, 219
 principal component axis, 220
 projecting down to d dimensions, 221
 randomized, 225
 using Scikit-Learn, 222
 undercomplete linear autoencoders for, 570
Pearson's r, 58
peephole connections, 518
penalties, 14
Perceptron, 284-288
Perceptron convergence theorem, 287
performance measures (see metrics)
performance scheduling, 361
piecewise constant scheduling, 361
pipelines, 38, 424
pixelwise normalization layers, 603
policies, 14, 612
policy gradients (PG), 613, 620-625
policy parameters, 612
policy search, 612
policy space, 612
polynomial features, 158
polynomial kernels, 170
Polynomial Regression, 112, 128
pooling kernel, 457

- pooling layer, 456
- positional embeddings, 556
- post-training quantization, 686
- power scheduling, 360
- pre-images, 228
- precision, 91-97
- prediction problems, 8, 17, 189
- prediction service
 - creating on GCP AI, 677-681
 - using, 682-685
- predictors, 65
- preprocessing, 251, 430-439
- pretraining
 - for transfer learning, 481
 - greedy layer-wise pretraining, 349
 - models from Keras, 479
 - on auxiliary tasks, 350
 - reusing pretrained embeddings, 540
 - reusing pretrained layers, 345-351
 - unsupervised pretraining, 349
 - using stacked autoencoders, 576-579
- primal problem, 168
- prioritized experience replay (PER), 640
- probabilistic autoencoders, 586
- probability density function (PDF), 236, 264
- projection, 215
- propositional logic, 280
- protocol buffers (protobufs), 425
- Proximal Policy Optimization (PPO), 663
- pruning, 182
- PyTorch library, 296

Q

- Q-Learning
 - Approximate Q-Learning and Deep Q-Learning, 633
 - exploration policy, 632
 - implementing, 631
 - overview of, 630
- Q-Value Iteration, 628
- Q-Values, 628
- Quadratic Programming (QP) problems, 167
- quantization-aware training, 687
- queries per second (QPS), 667
- questions and comments, xxiii, 718
- queues, 383, 788

R

- Radial Basis Function (RBF), 159

- ragged tensors, 383, 784
- Rainbow agent, 642
- Random Forests
 - benefits of, 189
 - Extra-Trees, 198
 - feature importance, 198
 - overview of, 197
- random initialization, 118
- random patches and random subspaces, 196
- random projections, 232
- randomized leaky ReLU (RReLU), 335
- Randomized PCA, 225
- recall, 91-97
- receiver operating characteristic (ROC) curve, 97
- recognition network, 569
- recommender systems, 237
- reconstruction error, 224
- reconstruction loss, 397, 570
- reconstruction pre-images, 228
- reconstructions, 570
- Rectified Linear Unit function (ReLU), 292-293
- recurrent autoencoders, 580
- recurrent neural networks (RNNs)
 - bidirectional RNNs, 546
 - forecasting time series, 503-511
 - generating text using character RNNs, 526-534
 - handling long sequences, 511-523
 - overview of, 497
 - recurrent neurons and layers, 498-502
 - stateless and stateful, 525, 532
 - training, 502
- recurrent neurons, 498
- Region Proposal Network (RPN), 492
- regression problems
 - Decision Trees, 183
 - defined, 8
 - k-Nearest Neighbors regression, 22
 - Lasso Regression, 137
 - Linear Regression, 112-117
 - Logistic Regression, 142-151
 - multiple regression problems, 39
 - multivariate regression problems, 39
 - Polynomial Regression, 128
 - regression MLPs, 292
 - regression MLPs using Sequential API, 307
 - Ridge Regression, 135
 - Softmax Regression, 148-151

- SVM regression, 162
 - univariate regression problems, 39
- regular expressions, 536
- regularization
 - avoiding overfitting through, 364-371
 - defined, 28
 - hyperparameters for Decision Trees, 181
 - multiple outputs for, 311
 - shrinkage technique, 205
- regularization terms, 135
- regularized linear models
 - Elastic Net, 140
 - Lasso Regression, 137
 - overview of, 134
 - Ridge Regression, 135
- REINFORCE algorithms, 620
- Reinforcement Learning (RL)
 - algorithms for, 662
 - Deep Q-Learning, 633-638
 - evaluating actions, 619
 - Markov Decision Processes (MDP), 625-629
 - neural network policies, 617
 - OpenAI Gym, 613-617
 - optimizing rewards, 610
 - overview of, 14, 609
 - policy gradients, 620-625
 - policy search, 612
 - Q-Learning, 630-634
 - Temporal Difference Learning, 629
 - TF-Agents library, 642-662
- ReLU (Rectified Linear Unit function), 292-293
- replay buffers, 635, 649, 654
- replay memory, 635
- representation learning, 68, 434 (see also autoencoders)
- residual blocks, 395
- residual errors, 203
- residual learning, 471
- residual units, 471
- ResNet (Residual Network), 471
- ResNet-34 CNN, 478
- responsibilities (clustering), 262
- restoring models, 314
- restricted Boltzmann machines (RBMs), 13, 349, 776
- reverse-mode autodiff, 290, 770
- rewards, 14
- Ridge Regression, 135
- RMSProp, 355

- Root Mean Square Error (RMSE), 39, 120
- root nodes, 176

S

- SAMME (Stagewise Additive Modeling using a Multiclass Exponential loss function), 203
- sample inefficiency, 625
- sampled softmax technique, 544
- sampling bias, 25
- sampling noise, 25
- SavedModel format, 669
- saving and restoring models, 314
- Scaled Dot-Product Attention layer, 559
- Scaled Exponential Linear Unit (SELU) function, 334, 337-338, 368
- Scikit-Learn
 - AdaBoost version used in, 203
 - anomaly and novelty detection, 274
 - automatic reconstruction with, 229
 - bagging and pasting in, 194
 - benefits of, xvi
 - CART training algorithm, 177, 179
 - clustering algorithms in, 258
 - computing classifier metrics, 92-107
 - converting text to numbers, 66
 - cross_val_score() function, 89
 - data centering in, 221
 - dataset dictionary structure, 85
 - DecisionTreeRegressor class, 183
 - design principles, 64
 - dimensionality reduction in, 232
 - ExtraTreesClassifier class, 198
 - feature importance scoring, 198
 - feature scaling, 154
 - full SVD approach, 225
 - GBRT ensemble training in, 204
 - GridSearchCV, 76
 - incremental training in, 207
 - IncrementalPCA class, 226
 - installing, 42
 - K-fold cross-validation feature, 73
 - KernelPCA class, 227
 - launching, monitoring, and maintaining your system, 80
 - linear model using, 21
 - linear regression using, 116
 - LLE (Locally Linear Embedding), 230, 232
 - max_depth hyperparameter, 181
 - mean_squared_error function, 72

- missing value handling, 63
- one-hot vectors, 67
- out-of-bag evaluation, 195
- PCA using, 222
- Perceptron class, 287
- presorting data with, 180
- Randomized PCA algorithm, 225
- random_state hyperparameter, 185
- saving models, 75
- SGDClassifier class, 88
- splitting datasets into subsets, 53
- stratified sampling using, 54
- SVM classification classes, 162
- SVM models, 155
- tolerance hyperparameter, 162
- transformation sequences, 70
- transformers and, 68
- voting classifiers in, 191
- Scikit-Optimize, 322
- SE block, 476
- SE-Inception, 476
- SE-ResNet, 476
- search engines, 238
- second-order partial derivatives (Hessians), 358
- self-attention mechanism, 556
- self-normalization, 337
- self-organizing maps (SOMs), 780
- self-supervised learning, 351
- SELU (Scaled Exponential Linear Unit) function (see Scaled Exponential Linear Unit (SELU) function)
- semantic interpolation, 590
- semantic segmentation, 249, 458, 492
- semi-supervised learning
 - clustering algorithms for, 237, 253
 - defined, 13
 - examples of, 13
- SENet (Squeeze-and-Excitation Network), 476
- sensitivity, 91
- sentence encoders, 541
- sentiment analysis
 - defined, 526
 - masking, 538
 - overview of, 534
 - reusing pretrained embeddings, 540
- separable convolution, 474
- sequence-to-sequence models, 510
- sequence-to-vector networks, 501
- SequenceExample protobuf (TensorFlow), 429
- sequences
 - forecasting time series, 503-511
 - handling long, 511-523
 - input and output, 501
 - RNNs for, 497
- Sequential API
 - image classifiers using, 297-307
 - regression MLP using, 307
- service account, 682
- sets, 383, 787
- Shannon's information theory, 180
- short-term memory problems, 514-523
- shortcut connections, 471
- shrinkage, 205
- shuffling-buffer approach, 417
- sigmoid (Logistic) activation function, 143, 293-294, 302, 332
- sigmoid kernel, 171
- silhouette coefficient, 246
- silhouette diagram, 247
- silhouette score, 246
- similarity functions, 159
- simulated annealing, 125
- simulated environments, 614
- single-shot learning, 495
- Singular Value Decomposition (SVD), 117, 221
- skewed datasets, 89
- skip connections, 337, 471
- Sklearn-Deap, 323
- slack variables, 167
- smoothing term, 340
- Soft Actor-Critic algorithm, 663
- soft clustering, 240
- soft margin classification, 154
- soft voting, 192
- softmax function, 148, 294, 299, 470, 482, 488, 543
- Softmax Regression, 148
- softplus activation function, 293
- spam filters, 1, 2
- spare replicas, 706
- sparse autoencoders, 582
- sparse matrix, 67
- sparse models, 359
- sparse tensors, 383, 785
- sparsity, 582
- sparsity loss, 583
- Spearmint library, 322
- spectral clustering, 259

- spurious patterns, 774
- stacked autoencoders
 - overview of, 572
 - stacked denoising autoencoders, 581
 - unsupervised pretraining using, 576-579
 - using Keras, 572
 - visualizing Fashion MNIST Dataset, 574
 - visualizing reconstructions, 574
- stacked denoising autoencoders, 581
- stacked generalization, 208
- stacking, 208
- stale gradients, 707
- standard correlation coefficient, 58
- standardization, 69
- start of sequence (SoS) token, 535
- state-action values, 628
- stateful metrics, 389
- stationary point, 761
- statistical mode, 193
- statistical significance, 182
- step function, 284
- Stochastic Gradient Boosting, 207
- Stochastic Gradient Descent (SGD), 88, 124
- stochastic neurons, 775
- stochastic policy, 612
- stratified sampling, 53
- streaming metrics, 389
- stride, 449
- string kernels, 161
- string subsequence kernel, 161
- string tensors, 383, 783
- strong learners, 190
- style mixing, 606
- style transfer, 604
- StyleGANs, 567, 604
- Subclassing API, 313
- subderivatives, 173
- subgradient vector, 140
- subsampling, 456
- subspace, 215
- summaries (TensorFlow), 317
- supervised learning
 - algorithms covered, 9
 - common tasks, 8
 - defined, 8
- Support Vector Machines (SVMs)
 - benefits of, 153
 - decision function and prediction, 165
 - dual problem, 168, 761
 - kernelized SVM, 169
 - linear SVM classification, 153
 - nonlinear SVM classification, 157-162
 - online SVMs, 172
 - SVM regression, 162
 - training objective, 166
- support vectors, 154
- symbolic differentiation, 768
- symbolic tensors, 408, 792
- symmetry, breaking in backpropagation, 291
- synchronous updates, 706

T

- t-Distributed Stochastic Neighbor Embedding (t-SNE), 233
- tail-heavy histograms, 51
- Talos library, 322
- target model, 639
- TD error, 630
- TD target, 630
- temperature
 - in Boltzmann machines, 775
 - in text generation, 531
- Temporal Difference Learning (TD Learning), 629
- tensor arrays, 383, 786
- TensorBoard, 317
- TensorFlow Addons, 545
- TensorFlow cluster, 711
- TensorFlow Extended (TFX), 440
- TensorFlow Hub, 378, 540
- TensorFlow Lite, 378
- TensorFlow Model Optimization Toolkit (TF-MOT), 359
- TensorFlow Playground, 295
- TensorFlow, basics of
 - architecture, 377
 - benefits, xvi, 376
 - community support, 379
 - features, 376
 - getting help, 379
 - installing, 296
 - library ecosystem, 378
 - operating system compatibility, 378
 - PyTorch library and, 296
 - versions covered, 375
- TensorFlow, CNNs
 - convolution operations, 494
 - convolutional layers, 453

- pooling layer, 458
- TensorFlow, custom models and training
 - about, 375
 - activation functions, initializers, regularizers, and constraints, 387
 - computing gradients using Autodiff, 399, 765-772
 - implementing learning rate scheduling, 363
 - layers, 391
 - loss functions, 384
 - losses and metrics, 397
 - metrics, 388
 - models, 394
 - saving and loading, 385
 - special data structures, 783-789
 - training loops, 402
- TensorFlow, data loading and preprocessing
 - Data API, 414-424
 - overview of, 413
 - preprocessing input features, 430-439
 - TensorFlow Datasets (TFDS) Project, 441, 441
 - TF Transform, 439
 - TFRecord format, 424-430
- TensorFlow, functions and graphs
 - AutoGraph and tracing, 407, 791-799
 - overview of, 405
 - TF Function rules, 409
- TensorFlow, model deployment at scale
 - deploying on AI platforms, 81
 - deploying to mobile and embedded devices, 685-688
 - overview of, 667
 - serving TensorFlow models, 668-685
 - training models across multiple devices, 701-717
 - using GPUs to speed computations, 689-701
- TensorFlow, NumPy-like operations
 - other data structures, 383
 - tensors and NumPy, 381
 - tensors and operations, 379
 - type conversions, 381
 - variables, 382
- TensorFlow.js, 378
- tensors, 379
- Term-Frequency \times Inverse-Document-Frequency (TF-IDF), 439
- terminal state, 626
- test sets, 30, 51
- testing and validation
 - data mismatch, 32
 - hyperparameter tuning, 31
 - model selection, 31
- text generation
 - building and training models for, 530
 - chopping sequential datasets, 528
 - generating Shakespearean text, 531
 - overview of, 526
 - splitting sequential datasets, 527
 - stateful RNNs and, 532
 - training dataset creation, 527
 - using models for, 531
- TF Datasets (TFDS), 414, 441
- TF Functions
 - graphs generated by, 791-799
 - rules, 409
- TF Transform (tf.Transform), 414, 439
- TF-Agents library
 - collect driver, 656
 - datasets, 658
 - deep Q-networks (DQNs), 650
 - DQN agents, 652
 - environment specifications, 644
 - environment wrappers, 645
 - environments, 643
 - installing, 643
 - overview of, 642
 - replay buffer and observer, 654
 - training architecture, 649
 - training loops, 661
 - training metrics, 655
- tf.keras, 295, 363, 363, 423
- tf.summary package, 319
- TF.Text library, 536
- TFRecord format
 - compressed TFRecord files, 425
 - lists of lists using SequenceExample Protocol buffer, 429
 - loading and parsing examples, 428
 - overview of, 424
 - protocol buffers (protobufs), 425
 - TensorFlow protobufs, 427
- Theano, 295
- theoretical information criterion, 267
- thermal equilibrium, 775
- threshold logic unit (TLU), 284
- Tikhonov regularization, 135
- time series data

- additional models for, 506
- baseline metrics, 505
- deep RNNs, 506
- forecasting several steps ahead, 508
- overview of, 503
- RNNs for, 497
- simple RNNs, 505
- time step, 498
- tokenization, 536
- tolerance, 123
- TPUs (tensor processing units), 377
- train-dev sets, 32
- training data
 - defined, 2
 - hold outs, 31
 - insufficient quantity of, 23
 - irrelevant features, 27
 - nonrepresentative, 25
 - overfitting, 27
 - poor quality, 26
 - training dataset creation, 527
 - underfitting, 29
- training instances, 2, 215
- training models
 - defined, 20
 - example project, 72
 - Gradient Descent, 118-128
 - learning curves, 130-134
 - Linear Regression, 112-117
 - Logistic Regression, 142-151
 - overview of, 111
 - Polynomial Regression, 128-130
 - regularized linear models, 134-142
- training samples, 2
- training set rotation, 185
- training sets, 2, 30, 213
- training/serving skew, 440
- trajectories, 649
- trajectory, 650
- transfer learning, 324, 345, 481
- transformations
 - affine transformations, 604
 - chaining, 415
 - custom, 68
 - inverse transformation, 225
 - purpose of, 64
 - transformation pipelines, 70
- Transformer architecture, 554
- transposed convolutional layer, 493
- true negative rate (TNR), 97
- true positive rate (TPR), 91
- truncated backpropagation through time, 529
- Turing test, 525
- tying weights, 577
- type conversions, 381

U

- uncertainty sampling, 255
- undercomplete autoencoders, 570
- underfitting, 29
- undiscounted rewards, 656
- univariate regression problems, 39
- univariate time series, 503
- unrolling the network through time, 498
- unstable gradients problem, 512
- unsupervised learning
 - algorithms covered, 10
 - clustering, 236-260
 - common tasks, 10
 - defined, 9
 - Gaussian mixtures model (GMM), 260-275
 - overview of, 235
 - pretraining using stacked autoencoders, 576-579
- unsupervised pretraining, 349
- upsampling layer, 493
- utility functions, 20

V

- validation sets, 31
- Value Iteration algorithm, 627
- vanishing/exploding gradients problems, 332-345
- variables, 382
- variance
 - explained variance ratio, 222
 - preserving, 219
- variational autoencoders, 586-591
- variational inference, 272
- variational parameters, 272
- vector-to-sequence networks, 501
- vectors
 - column vectors, 113
 - feature vectors, 113
 - momentum vector, 352
 - parameter vectors, 113
 - subgradient vectors, 140
- VGGNet, 470

- virtual GPU devices, 695
- visible units, 775
- visual attention, 552
- visualization algorithms, 11
- vocabulary, 432
- voice recognition, 445

W

- wall time, 341
- warmup phase, 708
- WaveNet, 498, 521
- weak learners, 190
- weighted moving average model, 506
- white box models, 178
- Wide & Deep neural networks, 308
- wisdom of the crowd, 189
- word embeddings, 434

- word tokenization, 536
- WordTrees, 490
- workspace creation, 42

X

- Xavier initialization, 333
- Xception (Extreme Inception), 474
- XGBoost, 208

Y

- You Only Look Once (YOLO), 489

Z

- zero padding, 449
- zero-shot learning (ZSL), 564
- ZF Net, 466

About the Author

Aurélien Géron is a Machine Learning consultant and lecturer. A former Googler, he led YouTube's video classification team from 2013 to 2016. He's been a founder of and CTO at a few different companies: Wifirst, a leading wireless ISP in France; Polyconseil, a consulting firm focused on telecoms, media, and strategy; and Kiwisoft, a consulting firm focused on Machine Learning and data privacy.

Before all that he worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He also published a few technical books (on C++, WiFi, and internet architectures) and lectured about computer science at a French engineering school.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1,023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.

Colophon

The animal on the cover of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* is the fire salamander (*Salamandra salamandra*), an amphibian found across most of Europe. Its black, glossy skin features large yellow spots on the head and back, signaling the presence of alkaloid toxins. This is a possible source of this amphibian's common name: contact with these toxins (which they can also spray short distances) causes convulsions and hyperventilation. Either the painful poisons or the moistness of the salamander's skin (or both) led to a misguided belief that these creatures not only could survive being placed in fire but could extinguish it as well.

Fire salamanders live in shaded forests, hiding in moist crevices and under logs near the pools or other freshwater bodies that facilitate their breeding. Though they spend most of their lives on land, they give birth to their young in water. They subsist mostly on a diet of insects, spiders, slugs, and worms. Fire salamanders can grow up to a foot in length, and in captivity may live as long as 50 years.

The fire salamander's numbers have been reduced by destruction of their forest habitat and capture for the pet trade, but the greatest threat they face is the susceptibility of their moisture-permeable skin to pollutants and microbes. Since 2014, they have become extinct in parts of the Netherlands and Belgium due to an introduced fungus.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. The cover illustration is by Karen Montgomery, based on an engraving from *Wood's Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

The O'Reilly logo is displayed in white, bold, sans-serif capital letters. The background of the entire advertisement is a vibrant red-to-orange gradient, overlaid with several large, semi-transparent, overlapping circles in varying shades of red and orange, creating a dynamic, abstract pattern.

O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning