

`String()` are not TensorFlow operations (and neither are the other operations in this code), so they cannot be included in a TensorFlow Function (except by wrapping them in a `tf.py_function()` operation, which would make the code slower and less portable, as we saw in [Chapter 12](#)). Fortunately, TensorFlow does include special protobuf definitions for which it provides parsing operations.

TensorFlow Protobufs

The main protobuf typically used in a TFRecord file is the `Example` protobuf, which represents one instance in a dataset. It contains a list of named features, where each feature can either be a list of byte strings, a list of floats, or a list of integers. Here is the protobuf definition:

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

The definitions of `BytesList`, `FloatList`, and `Int64List` are straightforward enough. Note that `[packed = true]` is used for repeated numerical fields, for a more efficient encoding. A `Feature` contains either a `BytesList`, a `FloatList`, or an `Int64List`. A `Features` (with an s) contains a dictionary that maps a feature name to the corresponding feature value. And finally, an `Example` contains only a `Features` object.⁸ Here is how you could create a `tf.train.Example` representing the same person as earlier and write it to a TFRecord file:

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
```

⁸ Why was `Example` even defined, since it contains no more than a `Features` object? Well, TensorFlow's developers may one day decide to add more fields to it. As long as the new `Example` definition still contains the `features` field, with the same ID, it will be backward compatible. This extensibility is one of the great features of protobufs.

```
        "id": Feature(int64_list=Int64List(value=[123])),  
        "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",  
                                         b"c@d.com"])))  
    }))
```

The code is a bit verbose and repetitive, but it's rather straightforward (and you could easily wrap it inside a small helper function). Now that we have an `Example` protobuf, we can serialize it by calling its `SerializeToString()` method, then write the resulting data to a TFRecord file:

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:  
    f.write(person_example.SerializeToString())
```

Normally you would write much more than one `Example`! Typically, you would create a conversion script that reads from your current format (say, CSV files), creates an `Example` protobuf for each instance, serializes them, and saves them to several TFRecord files, ideally shuffling them in the process. This requires a bit of work, so once again make sure it is really necessary (perhaps your pipeline works fine with CSV files).

Now that we have a nice TFRecord file containing a serialized `Example`, let's try to load it.

Loading and Parsing Examples

To load the serialized `Example` protobufs, we will use a `tf.data.TFRecordDataset` once again, and we will parse each `Example` using `tf.io.parse_single_example()`. This is a TensorFlow operation, so it can be included in a TF Function. It requires at least two arguments: a string scalar tensor containing the serialized data, and a description of each feature. The description is a dictionary that maps each feature name to either a `tf.io.FixedLenFeature` descriptor indicating the feature's shape, type, and default value, or a `tf.io.VarLenFeature` descriptor indicating only the type (if the length of the feature's list may vary, such as for the "`emails`" feature).

The following code defines a description dictionary, then it iterates over the TFRecord Dataset and parses the serialized `Example` protobuf this dataset contains:

```
feature_description = {  
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),  
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),  
    "emails": tf.io.VarLenFeature(tf.string),  
}  
  
for serialized_example in tf.data.TFRecordDataset(["my_contacts.tfrecord"]):  
    parsed_example = tf.io.parse_single_example(serialized_example,  
                                              feature_description)
```

The fixed-length features are parsed as regular tensors, but the variable-length features are parsed as sparse tensors. You can convert a sparse tensor to a dense tensor using `tf.sparse.to_dense()`, but in this case it is simpler to just access its values:

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
```

A `BytesList` can contain any binary data you want, including any serialized object. For example, you can use `tf.io.encode_jpeg()` to encode an image using the JPEG format and put this binary data in a `BytesList`. Later, when your code reads the TFRecord, it will start by parsing the `Example`, then it will need to call `tf.io.decode_jpeg()` to parse the data and get the original image (or you can use `tf.io.decode_image()`, which can decode any BMP, GIF, JPEG, or PNG image). You can also store any tensor you want in a `BytesList` by serializing the tensor using `tf.io.serialize_tensor()` then putting the resulting byte string in a `BytesList` feature. Later, when you parse the TFRecord, you can parse this data using `tf.io.parse_tensor()`.

Instead of parsing examples one by one using `tf.io.parse_single_example()`, you may want to parse them batch by batch using `tf.io.parse_example()`:

```
dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                           feature_description)
```

As you can see, the `Example` protobuf will probably be sufficient for most use cases. However, it may be a bit cumbersome to use when you are dealing with lists of lists. For example, suppose you want to classify text documents. Each document may be represented as a list of sentences, where each sentence is represented as a list of words. And perhaps each document also has a list of comments, where each comment is represented as a list of words. There may be some contextual data too, such as the document's author, title, and publication date. TensorFlow's `SequenceExample` protobuf is designed for such use cases.

Handling Lists of Lists Using the `SequenceExample` Protobuf

Here is the definition of the `SequenceExample` protobuf:

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

A `SequenceExample` contains a `Features` object for the contextual data and a `FeatureLists` object that contains one or more named `FeatureList` objects (e.g., a `FeatureList` named "content" and another named "comments"). Each `FeatureList` contains a list of `Feature` objects, each of which may be a list of byte strings, a list of 64-bit integers, or a list of floats (in this example, each `Feature` would represent a sentence or a comment, perhaps in the form of a list of word identifiers). Building a `SequenceExample`, serializing it, and parsing it is similar to building, serializing, and parsing an `Example`, but you must use `tf.io.parse_single_sequence_example()` to parse a single `SequenceExample` or `tf.io.parse_sequence_example()` to parse a batch. Both functions return a tuple containing the context features (as a dictionary) and the feature lists (also as a dictionary). If the feature lists contain sequences of varying sizes (as in the preceding example), you may want to convert them to ragged tensors, using `tf.RaggedTensor.from_sparse()` (see the notebook for the full code):

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(  
    serialized_sequence_example, context_feature_descriptions,  
    sequence_feature_descriptions)  
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

Now that you know how to efficiently store, load, and parse data, the next step is to prepare it so that it can be fed to a neural network.

Preprocessing the Input Features

Preparing your data for a neural network requires converting all features into numerical features, generally normalizing them, and more. In particular, if your data contains categorical features or text features, they need to be converted to numbers. This can be done ahead of time when preparing your data files, using any tool you like (e.g., NumPy, pandas, or Scikit-Learn). Alternatively, you can preprocess your data on the fly when loading it with the Data API (e.g., using the dataset's `map()` method, as we saw earlier), or you can include a preprocessing layer directly in your model. Let's look at this last option now.

For example, here is how you can implement a standardization layer using a `Lambda` layer. For each feature, it subtracts the mean and divides by its standard deviation (plus a tiny smoothing term to avoid division by zero):

```
means = np.mean(X_train, axis=0, keepdims=True)  
stds = np.std(X_train, axis=0, keepdims=True)  
eps = keras.backend.epsilon()  
model = keras.models.Sequential([  
    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),  
    [...] # other layers  
])
```

That's not too hard! However, you may prefer to use a nice self-contained custom layer (much like Scikit-Learn's `StandardScaler`), rather than having global variables like `means` and `stds` dangling around:

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())
```

Before you can use this standardization layer, you will need to adapt it to your dataset by calling the `adapt()` method and passing it a data sample. This will allow it to use the appropriate mean and standard deviation for each feature:

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

This sample must be large enough to be representative of your dataset, but it does not have to be the full training set: in general, a few hundred randomly selected instances will suffice (however, this depends on your task). Next, you can use this preprocessing layer like a normal layer:

```
model = keras.Sequential()
model.add(std_layer)
[...] # create the rest of the model
model.compile([...])
model.fit([...])
```

If you are thinking that Keras should contain a standardization layer like this one, here's some good news for you: by the time you read this, the `keras.layers.Normalization` layer will probably be available. It will work very much like our custom Standardization layer: first, create the layer, then adapt it to your dataset by passing a data sample to the `adapt()` method, and finally use the layer normally.

Now let's look at categorical features. We will start by encoding them as one-hot vectors.

Encoding Categorical Features Using One-Hot Vectors

Consider the `ocean_proximity` feature in the California housing dataset we explored in [Chapter 2](#): it is a categorical feature with five possible values: "`<1H OCEAN`", "`INLAND`", "`NEAR OCEAN`", "`NEAR BAY`", and "`ISLAND`". We need to encode this feature before we feed it to a neural network. Since there are very few categories, we can use one-hot encoding. For this, we first need to map each category to its index (0 to 4), which can be done using a lookup table:

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)
```

```

table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)

```

Let's go through this code:

- We first define the *vocabulary*: this is the list of all possible categories.
- Then we create a tensor with the corresponding indices (0 to 4).
- Next, we create an initializer for the lookup table, passing it the list of categories and their corresponding indices. In this example, we already have this data, so we use a `KeyValueTensorInitializer`; but if the categories were listed in a text file (with one category per line), we would use a `TextFileInitializer` instead.
- In the last two lines we create the lookup table, giving it the initializer and specifying the number of *out-of-vocabulary* (oov) buckets. If we look up a category that does not exist in the vocabulary, the lookup table will compute a hash of this category and use it to assign the unknown category to one of the oov buckets. Their indices start after the known categories, so in this example the indices of the two oov buckets are 5 and 6.

Why use oov buckets? Well, if the number of categories is large (e.g., zip codes, cities, words, products, or users) and the dataset is large as well, or it keeps changing, then getting the full list of categories may not be convenient. One solution is to define the vocabulary based on a data sample (rather than the whole training set) and add some oov buckets for the other categories that were not in the data sample. The more unknown categories you expect to find during training, the more oov buckets you should use. Indeed, if there are not enough oov buckets, there will be collisions: different categories will end up in the same bucket, so the neural network will not be able to distinguish them (at least not based on this feature).

Now let's use the lookup table to encode a small batch of categorical features to one-hot vectors:

```

>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
>>> cat_one_hot
<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
array([[0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.]], dtype=float32)>

```

As you can see, "NEAR BAY" was mapped to index 3, the unknown category "DESERT" was mapped to one of the two oov buckets (at index 5), and "INLAND" was mapped to

index 1, twice. Then we used `tf.one_hot()` to one-hot encode these indices. Notice that we have to tell this function the total number of indices, which is equal to the vocabulary size plus the number of oov buckets. Now you know how to encode categorical features to one-hot vectors using TensorFlow!

Just like earlier, it wouldn't be too difficult to bundle all of this logic into a nice self-contained class. Its `adapt()` method would take a data sample and extract all the distinct categories it contains. It would create a lookup table to map each category to its index (including unknown categories using oov buckets). Then its `call()` method would use the lookup table to map the input categories to their indices. Well, here's more good news: by the time you read this, Keras will probably include a layer called `keras.layers.TextVectorization`, which will be capable of doing exactly that: its `adapt()` method will extract the vocabulary from a data sample, and its `call()` method will convert each category to its index in the vocabulary. You could add this layer at the beginning of your model, followed by a `Lambda` layer that would apply the `tf.one_hot()` function, if you want to convert these indices to one-hot vectors.

This may not be the best solution, though. The size of each one-hot vector is the vocabulary length plus the number of oov buckets. This is fine when there are just a few possible categories, but if the vocabulary is large, it is much more efficient to encode them using *embeddings* instead.



As a rule of thumb, if the number of categories is lower than 10, then one-hot encoding is generally the way to go (but your mileage may vary!). If the number of categories is greater than 50 (which is often the case when you use hash buckets), then embeddings are usually preferable. In between 10 and 50 categories, you may want to experiment with both options and see which one works best for your use case.

Encoding Categorical Features Using Embeddings

An embedding is a trainable dense vector that represents a category. By default, embeddings are initialized randomly, so for example the "NEAR BAY" category could be represented initially by a random vector such as [0.131, 0.890], while the "NEAR OCEAN" category might be represented by another random vector such as [0.631, 0.791]. In this example, we use 2D embeddings, but the number of dimensions is a hyperparameter you can tweak. Since these embeddings are trainable, they will gradually improve during training; and as they represent fairly similar categories, Gradient Descent will certainly end up pushing them closer together, while it will tend to move them away from the "INLAND" category's embedding (see [Figure 13-4](#)). Indeed, the better the representation, the easier it will be for the neural network to make accurate predictions, so training tends to make embeddings useful representations of

the categories. This is called *representation learning* (we will see other types of representation learning in [Chapter 17](#)).

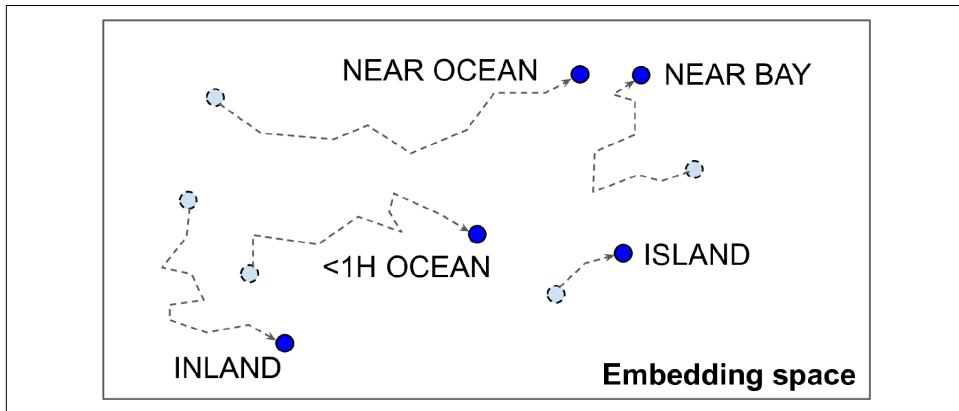


Figure 13-4. Embeddings will gradually improve during training

Word Embeddings

Not only will embeddings generally be useful representations for the task at hand, but quite often these same embeddings can be reused successfully for other tasks. The most common example of this is *word embeddings* (i.e., embeddings of individual words): when you are working on a natural language processing task, you are often better off reusing pretrained word embeddings than training your own.

The idea of using vectors to represent words dates back to the 1960s, and many sophisticated techniques have been used to generate useful vectors, including using neural networks. But things really took off in 2013, when Tomáš Mikolov and other Google researchers published a [paper](#)⁹ describing an efficient technique to learn word embeddings using neural networks, significantly outperforming previous attempts. This allowed them to learn embeddings on a very large corpus of text: they trained a neural network to predict the words near any given word, and obtained astounding word embeddings. For example, synonyms had very close embeddings, and semantically related words such as France, Spain, and Italy ended up clustered together.

It's not just about proximity, though: word embeddings were also organized along meaningful axes in the embedding space. Here is a famous example: if you compute King – Man + Woman (adding and subtracting the embedding vectors of these words), then the result will be very close to the embedding of the word Queen (see [Figure 13-5](#)). In other words, the word embeddings encode the concept of gender!

⁹ Tomas Mikolov et al., “Distributed Representations of Words and Phrases and Their Compositionalities,” *Proceedings of the 26th International Conference on Neural Information Processing Systems 2* (2013): 3111–3119.

Similarly, you can compute Madrid – Spain + France, and the result is close to Paris, which seems to show that the notion of capital city was also encoded in the embeddings.

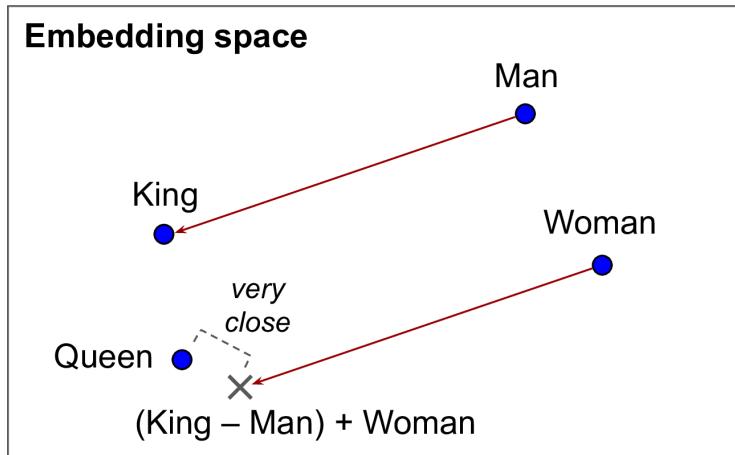


Figure 13-5. Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts

Unfortunately, word embeddings sometimes capture our worst biases. For example, although they correctly learn that Man is to King as Woman is to Queen, they also seem to learn that Man is to Doctor as Woman is to Nurse: quite a sexist bias! To be fair, this particular example is probably exaggerated, as was pointed out in a [2019 paper¹⁰](#) by Malvina Nissim et al. Nevertheless, ensuring fairness in Deep Learning algorithms is an important and active research topic.

Let's look at how we could implement embeddings manually, to understand how they work (then we will use a simple Keras layer instead). First, we need to create an *embedding matrix* containing each category's embedding, initialized randomly; it will have one row per category and per oov bucket, and one column per embedding dimension:

```
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])
embedding_matrix = tf.Variable(embed_init)
```

¹⁰ Malvina Nissim et al., “Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor,” arXiv preprint arXiv:1905.09866 (2019).

In this example we are using 2D embeddings, but as a rule of thumb embeddings typically have 10 to 300 dimensions, depending on the task and the vocabulary size (you will have to tune this hyperparameter).

This embedding matrix is a random 6×2 matrix, stored in a variable (so it can be tweaked by Gradient Descent during training):

```
>>> embedding_matrix
<tf.Variable 'Variable:0' shape=(6, 2) dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236],
       [0.74011743, 0.8724445 ],
       [0.22632635, 0.22319686],
       [0.3103881 , 0.7223358 ]], dtype=float32)>
```

Now let's encode the same batch of categorical features as earlier, but this time using these embeddings:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=741, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> tf.nn.embedding_lookup(embedding_matrix, cat_indices)
<tf.Tensor: id=864, shape=(4, 2), dtype=float32, numpy=
array([[0.74011743, 0.8724445 ],
       [0.3103881 , 0.7223358 ],
       [0.3528825 , 0.46448255],
       [0.3528825 , 0.46448255]], dtype=float32)>
```

The `tf.nn.embedding_lookup()` function looks up the rows in the embedding matrix, at the given indices—that's all it does. For example, the lookup table says that the "INLAND" category is at index 1, so the `tf.nn.embedding_lookup()` function returns the embedding at row 1 in the embedding matrix (twice): $[0.3528825, 0.46448255]$.

Keras provides a `keras.layers.Embedding` layer that handles the embedding matrix (trainable, by default); when the layer is created it initializes the embedding matrix randomly, and then when it is called with some category indices it returns the rows at those indices in the embedding matrix:

```
>>> embedding = keras.layers.Embedding(input_dim=len(vocab) + num_oov_buckets,
...                                         output_dim=embedding_dim)
...
>>> embedding(cat_indices)
<tf.Tensor: id=814, shape=(4, 2), dtype=float32, numpy=
array([[ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358],
       [-0.01471175, -0.00355174],
       [-0.01471175, -0.00355174]], dtype=float32)>
```

Putting everything together, we can now create a Keras model that can process categorical features (along with regular numerical features) and learn an embedding for each category (as well as for each oov bucket):

```
regular_inputs = keras.layers.Input(shape=[8])
categories = keras.layers.Input(shape=[], dtype=tf.string)
cat_indices = keras.layers.Lambda(lambda cats: table.lookup(cats))(categories)
cat_embed = keras.layers.Embedding(input_dim=6, output_dim=2)(cat_indices)
encoded_inputs = keras.layers.concatenate([regular_inputs, cat_embed])
outputs = keras.layers.Dense(1)(encoded_inputs)
model = keras.models.Model(inputs=[regular_inputs, categories],
                           outputs=[outputs])
```

This model takes two inputs: a regular input containing eight numerical features per instance, plus a categorical input (containing one categorical feature per instance). It uses a `Lambda` layer to look up each category's index, then it looks up the embeddings for these indices. Next, it concatenates the embeddings and the regular inputs in order to give the encoded inputs, which are ready to be fed to a neural network. We could add any kind of neural network at this point, but we just add a dense output layer, and we create the Keras model.

When the `keras.layers.TextVectorization` layer is available, you can call its `adapt()` method to make it extract the vocabulary from a data sample (it will take care of creating the lookup table for you). Then you can add it to your model, and it will perform the index lookup (replacing the `Lambda` layer in the previous code example).



One-hot encoding followed by a `Dense` layer (with no activation function and no biases) is equivalent to an `Embedding` layer. However, the `Embedding` layer uses way fewer computations (the performance difference becomes clear when the size of the embedding matrix grows). The `Dense` layer's weight matrix plays the role of the embedding matrix. For example, using one-hot vectors of size 20 and a `Dense` layer with 10 units is equivalent to using an `Embedding` layer with `input_dim=20` and `output_dim=10`. As a result, it would be wasteful to use more embedding dimensions than the number of units in the layer that follows the `Embedding` layer.

Now let's look a bit more closely at the Keras preprocessing layers.

Keras Preprocessing Layers

The TensorFlow team is working on providing a set of standard [Keras preprocessing layers](#). They will probably be available by the time you read this; however, the API may change slightly by then, so please refer to the notebook for this chapter if anything behaves unexpectedly. This new API will likely supersede the existing Feature

Columns API, which is harder to use and less intuitive (if you want to learn more about the Feature Columns API anyway, please check out the notebook for this chapter).

We already discussed two of these layers: the `keras.layers.Normalization` layer that will perform feature standardization (it will be equivalent to the `Standardization` layer we defined earlier), and the `TextVectorization` layer that will be capable of encoding each word in the inputs into its index in the vocabulary. In both cases, you create the layer, you call its `adapt()` method with a data sample, and then you use the layer normally in your model. The other preprocessing layers will follow the same pattern.

The API will also include a `keras.layers.Discretization` layer that will chop continuous data into different bins and encode each bin as a one-hot vector. For example, you could use it to discretize prices into three categories, (low, medium, high), which would be encoded as [1, 0, 0], [0, 1, 0], and [0, 0, 1], respectively. Of course this loses a lot of information, but in some cases it can help the model detect patterns that would otherwise not be obvious when just looking at the continuous values.



The `Discretization` layer will not be differentiable, and it should only be used at the start of your model. Indeed, the model's preprocessing layers will be frozen during training, so their parameters will not be affected by Gradient Descent, and thus they do not need to be differentiable. This also means that you should not use an `Embedding` layer directly in a custom preprocessing layer, if you want it to be trainable: instead, it should be added separately to your model, as in the previous code example.

It will also be possible to chain multiple preprocessing layers using the `PreprocessingStage` class. For example, the following code will create a preprocessing pipeline that will first normalize the inputs, then discretize them (this may remind you of Scikit-Learn pipelines). After you adapt this pipeline to a data sample, you can use it like a regular layer in your models (but again, only at the start of the model, since it contains a nondifferentiable preprocessing layer):

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...])
pipeline = keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

The `TextVectorization` layer will also have an option to output word-count vectors instead of word indices. For example, if the vocabulary contains three words, say ["and", "basketball", "more"], then the text "more and more" will be mapped to the vector [1, 0, 2]: the word "and" appears once, the word "basketball" does not appear at all, and the word "more" appears twice. This text representation is called a

bag of words, since it completely loses the order of the words. Common words like "and" will have a large value in most texts, even though they are usually the least interesting (e.g., in the text "more and more basketball" the word "basketball" is clearly the most important, precisely because it is not a very frequent word). So, the word counts should be normalized in a way that reduces the importance of frequent words. A common way to do this is to divide each word count by the log of the total number of training instances in which the word appears. This technique is called *Term-Frequency × Inverse-Document-Frequency* (TF-IDF). For example, let's imagine that the words "and", "basketball", and "more" appear respectively in 200, 10, and 100 text instances in the training set: in this case, the final vector will be $[1/\log(200), 0/\log(10), 2/\log(100)]$, which is approximately equal to $[0.19, 0., 0.43]$. The `TextVectorization` layer will (likely) have an option to perform TF-IDF.



If the standard preprocessing layers are insufficient for your task, you will still have the option to create your own custom preprocessing layer, much like we did earlier with the `Standardization` class. Create a subclass of the `keras.layers.PreprocessingLayer` class with an `adapt()` method, which should take a `data_sample` argument and optionally an `extra reset_state` argument: if `True`, then the `adapt()` method should reset any existing state before computing the new state; if `False`, it should try to update the existing state.

As you can see, these Keras preprocessing layers will make preprocessing much easier! Now, whether you choose to write your own preprocessing layers or use Keras's (or even use the Feature Columns API), all the preprocessing will be done on the fly. During training, however, it may be preferable to perform preprocessing ahead of time. Let's see why we'd want to do that and how we'd go about it.

TF Transform

If preprocessing is computationally expensive, then handling it before training rather than on the fly may give you a significant speedup: the data will be preprocessed just once per instance *before* training, rather than once per instance and per epoch *during* training. As mentioned earlier, if the dataset is small enough to fit in RAM, you can use its `cache()` method. But if it is too large, then tools like Apache Beam or Spark will help. They let you run efficient data processing pipelines over large amounts of data, even distributed across multiple servers, so you can use them to preprocess all the training data before training.

This works great and indeed can speed up training, but there is one problem: once your model is trained, suppose you want to deploy it to a mobile app. In that case you will need to write some code in your app to take care of preprocessing the data before

it is fed to the model. And suppose you also want to deploy the model to TensorFlow.js so that it runs in a web browser? Once again, you will need to write some pre-processing code. This can become a maintenance nightmare: whenever you want to change the preprocessing logic, you will need to update your Apache Beam code, your mobile app code, and your JavaScript code. This is not only time-consuming, but also error-prone: you may end up with subtle differences between the preprocessing operations performed before training and the ones performed in your app or in the browser. This *training/serving skew* will lead to bugs or degraded performance.

One improvement would be to take the trained model (trained on data that was pre-processed by your Apache Beam or Spark code) and, before deploying it to your app or the browser, add extra preprocessing layers to take care of preprocessing on the fly. That's definitely better, since now you just have two versions of your preprocessing code: the Apache Beam or Spark code, and the preprocessing layers' code.

But what if you could define your preprocessing operations just once? This is what TF Transform was designed for. It is part of [TensorFlow Extended \(TFX\)](#), an end-to-end platform for productionizing TensorFlow models. First, to use a TFX component such as TF Transform, you must install it; it does not come bundled with TensorFlow. You then define your preprocessing function just once (in Python), by using TF Transform functions for scaling, bucketizing, and more. You can also use any TensorFlow operation you need. Here is what this preprocessing function might look like if we just had two features:

```
import tensorflow_transform as tft

def preprocess(inputs): # inputs = a batch of input features
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age)
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
    }
```

Next, TF Transform lets you apply this `preprocess()` function to the whole training set using Apache Beam (it provides an `AnalyzeAndTransformDataset` class that you can use for this purpose in your Apache Beam pipeline). In the process, it will also compute all the necessary statistics over the whole training set: in this example, the mean and standard deviation of the `housing_median_age` feature, and the vocabulary for the `ocean_proximity` feature. The components that compute these statistics are called *analyzers*.

Importantly, TF Transform will also generate an equivalent TensorFlow Function that you can plug into the model you deploy. This TF Function includes some constants

that correspond to all the necessary statistics computed by Apache Beam (the mean, standard deviation, and vocabulary).

With the Data API, TFRecords, the Keras preprocessing layers, and TF Transform, you can build highly scalable input pipelines for training and benefit from fast and portable data preprocessing in production.

But what if you just wanted to use a standard dataset? Well in that case, things are much simpler: just use TFDS!

The TensorFlow Datasets (TFDS) Project

The [TensorFlow Datasets](#) project makes it very easy to download common datasets, from small ones like MNIST or Fashion MNIST to huge datasets like ImageNet (you will need quite a bit of disk space!). The list includes image datasets, text datasets (including translation datasets), and audio and video datasets. You can visit <https://hml.info/tfds> to view the full list, along with a description of each dataset.

TFDS is not bundled with TensorFlow, so you need to install the `tensorflow-datasets` library (e.g., using pip). Then call the `tfds.load()` function, and it will download the data you want (unless it was already downloaded earlier) and return the data as a dictionary of datasets (typically one for training and one for testing, but this depends on the dataset you choose). For example, let's download MNIST:

```
import tensorflow_datasets as tfds

dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

You can then apply any transformation you want (typically shuffling, batching, and prefetching), and you're ready to train your model. Here is a simple example:

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```



The `load()` function shuffles each data shard it downloads (only for the training set). This may not be sufficient, so it's best to shuffle the training data some more.

Note that each item in the dataset is a dictionary containing both the features and the labels. But Keras expects each item to be a tuple containing two elements (again, the features and the labels). You could transform the dataset using the `map()` method, like this:

```
mnist_train = mnist_train.shuffle(10000).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

But it's simpler to ask the `load()` function to do this for you by setting `as_supervised=True` (obviously this works only for labeled datasets). You can also specify the batch size if you want. Then you can pass the dataset directly to your tf.keras model:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, epochs=5)
```

This was quite a technical chapter, and you may feel that it is a bit far from the abstract beauty of neural networks, but the fact is Deep Learning often involves large amounts of data, and knowing how to load, parse, and preprocess it efficiently is a crucial skill to have. In the next chapter, we will look at convolutional neural networks, which are among the most successful neural net architectures for image processing and many other applications.

Exercises

1. Why would you want to use the Data API?
2. What are the benefits of splitting a large dataset into multiple files?
3. During training, how can you tell that your input pipeline is the bottleneck? What can you do to fix it?
4. Can you save any binary data to a TFRecord file, or only serialized protocol buffers?
5. Why would you go through the hassle of converting all your data to the `Example` protobuf format? Why not use your own protobuf definition?
6. When using TFRecords, when would you want to activate compression? Why not do it systematically?
7. Data can be preprocessed directly when writing the data files, or within the `tf.data` pipeline, or in preprocessing layers within your model, or using TF Transform. Can you list a few pros and cons of each option?
8. Name a few common techniques you can use to encode categorical features. What about text?
9. Load the Fashion MNIST dataset (introduced in [Chapter 10](#)); split it into a training set, a validation set, and a test set; shuffle the training set; and save each dataset to multiple TFRecord files. Each record should be a serialized `Example` protobuf with two features: the serialized image (use `tf.io.serialize_tensor()`

to serialize each image), and the label.¹¹ Then use `tf.data` to create an efficient dataset for each set. Finally, use a Keras model to train these datasets, including a preprocessing layer to standardize each input feature. Try to make the input pipeline as efficient as possible, using TensorBoard to visualize profiling data.

10. In this exercise you will download a dataset, split it, create a `tf.data.Dataset` to load it and preprocess it efficiently, then build and train a binary classification model containing an `Embedding` layer:
 - a. Download the [Large Movie Review Dataset](#), which contains 50,000 movies reviews from the [Internet Movie Database](#). The data is organized in two directories, `train` and `test`, each containing a `pos` subdirectory with 12,500 positive reviews and a `neg` subdirectory with 12,500 negative reviews. Each review is stored in a separate text file. There are other files and folders (including pre-processed bag-of-words), but we will ignore them in this exercise.
 - b. Split the test set into a validation set (15,000) and a test set (10,000).
 - c. Use `tf.data` to create an efficient dataset for each set.
 - d. Create a binary classification model, using a `TextVectorization` layer to preprocess each review. If the `TextVectorization` layer is not yet available (or if you like a challenge), try to create your own custom preprocessing layer: you can use the functions in the `tf.strings` package, for example `lower()` to make everything lowercase, `regex_replace()` to replace punctuation with spaces, and `split()` to split words on spaces. You should use a lookup table to output word indices, which must be prepared in the `adapt()` method.
 - e. Add an `Embedding` layer and compute the mean embedding for each review, multiplied by the square root of the number of words (see [Chapter 16](#)). This rescaled mean embedding can then be passed to the rest of your model.
 - f. Train the model and see what accuracy you get. Try to optimize your pipelines to make training as fast as possible.
 - g. Use TFDS to load the same dataset more easily: `tfds.load("imdb_reviews")`.

Solutions to these exercises are available in [Appendix A](#).

¹¹ For large images, you could use `tf.io.encode_jpeg()` instead. This would save a lot of space, but it would lose a bit of image quality.

Deep Computer Vision Using Convolutional Neural Networks

Although IBM's Deep Blue supercomputer beat the chess world champion Garry Kasparov back in 1996, it wasn't until fairly recently that computers were able to reliably perform seemingly trivial tasks such as detecting a puppy in a picture or recognizing spoken words. Why are these tasks so effortless to us humans? The answer lies in the fact that perception largely takes place outside the realm of our consciousness, within specialized visual, auditory, and other sensory modules in our brains. By the time sensory information reaches our consciousness, it is already adorned with high-level features; for example, when you look at a picture of a cute puppy, you cannot choose *not* to see the puppy, *not* to notice its cuteness. Nor can you explain *how* you recognize a cute puppy; it's just obvious to you. Thus, we cannot trust our subjective experience: perception is not trivial at all, and to understand it we must look at how the sensory modules work.

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In the last few years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in [Chapter 11](#) for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful at many other tasks, such as voice recognition and natural language processing. However, we will focus on visual applications for now.

In this chapter we will explore where CNNs came from, what their building blocks look like, and how to implement them using TensorFlow and Keras. Then we will discuss some of the best CNN architectures, as well as other visual tasks, including

object detection (classifying multiple objects in an image and placing bounding boxes around them) and semantic segmentation (classifying each pixel according to the class of the object it belongs to).

The Architecture of the Visual Cortex

David H. Hubel and Torsten Wiesel performed a series of experiments on cats in 1958¹ and 1959² (and a few years later on monkeys³), giving crucial insights into the structure of the visual cortex (the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work). In particular, they showed that many neurons in the visual cortex have a small *local receptive field*, meaning they react only to visual stimuli located in a limited region of the visual field (see Figure 14-1, in which the local receptive fields of five neurons are represented by dashed circles). The receptive fields of different neurons may overlap, and together they tile the whole visual field.

Moreover, the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). They also noticed that some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons (in Figure 14-1, notice that each neuron is connected only to a few neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

¹ David H. Hubel, “Single Unit Activity in Striate Cortex of Unrestrained Cats,” *The Journal of Physiology* 147 (1959): 226–238.

² David H. Hubel and Torsten N. Wiesel, “Receptive Fields of Single Neurons in the Cat’s Striate Cortex,” *The Journal of Physiology* 148 (1959): 574–591.

³ David H. Hubel and Torsten N. Wiesel, “Receptive Fields and Functional Architecture of Monkey Striate Cortex,” *The Journal of Physiology* 195 (1968): 215–243.

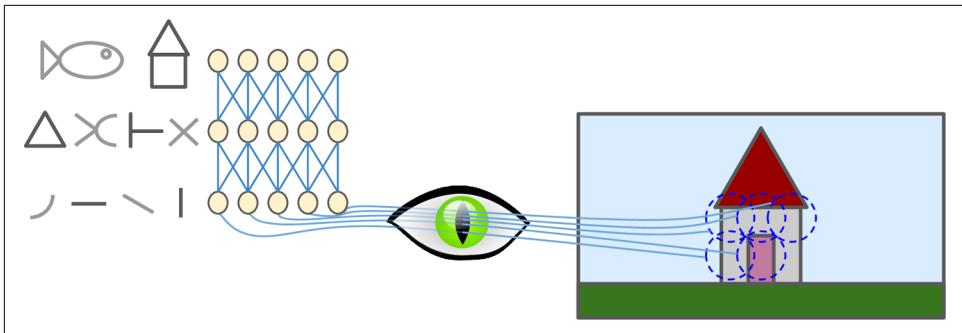


Figure 14-1. Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields.

These studies of the visual cortex inspired the [neocognitron](#),⁴ introduced in 1980, which gradually evolved into what we now call *convolutional neural networks*. An important milestone was a [1998 paper](#)⁵ by Yann LeCun et al. that introduced the famous *LeNet-5* architecture, widely used by banks to recognize handwritten check numbers. This architecture has some building blocks that you already know, such as fully connected layers and sigmoid activation functions, but it also introduces two new building blocks: *convolutional layers* and *pooling layers*. Let's look at them now.



Why not simply use a deep neural network with fully connected layers for image recognition tasks? Unfortunately, although this works fine for small images (e.g., MNIST), it breaks down for larger images because of the huge number of parameters it requires. For example, a 100×100 -pixel image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer. CNNs solve this problem using partially connected layers and weight sharing.

⁴ Kunihiko Fukushima, “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position,” *Biological Cybernetics* 36 (1980): 193–202.

⁵ Yann LeCun et al., “Gradient-Based Learning Applied to Document Recognition,” *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

Convolutional Layers

The most important building block of a CNN is the *convolutional layer*:⁶ neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in the layers discussed in previous chapters), but only to pixels in their receptive fields (see [Figure 14-2](#)). In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

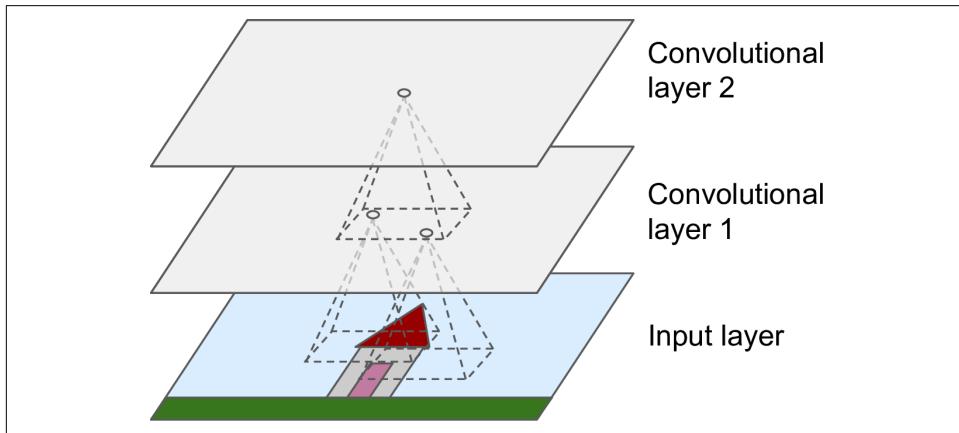


Figure 14-2. CNN layers with rectangular local receptive fields



All the multilayer neural networks we've looked at so far had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network. In a CNN each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

6 A convolution is a mathematical operation that slides one function over another and measures the integral of their pointwise multiplication. It has deep connections with the Fourier transform and the Laplace transform and is heavily used in signal processing. Convolutional layers actually use cross-correlations, which are very similar to convolutions (see <https://homl.info/76> for more details).

A neuron located in row i , column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field (see [Figure 14-3](#)). In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram. This is called *zero padding*.

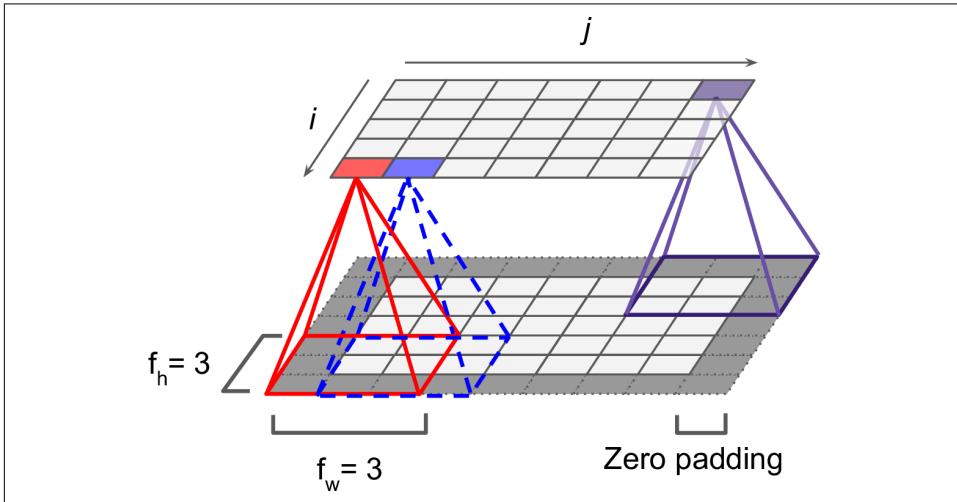


Figure 14-3. Connections between layers and zero padding

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in [Figure 14-4](#). This dramatically reduces the model's computational complexity. The shift from one receptive field to the next is called the *stride*. In the diagram, a 5×7 input layer (plus zero padding) is connected to a 3×4 layer, using 3×3 receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). A neuron located in row i , column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w$ to $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

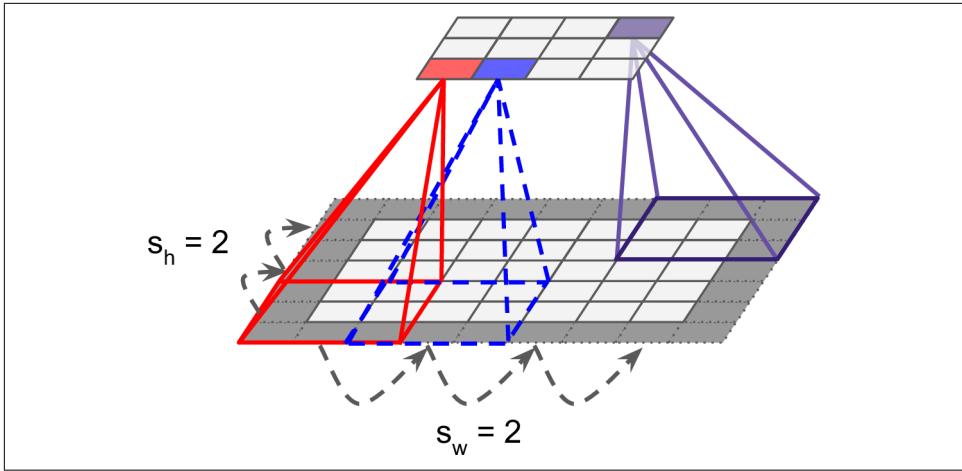


Figure 14-4. Reducing dimensionality using a stride of 2

Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, Figure 14-5 shows two possible sets of weights, called *filters* (or *convolution kernels*). The first one is represented as a black square with a vertical white line in the middle (it is a 7×7 matrix full of 0s except for the central column, which is full of 1s); neurons using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will get multiplied by 0, except for the ones located in the central vertical line). The second filter is a black square with a horizontal white line in the middle. Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

Now if all neurons in a layer use the same vertical line filter (and the same bias term), and you feed the network the input image shown in Figure 14-5 (the bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what you get if all neurons use the same horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out. Thus, a layer full of neurons using the same filter outputs a *feature map*, which highlights the areas in an image that activate the filter the most. Of course, you do not have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

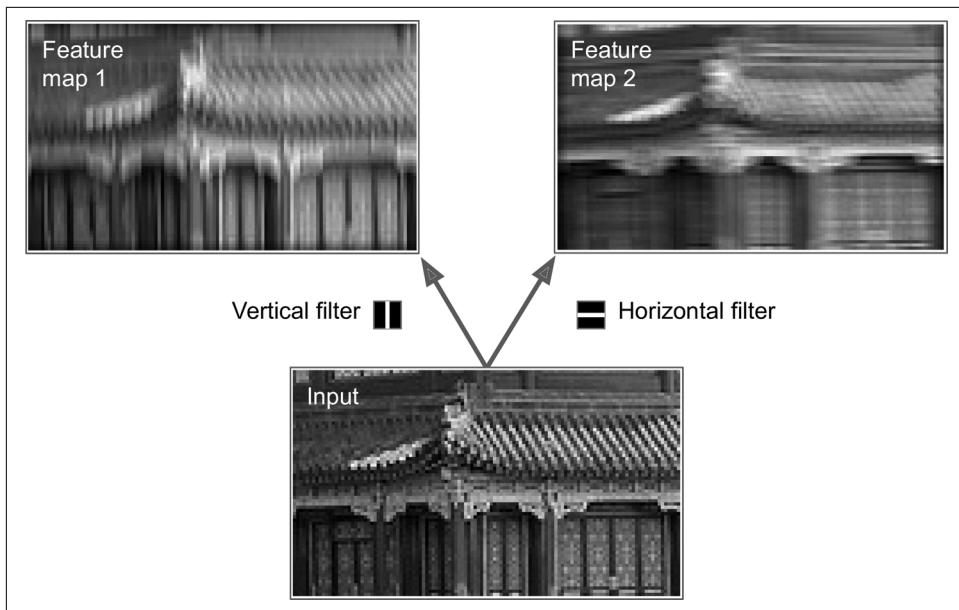


Figure 14-5. Applying two different filters to get two feature maps

Stacking Multiple Feature Maps

Up to now, for simplicity, I have represented the output of each convolutional layer as a 2D layer, but in reality a convolutional layer has multiple filters (you decide how many) and outputs one feature map per filter, so it is more accurately represented in 3D (see [Figure 14-6](#)). It has one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same weights and bias term). Neurons in different feature maps use different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the previous layers' feature maps. In short, a convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.



The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model. Once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location.

Input images are also composed of multiple sublayers: one per *color channel*. There are typically three: red, green, and blue (RGB). Grayscale images have just one

channel, but some images may have much more—for example, satellite images that capture extra light frequencies (such as infrared).

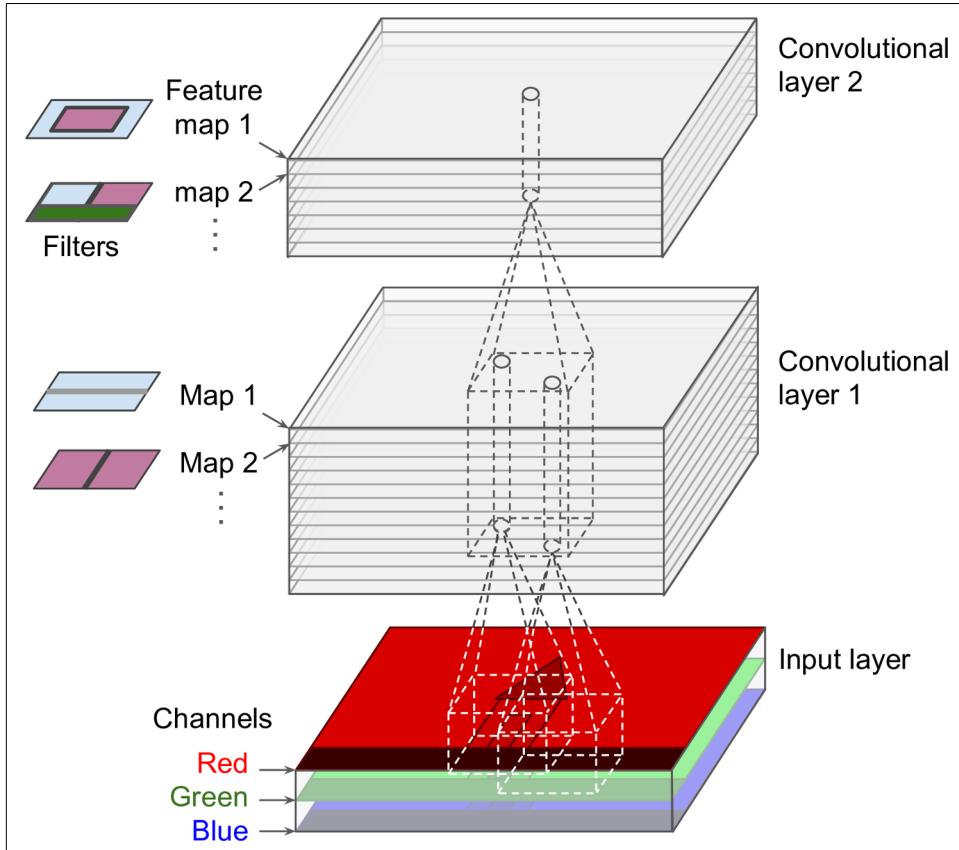


Figure 14-6. Convolutional layers with multiple feature maps, and images with three color channels

Specifically, a neuron located in row i , column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer $l - 1$, located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$, across all feature maps (in layer $l - 1$). Note that all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

Equation 14-1 summarizes the preceding explanations in one big mathematical equation: it shows how to compute the output of a given neuron in a convolutional layer.

It is a bit ugly due to all the different indices, but all it does is calculate the weighted sum of all the inputs, plus the bias term.

Equation 14-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

In this equation:

- $z_{i,j,k}$ is the output of the neuron located in row i , column j in feature map k of the convolutional layer (layer l).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l - 1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l - 1$, row i' , column j' , feature map k' (or channel k' if the previous layer is the input layer).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k .
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u , column v (relative to the neuron's receptive field), and feature map k' .

TensorFlow Implementation

In TensorFlow, each input image is typically represented as a 3D tensor of shape `[height, width, channels]`. A mini-batch is represented as a 4D tensor of shape `[mini-batch size, height, width, channels]`. The weights of a convolutional layer are represented as a 4D tensor of shape `[fh, fw, fn', fn]`. The bias terms of a convolutional layer are simply represented as a 1D tensor of shape `[fn]`.

Let's look at a simple example. The following code loads two sample images, using Scikit-Learn's `load_sample_image()` (which loads two color images, one of a Chinese temple, and the other of a flower), then it creates two filters and applies them to both images, and finally it displays one of the resulting feature maps. Note that you must pip install the `Pillow` package to use `load_sample_image()`.

```
from sklearn.datasets import load_sample_image

# Load sample images
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
```

```

images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # vertical line
filters[3, :, :, 1] = 1 # horizontal line

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.show()

```

Let's go through this code:

- The pixel intensity for each color channel is represented as a byte from 0 to 255, so we scale these features simply by dividing by 255, to get floats ranging from 0 to 1.
- Then we create two 7×7 filters (one with a vertical white line in the middle, and the other with a horizontal white line in the middle).
- We apply them to both images using the `tf.nn.conv2d()` function, which is part of TensorFlow's low-level Deep Learning API. In this example, we use zero padding (`padding="SAME"`) and a stride of 1.
- Finally, we plot one of the resulting feature maps (similar to the top-right image in [Figure 14-5](#)).

The `tf.nn.conv2d()` line deserves a bit more explanation:

- `images` is the input mini-batch (a 4D tensor, as explained earlier).
- `filters` is the set of filters to apply (also a 4D tensor, as explained earlier).
- `strides` is equal to 1, but it could also be a 1D array with four elements, where the two central elements are the vertical and horizontal strides (s_h and s_w). The first and last elements must currently be equal to 1. They may one day be used to specify a batch stride (to skip some instances) and a channel stride (to skip some of the previous layer's feature maps or channels).
- `padding` must be either "SAME" or "VALID":
 - If set to "SAME", the convolutional layer uses zero padding if necessary. The output size is set to the number of input neurons divided by the stride, rounded up. For example, if the input size is 13 and the stride is 5 (see [Figure 14-7](#)), then the output size is 3 (i.e., $13 / 5 = 2.6$, rounded up to 3). Then zeros are added as evenly as possible around the inputs, as needed. When `strides=1`, the layer's outputs will have the same spatial dimensions (width and height) as its inputs, hence the name *same*.

- If set to "VALID", the convolutional layer does *not* use zero padding and may ignore some rows and columns at the bottom and right of the input image, depending on the stride, as shown in [Figure 14-7](#) (for simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension). This means that every neuron's receptive field lies strictly within valid positions inside the input (it does not go out of bounds), hence the name *valid*.

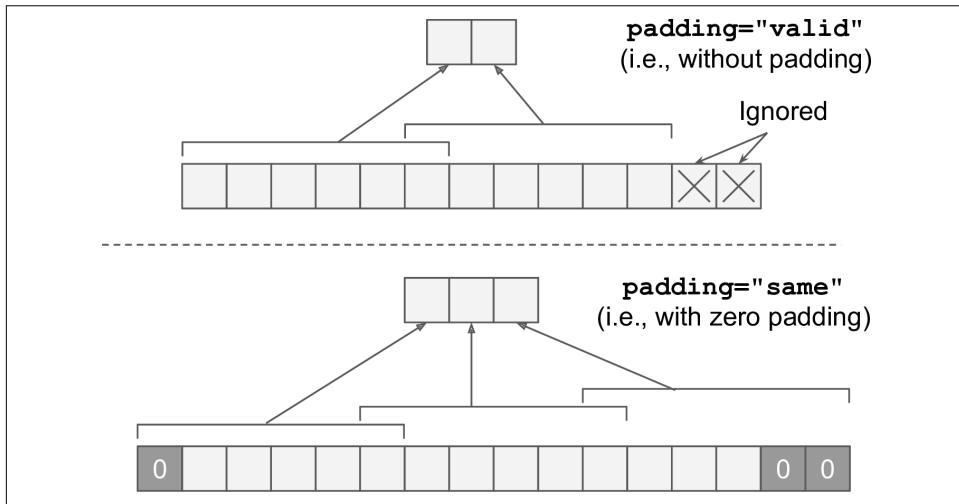


Figure 14-7. Padding="SAME" or "VALID" (with input width 13, filter width 6, stride 5)

In this example we manually defined the filters, but in a real CNN you would normally define filters as trainable variables so the neural net can learn which filters work best, as explained earlier. Instead of manually creating the variables, use the `keras.layers.Conv2D` layer:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="same", activation="relu")
```

This code creates a `Conv2D` layer with 32 filters, each 3×3 , using a stride of 1 (both horizontally and vertically) and "same" padding, and applying the ReLU activation function to its outputs. As you can see, convolutional layers have quite a few hyperparameters: you must choose the number of filters, their height and width, the strides, and the padding type. As always, you can use cross-validation to find the right hyperparameter values, but this is very time-consuming. We will discuss common CNN architectures later, to give you some idea of which hyperparameter values work best in practice.

Memory Requirements

Another problem with CNNs is that the convolutional layers require a huge amount of RAM. This is especially true during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass.

For example, consider a convolutional layer with 5×5 filters, outputting 200 feature maps of size 150×100 , with stride 1 and "same" padding. If the input is a 150×100 RGB image (three channels), then the number of parameters is $(5 \times 5 \times 3 + 1) \times 200 = 15,200$ (the + 1 corresponds to the bias terms), which is fairly small compared to a fully connected layer.⁷ However, each of the 200 feature maps contains 150×100 neurons, and each of these neurons needs to compute a weighted sum of its $5 \times 5 \times 3 = 75$ inputs: that's a total of 225 million float multiplications. Not as bad as a fully connected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy $200 \times 150 \times 100 \times 32 = 96$ million bits (12 MB) of RAM.⁸ And that's just for one instance—if a training batch contains 100 instances, then this layer will use up 1.2 GB of RAM!

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.



If training crashes because of an out-of-memory error, you can try reducing the mini-batch size. Alternatively, you can try reducing dimensionality using a stride, or removing a few layers. Or you can try using 16-bit floats instead of 32-bit floats. Or you could distribute the CNN across multiple devices.

Now let's look at the second common building block of CNNs: the *pooling layer*.

Pooling Layers

Once you understand how convolutional layers work, the pooling layers are quite easy to grasp. Their goal is to *subsample* (i.e., shrink) the input image in order to

⁷ A fully connected layer with 150×100 neurons, each connected to all $150 \times 100 \times 3$ inputs, would have $150^2 \times 100^2 \times 3 = 675$ million parameters!

⁸ In the international system of units (SI), 1 MB = 1,000 KB = $1,000 \times 1,000$ bytes = $1,000 \times 1,000 \times 8$ bits.

reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting).

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean. [Figure 14-8](#) shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2×2 *pooling kernel*,⁹ with a stride of 2 and no padding. Only the max input value in each receptive field makes it to the next layer, while the other inputs are dropped. For example, in the lower-left receptive field in [Figure 14-8](#), the input values are 1, 5, 3, 2, so only the max value, 5, is propagated to the next layer. Because of the stride of 2, the output image has half the height and half the width of the input image (rounded down since we use no padding).

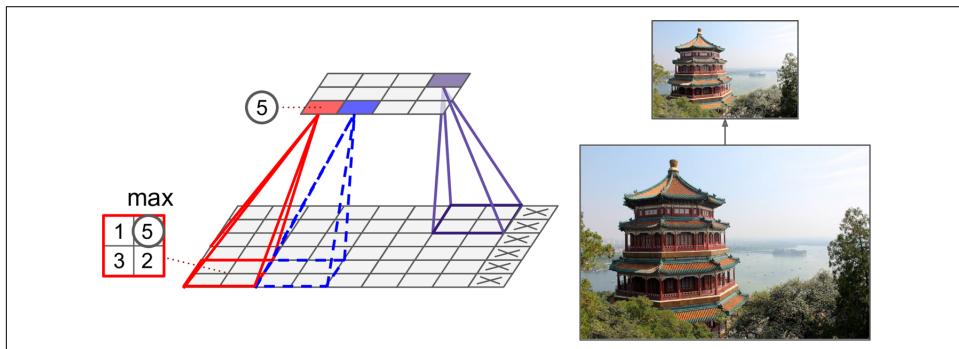


Figure 14-8. Max pooling layer (2×2 pooling kernel, stride 2, no padding)



A pooling layer typically works on every input channel independently, so the output depth is the same as the input depth.

Other than reducing computations, memory usage, and the number of parameters, a max pooling layer also introduces some level of *invariance* to small translations, as shown in [Figure 14-9](#). Here we assume that the bright pixels have a lower value than dark pixels, and we consider three images (A, B, C) going through a max pooling layer with a 2×2 kernel and stride 2. Images B and C are the same as image A, but

⁹ Other kernels we've discussed so far had weights, but pooling kernels do not: they are just stateless sliding windows.

shifted by one and two pixels to the right. As you can see, the outputs of the max pooling layer for images A and B are identical. This is what translation invariance means. For image C, the output is different: it is shifted one pixel to the right (but there is still 75% invariance). By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale. Moreover, max pooling offers a small amount of rotational invariance and a slight scale invariance. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

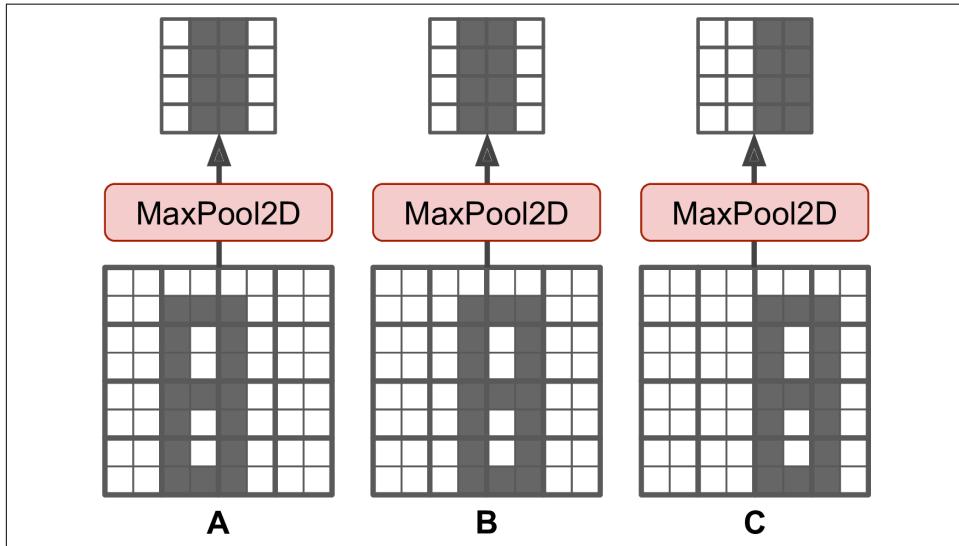


Figure 14-9. Invariance to small translations

However, max pooling has some downsides too. Firstly, it is obviously very destructive: even with a tiny 2×2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75% of the input values. And in some applications, invariance is not desirable. Take semantic segmentation (the task of classifying each pixel in an image according to the object that pixel belongs to, which we'll explore later in this chapter): obviously, if the input image is translated by one pixel to the right, the output should also be translated by one pixel to the right. The goal in this case is *equivariance*, not invariance: a small change to the inputs should lead to a corresponding small change in the output.

TensorFlow Implementation

Implementing a max pooling layer in TensorFlow is quite easy. The following code creates a max pooling layer using a 2×2 kernel. The strides default to the kernel size, so this layer will use a stride of 2 (both horizontally and vertically). By default, it uses "valid" padding (i.e., no padding at all):

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

To create an *average pooling layer*, just use `AvgPool2D` instead of `MaxPool2D`. As you might expect, it works exactly like a max pooling layer, except it computes the mean rather than the max. Average pooling layers used to be very popular, but people mostly use max pooling layers now, as they generally perform better. This may seem surprising, since computing the mean generally loses less information than computing the max. But on the other hand, max pooling preserves only the strongest features, getting rid of all the meaningless ones, so the next layers get a cleaner signal to work with. Moreover, max pooling offers stronger translation invariance than average pooling, and it requires slightly less compute.

Note that max pooling and average pooling can be performed along the depth dimension rather than the spatial dimensions, although this is not as common. This can allow the CNN to learn to be invariant to various features. For example, it could learn multiple filters, each detecting a different rotation of the same pattern (such as handwritten digits; see [Figure 14-10](#)), and the depthwise max pooling layer would ensure that the output is the same regardless of the rotation. The CNN could similarly learn to be invariant to anything else: thickness, brightness, skew, color, and so on.

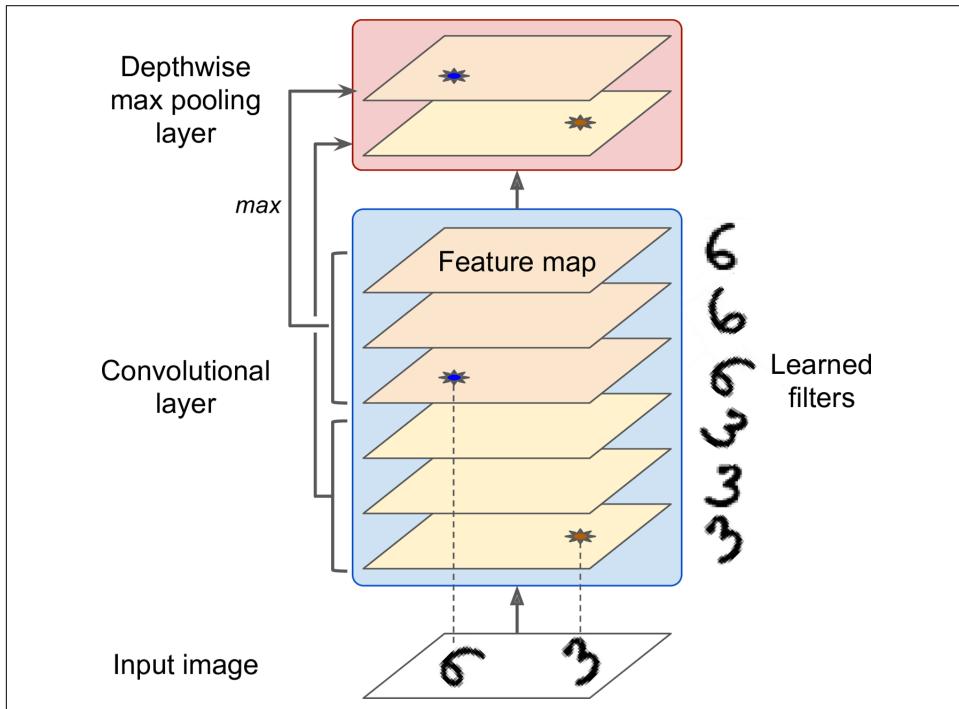


Figure 14-10. Depthwise max pooling can help the CNN learn any invariance

Keras does not include a depthwise max pooling layer, but TensorFlow's low-level Deep Learning API does: just use the `tf.nn.max_pool()` function, and specify the kernel size and strides as 4-tuples (i.e., tuples of size 4). The first three values of each should be 1: this indicates that the kernel size and stride along the batch, height, and width dimensions should be 1. The last value should be whatever kernel size and stride you want along the depth dimension—for example, 3 (this must be a divisor of the input depth; it will not work if the previous layer outputs 20 feature maps, since 20 is not a multiple of 3):

```
output = tf.nn.max_pool(images,
                        ksize=(1, 1, 1, 3),
                        strides=(1, 1, 1, 3),
                        padding="valid")
```

If you want to include this as a layer in your Keras models, wrap it in a `Lambda` layer (or create a custom Keras layer):

```
depth_pool = keras.layers.Lambda(
    lambda X: tf.nn.max_pool(X, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3),
                            padding="valid"))
```

One last type of pooling layer that you will often see in modern architectures is the *global average pooling layer*. It works very differently: all it does is compute the mean of each entire feature map (it's like an average pooling layer using a pooling kernel with the same spatial dimensions as the inputs). This means that it just outputs a single number per feature map and per instance. Although this is of course extremely destructive (most of the information in the feature map is lost), it can be useful as the output layer, as we will see later in this chapter. To create such a layer, simply use the `keras.layers.GlobalAvgPool2D` class:

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

It's equivalent to this simple `Lambda` layer, which computes the mean over the spatial dimensions (height and width):

```
global_avg_pool = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

Now you know all the building blocks to create convolutional neural networks. Let's see how to assemble them.

CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps), thanks to the convolutional layers (see [Figure 14-11](#)). At the top of the stack, a regular feedforward neural network is added, composed of a few

fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

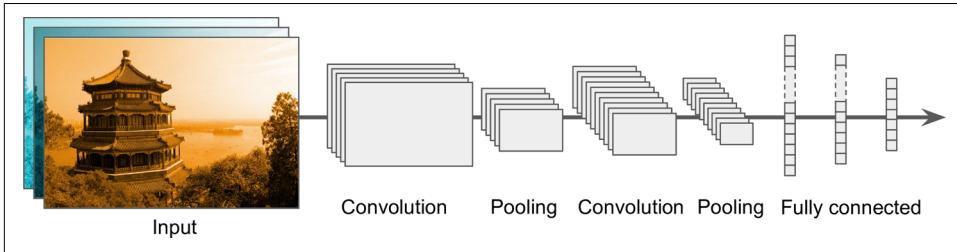


Figure 14-11. Typical CNN architecture



A common mistake is to use convolution kernels that are too large. For example, instead of using a convolutional layer with a 5×5 kernel, stack two layers with 3×3 kernels: it will use fewer parameters and require fewer computations, and it will usually perform better. One exception is for the first convolutional layer: it can typically have a large kernel (e.g., 5×5), usually with a stride of 2 or more: this will reduce the spatial dimension of the image without losing too much information, and since the input image only has three channels in general, it will not be too costly.

Here is how you can implement a simple CNN to tackle the Fashion MNIST dataset (introduced in [Chapter 10](#)):

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                       input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

Let's go through this model:

- The first layer uses 64 fairly large filters (7×7) but no stride because the input images are not very large. It also sets `input_shape=[28, 28, 1]`, because the images are 28×28 pixels, with a single color channel (i.e., grayscale).
- Next we have a max pooling layer which uses a pool size of 2, so it divides each spatial dimension by a factor of 2.
- Then we repeat the same structure twice: two convolutional layers followed by a max pooling layer. For larger images, we could repeat this structure several more times (the number of repetitions is a hyperparameter you can tune).
- Note that the number of filters grows as we climb up the CNN toward the output layer (it is initially 64, then 128, then 256); it makes sense for it to grow, since the number of low-level features is often fairly low (e.g., small circles, horizontal lines), but there are many different ways to combine them into higher-level features. It is a common practice to double the number of filters after each pooling layer: since a pooling layer divides each spatial dimension by a factor of 2, we can afford to double the number of feature maps in the next layer without fear of exploding the number of parameters, memory usage, or computational load.
- Next is the fully connected network, composed of two hidden dense layers and a dense output layer. Note that we must flatten its inputs, since a dense network expects a 1D array of features for each instance. We also add two dropout layers, with a dropout rate of 50% each, to reduce overfitting.

This CNN reaches over 92% accuracy on the test set. It's not state of the art, but it is pretty good, and clearly much better than what we achieved with dense networks in [Chapter 10](#).

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the ILSVRC [ImageNet challenge](#). In this competition the top-five error rate for image classification fell from over 26% to less than 2.3% in just six years. The top-five error rate is the number of test images for which the system's top five predictions did not include the correct answer. The images are large (256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds). Looking at the evolution of the winning entries is a good way to understand how CNNs work.

We will first look at the classical LeNet-5 architecture (1998), then three of the winners of the ILSVRC challenge: AlexNet (2012), GoogLeNet (2014), and ResNet (2015).

LeNet-5

The **LeNet-5 architecture**¹⁰ is perhaps the most widely known CNN architecture. As mentioned earlier, it was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition (MNIST). It is composed of the layers shown in [Table 14-1](#).

Table 14-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

There are a few extra details to be noted:

- MNIST images are 28×28 pixels, but they are zero-padded to 32×32 pixels and normalized before being fed to the network. The rest of the network does not use any padding, which is why the size keeps shrinking as the image progresses through the network.
- The average pooling layers are slightly more complex than usual: each neuron computes the mean of its inputs, then multiplies the result by a learnable coefficient (one per map) and adds a learnable bias term (again, one per map), then finally applies the activation function.
- Most neurons in C3 maps are connected to neurons in only three or four S2 maps (instead of all six S2 maps). See table 1 (page 8) in the original paper¹⁰ for details.
- The output layer is a bit special: instead of computing the matrix multiplication of the inputs and the weight vector, each neuron outputs the square of the Euclidean distance between its input vector and its weight vector. Each output measures how much the image belongs to a particular digit class. The cross-entropy

¹⁰ Yann LeCun et al., “Gradient-Based Learning Applied to Document Recognition,” *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

cost function is now preferred, as it penalizes bad predictions much more, producing larger gradients and converging faster.

Yann LeCun's [website](#) features great demos of LeNet-5 classifying digits.

AlexNet

The [AlexNet CNN architecture](#)¹¹ won the 2012 ImageNet ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best achieved only 26%! It was developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton. It is similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of one another, instead of stacking a pooling layer on top of each convolutional layer. [Table 14-2](#) presents this architecture.

Table 14-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	—	1,000	—	—	—	Softmax
F10	Fully connected	—	4,096	—	—	—	ReLU
F9	Fully connected	—	4,096	—	—	—	ReLU
S8	Max pooling	256	6×6	3×3	2	valid	—
C7	Convolution	256	13×13	3×3	1	same	ReLU
C6	Convolution	384	13×13	3×3	1	same	ReLU
C5	Convolution	384	13×13	3×3	1	same	ReLU
S4	Max pooling	256	13×13	3×3	2	valid	—
C3	Convolution	256	27×27	5×5	1	same	ReLU
S2	Max pooling	96	27×27	3×3	2	valid	—
C1	Convolution	96	55×55	11×11	4	valid	ReLU
In	Input	3 (RGB)	227×227	—	—	—	—

To reduce overfitting, the authors used two regularization techniques. First, they applied dropout (introduced in [Chapter 11](#)) with a 50% dropout rate during training to the outputs of layers F9 and F10. Second, they performed *data augmentation* by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

¹¹ Alex Krizhevsky et al., “ImageNet Classification with Deep Convolutional Neural Networks,” *Proceedings of the 25th International Conference on Neural Information Processing Systems* 1 (2012): 1097–1105.

Data Augmentation

Data augmentation artificially increases the size of the training set by generating many realistic variants of each training instance. This reduces overfitting, making this a regularization technique. The generated instances should be as realistic as possible: ideally, given an image from the augmented training set, a human should not be able to tell whether it was augmented or not. Simply adding white noise will not help; the modifications should be learnable (white noise is not).

For example, you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set (see [Figure 14-12](#)). This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. For a model that's more tolerant of different lighting conditions, you can similarly generate many images with various contrasts. In general, you can also flip the pictures horizontally (except for text, and other asymmetrical objects). By combining these transformations, you can greatly increase the size of your training set.

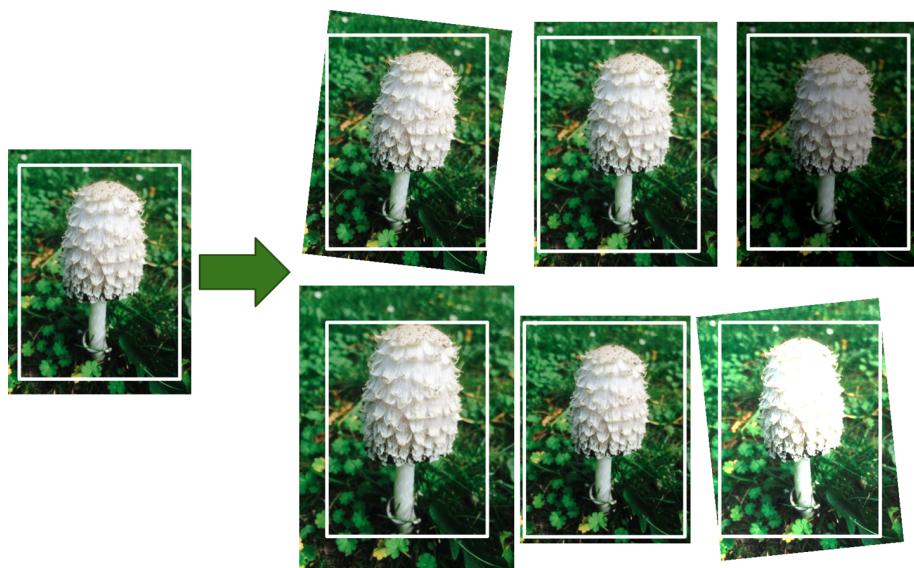


Figure 14-12. Generating new training instances from existing ones

AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, called *local response normalization* (LRN): the most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps (such competitive activation has been observed in biological neurons). This encourages different feature maps to specialize, pushing them apart and forcing

them to explore a wider range of features, ultimately improving generalization. [Equation 14-2](#) shows how to apply LRN.

Equation 14-2. Local response normalization (LRN)

$$b_i = a_i \left(k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2 \right)^{-\beta} \quad \text{with} \quad \begin{cases} j_{\text{high}} = \min \left(i + \frac{r}{2}, f_n - 1 \right) \\ j_{\text{low}} = \max \left(0, i - \frac{r}{2} \right) \end{cases}$$

In this equation:

- b_i is the normalized output of the neuron located in feature map i , at some row u and column v (note that in this equation we consider only neurons located at this row and column, so u and v are not shown).
- a_i is the activation of that neuron after the ReLU step, but before normalization.
- k , α , β , and r are hyperparameters. k is called the *bias*, and r is called the *depth radius*.
- f_n is the number of feature maps.

For example, if $r = 2$ and a neuron has a strong activation, it will inhibit the activation of the neurons located in the feature maps immediately above and below its own.

In AlexNet, the hyperparameters are set as follows: $r = 2$, $\alpha = 0.00002$, $\beta = 0.75$, and $k = 1$. This step can be implemented using the `tf.nn.local_response_normalization()` function (which you can wrap in a `Lambda` layer if you want to use it in a Keras model).

A variant of AlexNet called [ZF Net](#)¹² was developed by Matthew Zeiler and Rob Fergus and won the 2013 ILSVRC challenge. It is essentially AlexNet with a few tweaked hyperparameters (number of feature maps, kernel size, stride, etc.).

GoogLeNet

The [GoogLeNet architecture](#) was developed by Christian Szegedy et al. from Google Research,¹³ and it won the ILSVRC 2014 challenge by pushing the top-five error rate below 7%. This great performance came in large part from the fact that the network was much deeper than previous CNNs (as you'll see in [Figure 14-14](#)). This was made

¹² Matthew D. Zeiler and Rob Fergus, “Visualizing and Understanding Convolutional Networks,” *Proceedings of the European Conference on Computer Vision* (2014): 818–833.

¹³ Christian Szegedy et al., “Going Deeper with Convolutions,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 1–9.

possible by subnetworks called *inception modules*,¹⁴ which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

Figure 14-13 shows the architecture of an inception module. The notation “ $3 \times 3 + 1(S)$ ” means that the layer uses a 3×3 kernel, stride 1, and "same" padding. The input signal is first copied and fed to four different layers. All convolutional layers use the ReLU activation function. Note that the second set of convolutional layers uses different kernel sizes (1×1 , 3×3 , and 5×5), allowing them to capture patterns at different scales. Also note that every single layer uses a stride of 1 and "same" padding (even the max pooling layer), so their outputs all have the same height and width as their inputs. This makes it possible to concatenate all the outputs along the depth dimension in the final *depth concatenation layer* (i.e., stack the feature maps from all four top convolutional layers). This concatenation layer can be implemented in TensorFlow using the `tf.concat()` operation, with `axis=3` (the axis is the depth).

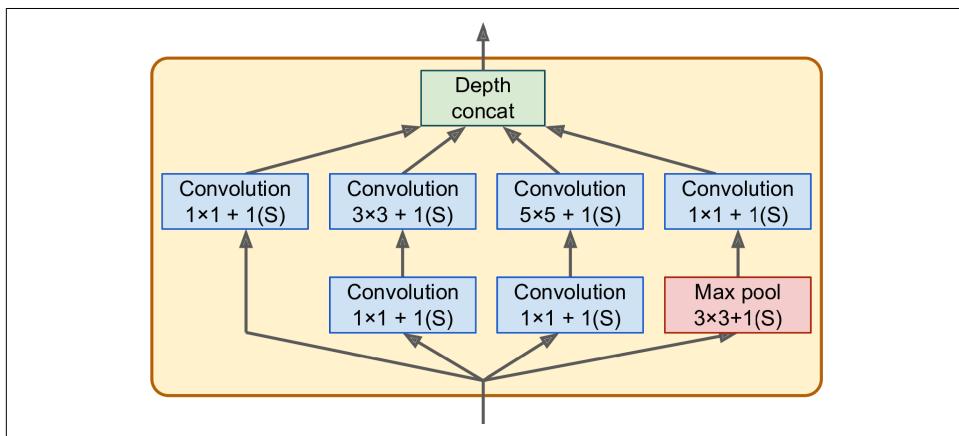


Figure 14-13. Inception module

You may wonder why inception modules have convolutional layers with 1×1 kernels. Surely these layers cannot capture any features because they look at only one pixel at a time? In fact, the layers serve three purposes:

- Although they cannot capture spatial patterns, they can capture patterns along the depth dimension.
- They are configured to output fewer feature maps than their inputs, so they serve as *bottleneck layers*, meaning they reduce dimensionality. This cuts the computa-

¹⁴ In the 2010 movie *Inception*, the characters keep going deeper and deeper into multiple layers of dreams; hence the name of these modules.

tional cost and the number of parameters, speeding up training and improving generalization.

- Each pair of convolutional layers ($[1 \times 1, 3 \times 3]$ and $[1 \times 1, 5 \times 5]$) acts like a single powerful convolutional layer, capable of capturing more complex patterns. Indeed, instead of sweeping a simple linear classifier across the image (as a single convolutional layer does), this pair of convolutional layers sweeps a two-layer neural network across the image.

In short, you can think of the whole inception module as a convolutional layer on steroids, able to output feature maps that capture complex patterns at various scales.



The number of convolutional kernels for each convolutional layer is a hyperparameter. Unfortunately, this means that you have six more hyperparameters to tweak for every inception layer you add.

Now let's look at the architecture of the GoogLeNet CNN (see [Figure 14-14](#)). The number of feature maps output by each convolutional layer and each pooling layer is shown before the kernel size. The architecture is so deep that it has to be represented in three columns, but GoogLeNet is actually one tall stack, including nine inception modules (the boxes with the spinning tops). The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module (in the same order as in [Figure 14-13](#)). Note that all the convolutional layers use the ReLU activation function.

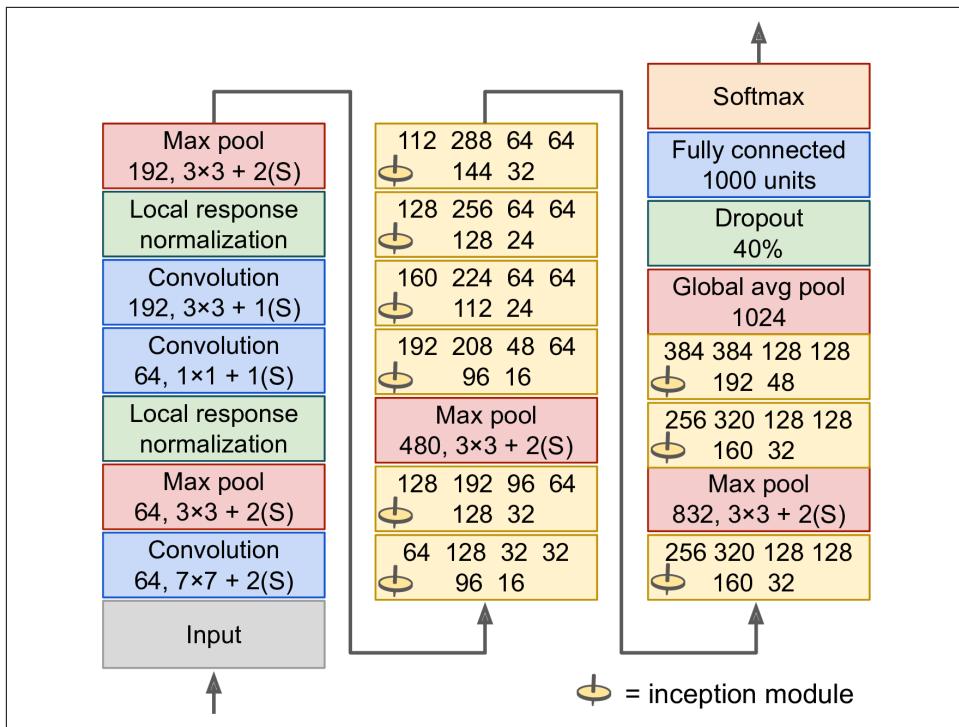


Figure 14-14. GoogLeNet architecture

Let's go through this network:

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load. The first layer uses a large kernel size so that much of the information is preserved.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features (as discussed earlier).
- Two convolutional layers follow, where the first acts like a bottleneck layer. As explained earlier, you can think of this pair as a single smarter convolutional layer.
- Again, a local response normalization layer ensures that the previous layers capture a wide variety of patterns.
- Next, a max pooling layer reduces the image height and width by 2, again to speed up computations.
- Then comes the tall stack of nine inception modules, interleaved with a couple max pooling layers to reduce dimensionality and speed up the net.

- Next, the global average pooling layer outputs the mean of each feature map: this drops any remaining spatial information, which is fine because there was not much spatial information left at that point. Indeed, GoogLeNet input images are typically expected to be 224×224 pixels, so after 5 max pooling layers, each dividing the height and width by 2, the feature maps are down to 7×7 . Moreover, it is a classification task, not localization, so it does not matter where the object is. Thanks to the dimensionality reduction brought by this layer, there is no need to have several fully connected layers at the top of the CNN (like in AlexNet), and this considerably reduces the number of parameters in the network and limits the risk of overfitting.
- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with 1,000 units (since there are 1,000 classes) and a softmax activation function to output estimated class probabilities.

This diagram is slightly simplified: the original GoogLeNet architecture also included two auxiliary classifiers plugged on top of the third and sixth inception modules. They were both composed of one average pooling layer, one convolutional layer, two fully connected layers, and a softmax activation layer. During training, their loss (scaled down by 70%) was added to the overall loss. The goal was to fight the vanishing gradients problem and regularize the network. However, it was later shown that their effect was relatively minor.

Several variants of the GoogLeNet architecture were later proposed by Google researchers, including Inception-v3 and Inception-v4, using slightly different inception modules and reaching even better performance.

VGGNet

The runner-up in the ILSVRC 2014 challenge was **VGGNet**,¹⁵ developed by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (VGG) research lab at Oxford University. It had a very simple and classical architecture, with 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on (reaching a total of just 16 or 19 convolutional layers, depending on the VGG variant), plus a final dense network with 2 hidden layers and the output layer. It used only 3×3 filters, but many filters.

¹⁵ Karen Simonyan and Andrew Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” arXiv preprint arXiv:1409.1556 (2014).

ResNet

Kaiming He et al. won the ILSVRC 2015 challenge using a *Residual Network* (or *ResNet*),¹⁶ that delivered an astounding top-five error rate under 3.6%. The winning variant used an extremely deep CNN composed of 152 layers (other variants had 34, 50, and 101 layers). It confirmed the general trend: models are getting deeper and deeper, with fewer and fewer parameters. The key to being able to train such a deep network is to use *skip connections* (also called *shortcut connections*): the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack. Let's see why this is useful.

When training a neural network, the goal is to make it model a target function $h(\mathbf{x})$. If you add the input \mathbf{x} to the output of the network (i.e., you add a skip connection), then the network will be forced to model $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ rather than $h(\mathbf{x})$. This is called *residual learning* (see Figure 14-15).

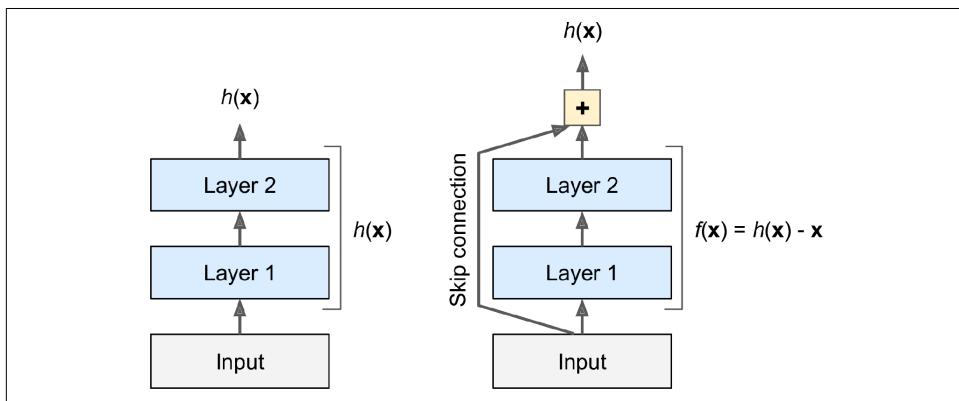


Figure 14-15. Residual learning

When you initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero. If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function. If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.

Moreover, if you add many skip connections, the network can start making progress even if several layers have not started learning yet (see Figure 14-16). Thanks to skip connections, the signal can easily make its way across the whole network. The deep residual network can be seen as a stack of *residual units* (RUs), where each residual unit is a small neural network with a skip connection.

¹⁶ Kaiming He et al., “Deep Residual Learning for Image Recognition,” arXiv preprint arXiv:1512:03385 (2015).

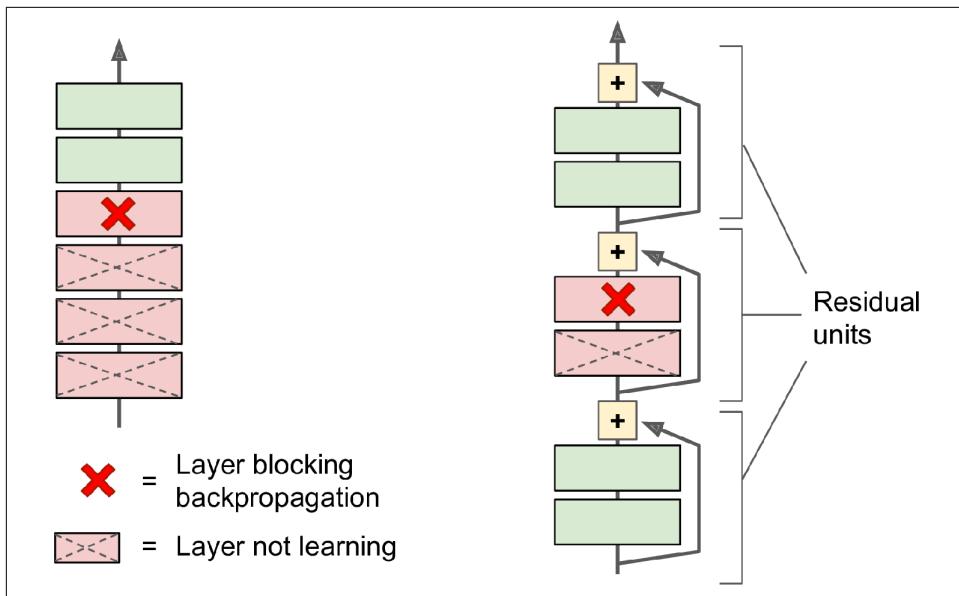


Figure 14-16. Regular deep neural network (left) and deep residual network (right)

Now let's look at ResNet's architecture (see [Figure 14-17](#)). It is surprisingly simple. It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of simple residual units. Each residual unit is composed of two convolutional layers (and no pooling layer!), with Batch Normalization (BN) and ReLU activation, using 3×3 kernels and preserving spatial dimensions (stride 1, "same" padding).

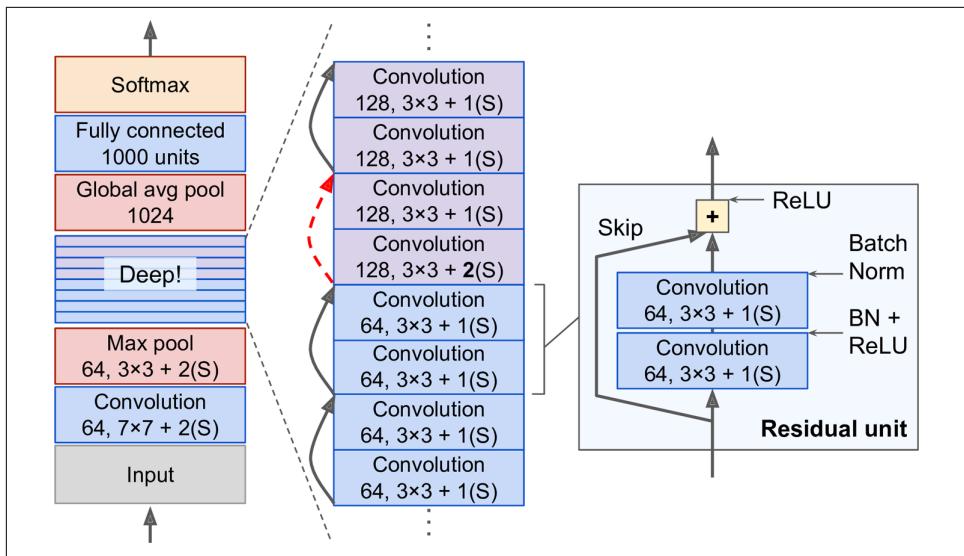


Figure 14-17. ResNet architecture

Note that the number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2). When this happens, the inputs cannot be added directly to the outputs of the residual unit because they don't have the same shape (for example, this problem affects the skip connection represented by the dashed arrow in Figure 14-17). To solve this problem, the inputs are passed through a 1×1 convolutional layer with stride 2 and the right number of output feature maps (see Figure 14-18).

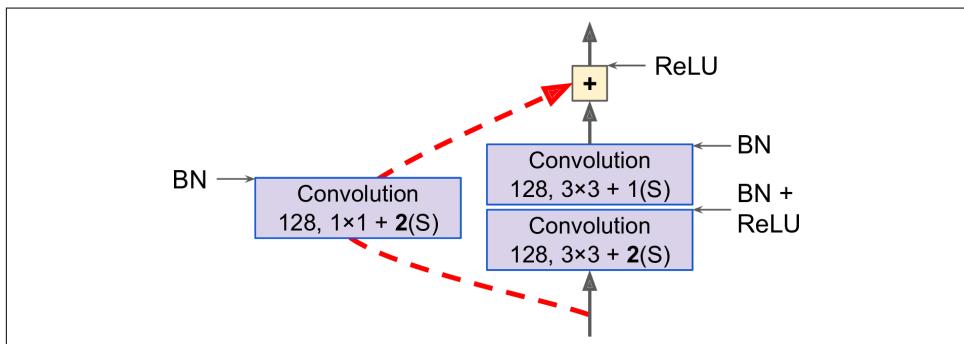


Figure 14-18. Skip connection when changing feature map size and depth

ResNet-34 is the ResNet with 34 layers (only counting the convolutional layers and the fully connected layer)¹⁷ containing 3 residual units that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps. We will implement this architecture later in this chapter.

ResNets deeper than that, such as ResNet-152, use slightly different residual units. Instead of two 3×3 convolutional layers with, say, 256 feature maps, they use three convolutional layers: first a 1×1 convolutional layer with just 64 feature maps (4 times less), which acts as a bottleneck layer (as discussed already), then a 3×3 layer with 64 feature maps, and finally another 1×1 convolutional layer with 256 feature maps (4 times 64) that restores the original depth. ResNet-152 contains 3 such RUs that output 256 maps, then 8 RUs with 512 maps, a whopping 36 RUs with 1,024 maps, and finally 3 RUs with 2,048 maps.



Google's [Inception-v4](#)¹⁸ architecture merged the ideas of GoogLeNet and ResNet and achieved a top-five error rate of close to 3% on ImageNet classification.

Xception

Another variant of the GoogLeNet architecture is worth noting: [Xception](#)¹⁹ (which stands for *Extreme Inception*) was proposed in 2016 by François Chollet (the author of Keras), and it significantly outperformed Inception-v3 on a huge vision task (350 million images and 17,000 classes). Just like Inception-v4, it merges the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a *depthwise separable convolution layer* (or *separable convolution layer* for short²⁰). These layers had been used before in some CNN architectures, but they were not as central as in the Xception architecture. While a regular convolutional layer uses filters that try to simultaneously capture spatial patterns (e.g., an oval) and cross-channel patterns (e.g., mouth + nose + eyes = face), a separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately (see [Figure 14-19](#)). Thus, it is composed of two parts: the first part applies a single spatial filter for each input feature map, then the second part looks

¹⁷ It is a common practice when describing a neural network to count only layers with parameters.

¹⁸ Christian Szegedy et al., "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," arXiv preprint arXiv:1602.07261 (2016).

¹⁹ François Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," arXiv preprint arXiv: 1610.02357 (2016).

²⁰ This name can sometimes be ambiguous, since spatially separable convolutions are often called "separable convolutions" as well.

exclusively for cross-channel patterns—it is just a regular convolutional layer with 1×1 filters.

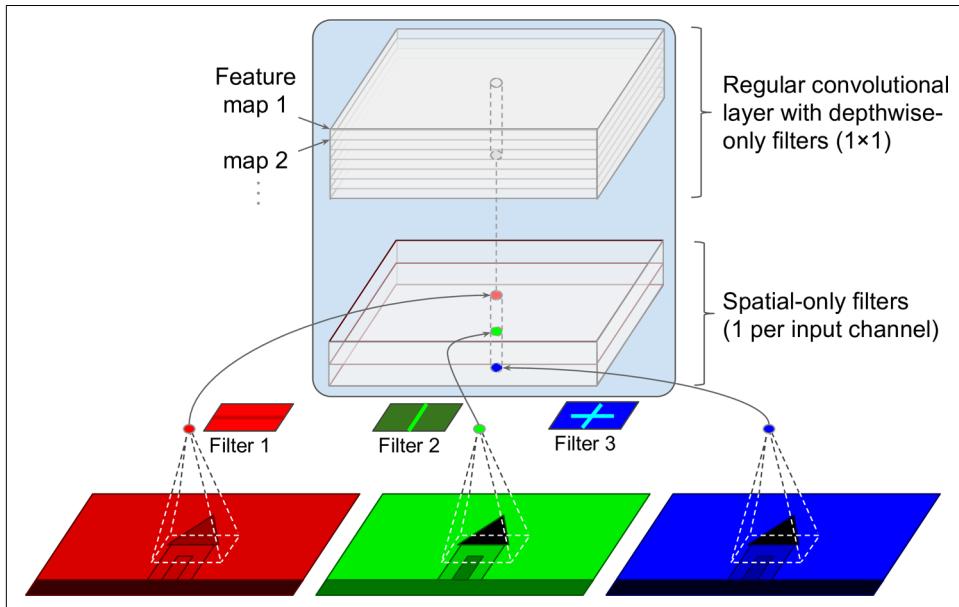


Figure 14-19. Depthwise separable convolutional layer

Since separable convolutional layers only have one spatial filter per input channel, you should avoid using them after layers that have too few channels, such as the input layer (granted, that's what Figure 14-19 represents, but it is just for illustration purposes). For this reason, the Xception architecture starts with 2 regular convolutional layers, but then the rest of the architecture uses only separable convolutions (34 in all), plus a few max pooling layers and the usual final layers (a global average pooling layer and a dense output layer).

You might wonder why Xception is considered a variant of GoogLeNet, since it contains no inception module at all. Well, as we discussed earlier, an inception module contains convolutional layers with 1×1 filters: these look exclusively for cross-channel patterns. However, the convolutional layers that sit on top of them are regular convolutional layers that look both for spatial and cross-channel patterns. So you can think of an inception module as an intermediate between a regular convolutional layer (which considers spatial patterns and cross-channel patterns jointly) and a separable convolutional layer (which considers them separately). In practice, it seems that separable convolutional layers generally perform better.



Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and in general they even perform better, so you should consider using them by default (except after layers with few channels).

The ILSVRC 2016 challenge was won by the CUImage team from the Chinese University of Hong Kong. They used an ensemble of many different techniques, including a sophisticated object-detection system called **GBD-Net**,²¹ to achieve a top-five error rate below 3%. Although this result is unquestionably impressive, the complexity of the solution contrasted with the simplicity of ResNets. Moreover, one year later another fairly simple architecture performed even better, as we will see now.

SENet

The winning architecture in the ILSVRC 2017 challenge was the **Squeeze-and-Excitation Network (SENet)**.²² This architecture extends existing architectures such as inception networks and ResNets, and boosts their performance. This allowed SENet to win the competition with an astonishing 2.25% top-five error rate! The extended versions of inception networks and ResNets are called *SE-Inception* and *SE-ResNet*, respectively. The boost comes from the fact that a SENet adds a small neural network, called an *SE block*, to every unit in the original architecture (i.e., every inception module or every residual unit), as shown in Figure 14-20.

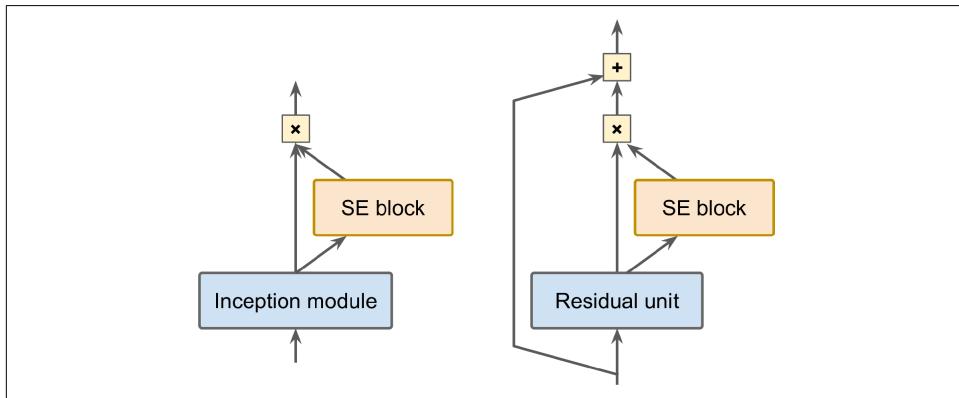


Figure 14-20. SE-Inception module (left) and SE-ResNet unit (right)

²¹ Xingyu Zeng et al., “Crafting GBD-Net for Object Detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, no. 9 (2018): 2109–2123.

²² Jie Hu et al., “Squeeze-and-Excitation Networks,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 7132–7141.

An SE block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension (it does not look for any spatial pattern), and it learns which features are usually most active together. It then uses this information to recalibrate the feature maps, as shown in [Figure 14-21](#). For example, an SE block may learn that mouths, noses, and eyes usually appear together in pictures: if you see a mouth and a nose, you should expect to see eyes as well. So if the block sees a strong activation in the mouth and nose feature maps, but only mild activation in the eye feature map, it will boost the eye feature map (more accurately, it will reduce irrelevant feature maps). If the eyes were somewhat confused with something else, this feature map recalibration will help resolve the ambiguity.

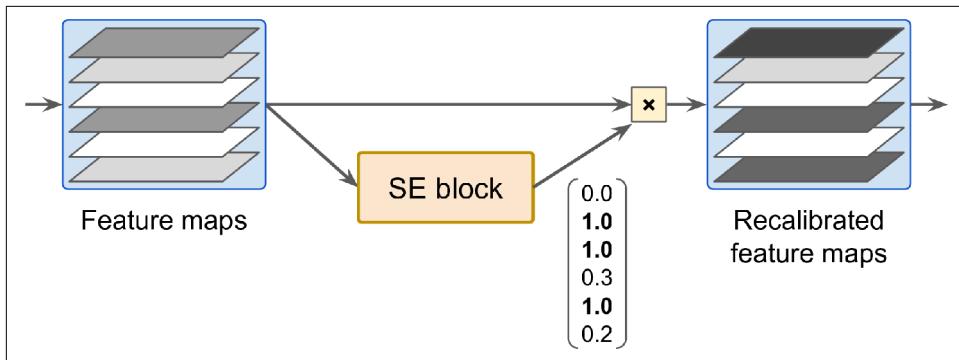


Figure 14-21. An SE block performs feature map recalibration

An SE block is composed of just three layers: a global average pooling layer, a hidden dense layer using the ReLU activation function, and a dense output layer using the sigmoid activation function (see [Figure 14-22](#)).

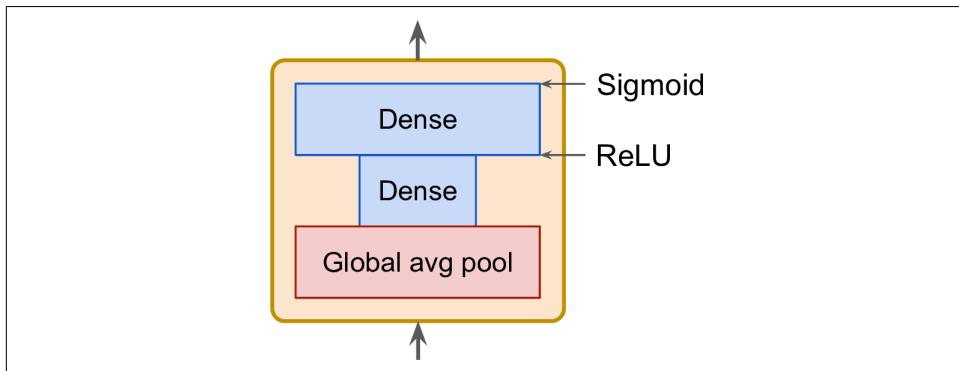


Figure 14-22. SE block architecture

As earlier, the global average pooling layer computes the mean activation for each feature map: for example, if its input contains 256 feature maps, it will output 256