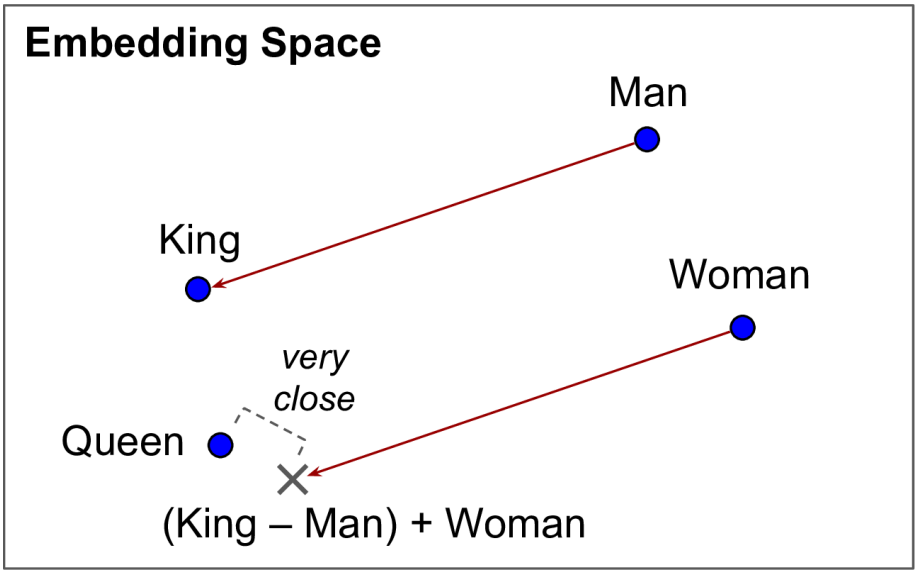


*Figure 13-5. Word Embeddings*

Let's go back to the Features API. Here is how you could encode the `ocean_proximity` categories as 2D embeddings:

```
ocean_proximity_embed = tf.feature_column.embedding_column(ocean_proximity,
                                                           dimension=2)
```

Each of the five `ocean_proximity` categories will now be represented as a 2D vector. These vectors are stored in an *embedding matrix* with one row per category, and one column per embedding dimension, so in this example it is a 5×2 matrix. When an embedding column is given a category index as input (say, 3, which corresponds to the category `"NEAR BAY"`), it just performs a lookup in the embedding matrix and returns the corresponding row (say, `[0.331, 0.190]`). Unfortunately, the embedding matrix can be quite large, especially when you have a large vocabulary: if this is the case, the model can only learn good representations for the categories for which it has sufficient training data. To reduce the size of the embedding matrix, you can of course try lowering the `dimension` hyperparameter, but if you reduce this parameter too much, the representations may not be as good. Another option is to reduce the vocabulary size (e.g., if you are dealing with text, you can try dropping the rare words from the vocabulary, and replace them all with a token like `"<unknown>"` or `"<UNK>"`). If you are using hash buckets, you can also try reducing the `hash_bucket_size` (but not too much, or else you will get collisions).

If there are no pretrained embeddings that you can reuse for the task you are trying to tackle, and if you do not have enough training data to learn them, then you can try to learn them on some auxiliary task for which it is easier to obtain plenty of training data. After that, you can reuse the trained embeddings for your main task.

## Using Feature Columns for Parsing

Let's suppose you have created feature columns for each of your input features, as well as for the target. What can you do with them? Well, for one you can pass them to the `make_parse_example_spec()` function to generate feature descriptions (so you don't have to do it manually, as we did earlier):

```
columns = [bucketized_age, ....., median_house_value] # all features + target
feature_descriptions = tf.feature_column.make_parse_example_spec(columns)
```

You don't always have to create a separate feature column for each and every feature. For example, instead of having 2 numerical feature columns, you could choose to have a single 2D column: just set `shape=[2]` when calling `numerical_column()`.

You can then create a function that parses serialized examples using these feature descriptions, and separates the target column from the input features:

```
def parse_examples(serialized_examples):
    examples = tf.io.parse_example(serialized_examples, feature_descriptions)
    targets = examples.pop("median_house_value") # separate the targets
    return examples, targets
```

Next, you can create a `TFRecordDataset` that will read batches of serialized examples (assuming the TFRecord file contains serialized `Example` protobufs with the appropriate features):

```
batch_size = 32
dataset = tf.data.TFRecordDataset(["my_data_with_features.tfrecords"])
dataset = dataset.repeat().shuffle(10000).batch(batch_size).map(parse_examples)
```

## Using Feature Columns in Your Models

Feature columns can also be used directly in your model, to convert all your input features into a single dense vector which the neural network can then process. For this, all you need to do is add a `keras.layers.DenseFeatures` layer as the first layer in your model, passing it the list of feature columns (excluding the target column):

```
columns_without_target = columns[:-1]
model = keras.models.Sequential([
    keras.layers.DenseFeatures(feature_columns=columns_without_target),
```

```
    keras.layers.Dense(1)
])
model.compile(loss="mse", optimizer="sgd", metrics=["accuracy"])
steps_per_epoch = len(X_train) // batch_size
history = model.fit(dataset, steps_per_epoch=steps_per_epoch, epochs=5)
```

The `DenseFeatures` layer will take care of converting every input feature to a dense representation, and it will also apply any extra transformation we specified, such as scaling the `housing_median_age` using the `normalizer_fn` function we provided. You can take a closer look at what the `DenseFeatures` layer does by calling it directly:

```
>>> some_columns = [ocean_proximity_embed, bucketized_income]
>>> dense_features = keras.layers.DenseFeatures(some_columns)
>>> dense_features({
...     "ocean_proximity": [["NEAR OCEAN"], ["INLAND"], ["INLAND"]],
...     "median_income": [[3.], [7.2], [1.]]
... })
...
<tf.Tensor: id=559790, shape=(3, 7), dtype=float32, numpy=
array([[ 0. , 0. , 1. , 0. , 0. ,-0.36277947 , 0.30109018],
       [ 0. , 0. , 0. , 0. , 1. , 0.22548223 , 0.33142096],
       [ 1. , 0. , 0. , 0. , 0. , 0.22548223 , 0.33142096]], dtype=float32)>
```

In this example, we create a `DenseFeatures` layer with just two columns, and we call it with some data, in the form of a dictionary of features. In this case, since the `bucketized_income` column relies on the `median_income` column, the dictionary must include the `"median_income"` key, and similarly since the `ocean_proximity_embed` column is based on the `ocean_proximity` column, the dictionary must include the `"ocean_proximity"` key. Columns are handled in alphabetical order, so first we look at the bucketized income column (its name is the same as the `median_income` column name, plus `"_bucketized"`). The incomes 3, 7.2 and 1 get mapped respectively to category 2 (for incomes between 1.5 and 3), category 0 (for incomes below 1.5), and category 4 (for incomes greater than 6). Then these category IDs get one-hot encoded: category 2 gets encoded as `[0., 0., 1., 0., 0.]` and so on (note that bucketized columns get one-hot encoded by default, no need to call `indicator_column()`). Now on to the `ocean_proximity_embed` column. The `"NEAR OCEAN"` and `"INLAND"` categories just get mapped to their respective embeddings (which were initialized randomly). The resulting tensor is the concatenation of the one-hot vectors and the embeddings.

Now you can feed all kinds of features to a neural network, including numerical features, categorical features, and even text (by splitting the text into words, then using word embedding)! However, performing all the preprocessing on the fly can slow down training. Let's see how this can be improved.

## 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. Tools like Apache Beam let you run efficient data processing pipelines over large amounts of data, even distributed across multiple servers, so why not use it to preprocess all the training data? 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: 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 it runs in a web browser? Once again, you will need to write some preprocessing 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. It 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 preprocessed by your Apache Beam code), and before deploying it to your app or the browser, add an extra input layer to take care of preprocessing on the fly (either by writing a custom layer or by using a `DenseFeatures` layer). That's definitely better, since now you just have two versions of your preprocessing code: the Apache Beam code and the preprocessing layer's 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 define your preprocessing function just once (in Python), by using TF Transform functions for scaling, bucketizing, crossing features, 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 is 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 - tft.mean(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 contains all the necessary statistics computed by Apache Beam (the mean, standard deviation, and vocabulary), simply included as constants.

At the time of this writing, TF Transform only supports TensorFlow 1. Moreover, Apache Beam only has partial support for Python 3. That said, both these limitations will likely be fixed by the time your read this.

With the Data API, TFRecords, the Features API and TF Transform, you can build highly scalable input pipelines for training, and also 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 trivial to download common datasets, from small ones like MNIST or Fashion MNIST, to huge datasets like ImageNet[11] (you will need quite a bit of disk space!). The list includes image datasets, text datasets (including translation datasets), audio and video datasets, and more. You can visit *https://homl.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 all you need to do is 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,

11 At the time of writing, TFDS requires you to download a few files manually for ImageNet (for legal reasons), but this will hopefully get resolved soon.

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 repeating, batching and prefetching), and you're ready to train your model. Here is a simple example:

```
mnist_train = mnist_train.repeat(5).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```

In general, `load()` returns a shuffled training set, so there's no need to shuffle it 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 2 elements (again, the features and the labels). You could transform the dataset using the `map()` method, like this:

```
mnist_train = mnist_train.repeat(5).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

Or you can just 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 the dataset can be passed directly to your tf.keras model:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].repeat().prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, steps_per_epoch=60000 // 32, 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.

CHAPTER 14

# Deep Computer Vision Using Convolutional Neural Networks

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 14 in the final release of the book.

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, or *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* or *natural language processing* (NLP); however, we will focus on visual applications for now.

In this chapter we will present 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, and discuss 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 1958[1] and 1959[2] (and a few years later on monkeys[3]), giving crucial insights on 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 "Single Unit Activity in Striate Cortex of Unrestrained Cats," D. Hubel and T. Wiesel (1958).

2 "Receptive Fields of Single Neurones in the Cat's Striate Cortex," D. Hubel and T. Wiesel (1959).

3 "Receptive Fields and Functional Architecture of Monkey Striate Cortex," D. Hubel and T. Wiesel (1968).

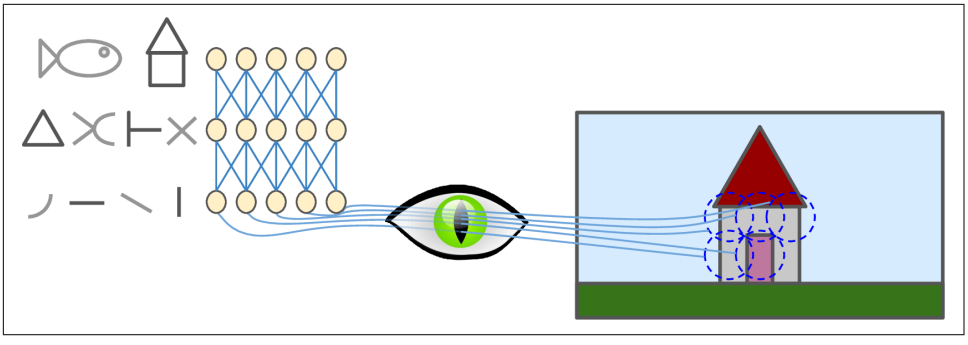

*Figure 14-1. Local receptive fields in the visual cortex*

These studies of the visual cortex inspired the neocognitron, introduced in 1980,[4] which gradually evolved into what we now call *convolutional neural networks*. An important milestone was a 1998 paper[5] by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, which introduced the famous *LeNet-5* architecture, widely used 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 regular 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 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 "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position," K. Fukushima (1980).

5 "Gradient-Based Learning Applied to Document Recognition," Y. LeCun et al. (1998).

## Convolutional Layer

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 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.

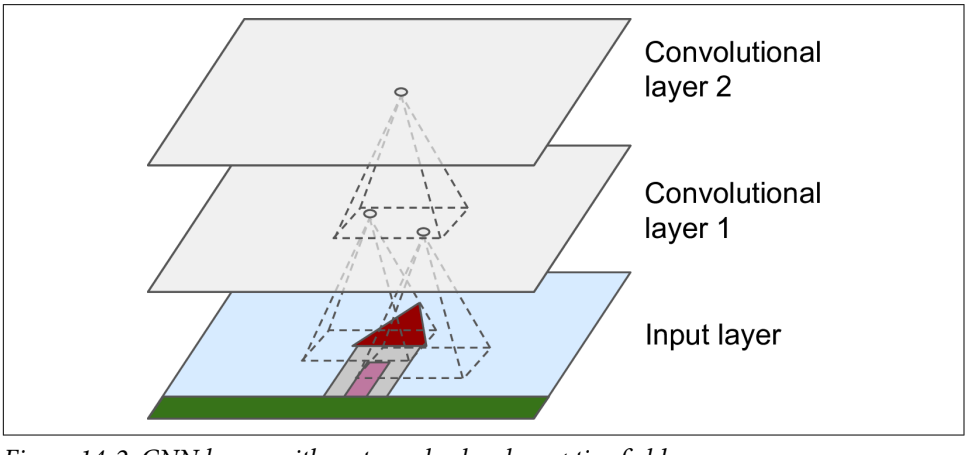


*Figure 14-2. CNN layers with rectangular local receptive fields*

Until now, all multilayer neural networks we looked at 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. Now each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

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 com-

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).