

# Unsupervised Learning Techniques

Yann LeCun, Turing Award winner and Meta's Chief AI Scientist, famously said that “if intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake” (NeurIPS 2016). In other words, there is a huge potential in unsupervised learning that we have only barely started to sink our teeth into. Indeed, the vast majority of the available data is unlabeled: we have the input features  $X$ , but we do not have the labels  $y$ .

Say you want to create a system that will take a few pictures of each item on a manufacturing production line and detect which items are defective. You can fairly easily create a system that will take pictures automatically, and this might give you thousands of pictures every day. You can then build a reasonably large dataset in just a few weeks. But wait, there are no labels! If you want to train a regular binary classifier that will predict whether an item is defective or not, you will need to label every single picture as “defective” or “normal”. This will generally require human experts to sit down and manually go through all the pictures. This is a long, costly, and tedious task, so it will usually only be done on a small subset of the available pictures. As a result, the labeled dataset will be quite small, and the classifier’s performance will be disappointing. Moreover, every time the company makes any change to its products, the whole process will need to be started over from scratch. Wouldn’t it be great if the algorithm could just exploit the unlabeled data without needing humans to label every picture? Enter unsupervised learning.

In [Chapter 7](#) we looked at the most common unsupervised learning task: dimensionality reduction. In this chapter we will look at a few more unsupervised tasks:

## *Clustering*

The goal is to group similar instances together into *clusters*. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search

engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.

#### *Anomaly detection (also called outlier detection)*

The objective is to learn what “normal” data looks like, and then use that to detect abnormal instances. These instances are called *anomalies*, or *outliers*, while the normal instances are called *inliers*. Anomaly detection is useful in a wide variety of applications, such as fraud detection, detecting defective products in manufacturing, identifying new trends in time series, or removing outliers from a dataset before training another model, which can significantly improve the performance of the resulting model.

#### *Density estimation*

This is the task of estimating the *probability density function* (PDF) of the random process that generated the dataset.<sup>1</sup> Density estimation is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualization.

Ready for some cake? We will start with two clustering algorithms, *k*-means and DBSCAN, then we’ll discuss Gaussian mixture models and see how they can be used for density estimation, clustering, and anomaly detection.

## Clustering Algorithms: k-means and DBSCAN

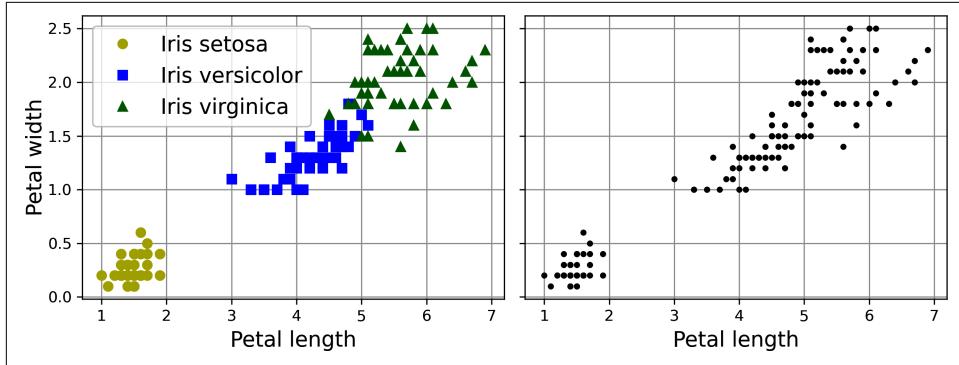
As you enjoy a hike in the mountains, you stumble upon a plant you have never seen before. You look around and you notice a few more. They are not identical, yet they are sufficiently similar for you to know that they most likely belong to the same species (or at least the same genus). You may need a botanist to tell you what species that is, but you certainly don’t need an expert to identify groups of similar-looking objects. This is called *clustering*: it is the task of identifying similar instances and assigning them to *clusters*, or groups of similar instances.

Just like in classification, each instance gets assigned to a group. However, unlike classification, clustering is an unsupervised task, there are no labels, so the algorithm needs to figure out on its own how to group instances. Consider Figure 8-1: on the left is the iris dataset (introduced in Chapter 4), where each instance’s species (i.e., its class) is represented with a different marker. It is a labeled dataset, for which classification algorithms such as logistic regression, SVMs, or random forest classifiers are well suited. On the right is the same dataset, but without the labels, so you cannot use a classification algorithm anymore. This is where clustering algorithms step in: many of them can easily detect the lower-left cluster. It is also quite easy to see with our own

---

<sup>1</sup> If you are not familiar with probability theory, I highly recommend the free online classes by Khan Academy.

eyes, but it is not so obvious that the upper-right cluster is composed of two distinct subclusters. That said, the dataset has two additional features (sepal length and width) that are not represented here, and clustering algorithms can make good use of all features, so in fact they identify the three clusters fairly well (e.g., using a Gaussian mixture model, only 5 instances out of 150 are assigned to the wrong cluster).



*Figure 8-1. Classification (left) versus clustering (right): in clustering, the dataset is unlabeled so the algorithm must identify the clusters without guidance*

Clustering is used in a wide variety of applications, including:

#### *Customer segmentation*

You can cluster your customers based on their purchases and their activity on your website. This is useful to understand who your customers are and what they need, so you can adapt your products and marketing campaigns to each segment. For example, customer segmentation can be useful in *recommender systems* to suggest content that other users in the same cluster enjoyed.

#### *Data analysis*

When you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.

#### *Dimensionality reduction*

Once a dataset has been clustered, it is usually possible to measure each instance's *affinity* with each cluster; affinity is any measure of how well an instance fits into a cluster. Each instance's feature vector  $\mathbf{x}$  can then be replaced with the vector of its cluster affinities. If there are  $k$  clusters, then this vector is  $k$ -dimensional. The new vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.

### *Feature engineering*

The cluster affinities can often be useful as extra features. For example, we used  $k$ -means in [Chapter 2](#) to add geographic cluster affinity features to the California housing dataset, and they helped us get better performance.

### *Anomaly detection (also called outlier detection)*

Any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if you have clustered the users of your website based on their behavior, you can detect users with unusual behavior, such as an unusual number of requests per second.

### *Semi-supervised learning*

If you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster. This technique can greatly increase the number of labels available for a subsequent supervised learning algorithm, and thus improve its performance.

### *Search engines*

Some search engines let you search for images that are similar to a reference image. To build such a system, you would first apply a clustering algorithm to all the images in your database; similar images would end up in the same cluster. Then when a user provides a reference image, all you'd need to do is use the trained clustering model to find this image's cluster, and you could then simply return all the images from this cluster.

### *Image segmentation*

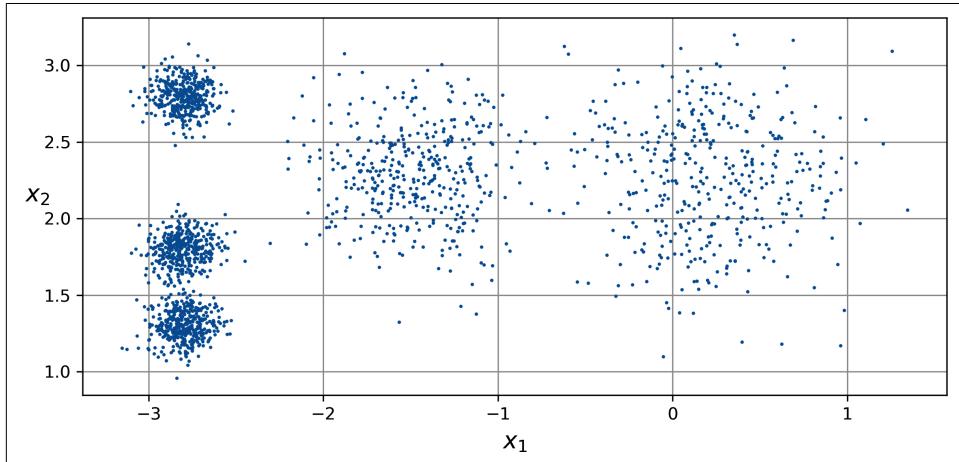
By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to considerably reduce the number of different colors in an image. Image segmentation is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.

There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms look for instances centered around a particular point, called a *centroid*. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

In this section, we will look at two popular clustering algorithms,  $k$ -means and DBSCAN, and explore some of their applications, such as nonlinear dimensionality reduction, semi-supervised learning, and anomaly detection.

## k-Means Clustering

Consider the unlabeled dataset represented in [Figure 8-2](#): you can clearly see five blobs of instances. The  $k$ -means algorithm is a simple algorithm capable of clustering this kind of dataset very quickly and efficiently, often in just a few iterations. It was proposed by Stuart Lloyd at Bell Labs in 1957 as a technique for pulse-code modulation, but it was only [published](#) outside of the company in 1982.<sup>2</sup> In 1965, Edward W. Forgy had published virtually the same algorithm, so  $k$ -means is sometimes referred to as the Lloyd–Forgy algorithm.



*Figure 8-2. An unlabeled dataset composed of five blobs of instances*

Let's train a  $k$ -means clusterer on this dataset. It will try to find each blob's center and assign each instance to the closest blob:

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

X, y = make_blobs([...]) # make the blobs: y contains the cluster IDs, but we
                        # will not use them; that's what we want to predict
k = 5
kmeans = KMeans(n_clusters=k, random_state=42)
y_pred = kmeans.fit_predict(X)
```

Note that you have to specify the number of clusters  $k$  that the algorithm must find. In this example, it is pretty obvious from looking at the data that  $k$  should be set to 5, but in general it is not that easy. We will discuss this shortly.

<sup>2</sup> Stuart P. Lloyd, "Least Squares Quantization in PCM", *IEEE Transactions on Information Theory* 28, no. 2 (1982): 129–137.

Each instance will be assigned to one of the five clusters. In the context of clustering, an instance's *label* is the index of the cluster to which the algorithm assigns this instance; this is not to be confused with the class labels in classification, which are used as targets (remember that clustering is an unsupervised learning task). The `KMeans` instance preserves the predicted labels of the instances it was trained on, available via the `labels_` instance variable:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

We can also take a look at the five centroids that the algorithm found:

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

You can easily assign new instances to the cluster whose centroid is closest:

```
>>> import numpy as np
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

If you plot the cluster's decision boundaries, you get a Voronoi tessellation: see [Figure 8-3](#), where each centroid is represented with an  $\otimes$ .

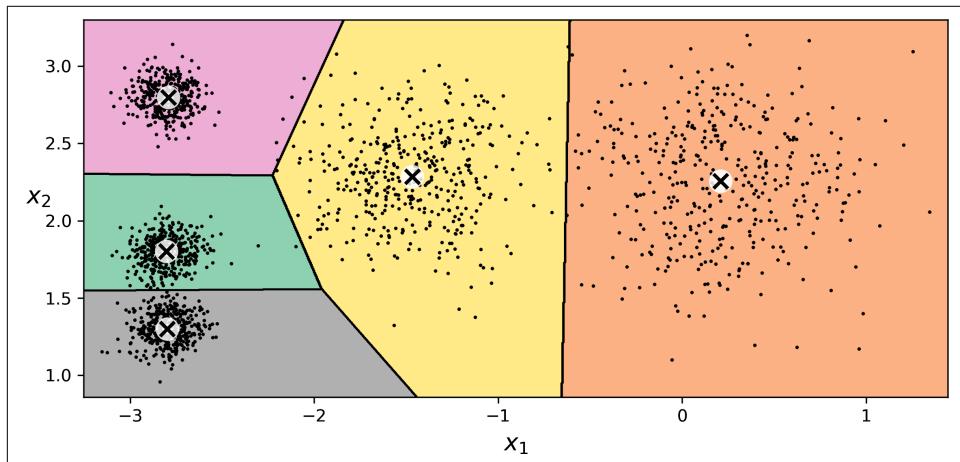


Figure 8-3. *k*-means decision boundaries (Voronoi tessellation)

The vast majority of the instances were clearly assigned to the appropriate cluster, but a few instances were probably mislabeled, especially near the boundary between the top-left cluster and the central cluster. Indeed, the  $k$ -means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid.

Instead of assigning each instance to a single cluster, which is called *hard clustering*, it can be useful to give each instance a score per cluster, which is called *soft clustering*. The score can be the distance between the instance and the centroid or a similarity score (or affinity), such as the Gaussian radial basis function we used in [Chapter 2](#). In the `KMeans` class, the `transform()` method measures the distance from each instance to every centroid:

```
>>> kmeans.transform(X_new).round(2)
array([[2.81, 0.33, 2.9 , 1.49, 2.89],
       [5.81, 2.8 , 5.85, 4.48, 5.84],
       [1.21, 3.29, 0.29, 1.69, 1.71],
       [0.73, 3.22, 0.36, 1.55, 1.22]])
```

In this example, the first instance in `X_new` is located at a distance of about 2.81 from the first centroid, 0.33 from the second centroid, 2.90 from the third centroid, 1.49 from the fourth centroid, and 2.89 from the fifth centroid. If you have a high-dimensional dataset and you transform it this way, you end up with a  $k$ -dimensional dataset: this transformation can be a very efficient nonlinear dimensionality reduction technique. Alternatively, you can use these distances as extra features to train another model, as in [Chapter 2](#).

## The k-means algorithm

So, how does the algorithm work? Well, suppose you were given the centroids. You could easily label all the instances in the dataset by assigning each of them to the cluster whose centroid is closest. Conversely, if you were given all the instance labels, you could easily locate each cluster's centroid by computing the mean of the instances in that cluster. But you are given neither the labels nor the centroids, so how can you proceed? Start by placing the centroids randomly (e.g., by picking  $k$  instances at random from the dataset and using their locations as centroids). Then label the instances, update the centroids, label the instances, update the centroids, and so on until the centroids stop moving. The algorithm is guaranteed to converge in a finite number of steps (usually quite small). That's because the mean squared distance between the instances and their closest centroids can only go down at each step, and since it cannot be negative, it's guaranteed to converge.

You can see the algorithm in action in [Figure 8-4](#): the centroids are initialized randomly (top left), then the instances are labeled (top right), then the centroids are updated (center left), the instances are relabeled (center right), and so on. As you can

see, in just three iterations the algorithm has reached a clustering that seems close to optimal.



The computational complexity of the algorithm is generally linear with regard to the number of instances  $m$ , the number of clusters  $k$ , and the number of dimensions  $n$ . However, this is only true when the data has a clustering structure. If it does not, then in the worst-case scenario the complexity can increase exponentially with the number of instances. In practice, this rarely happens, and  $k$ -means is generally one of the fastest clustering algorithms.

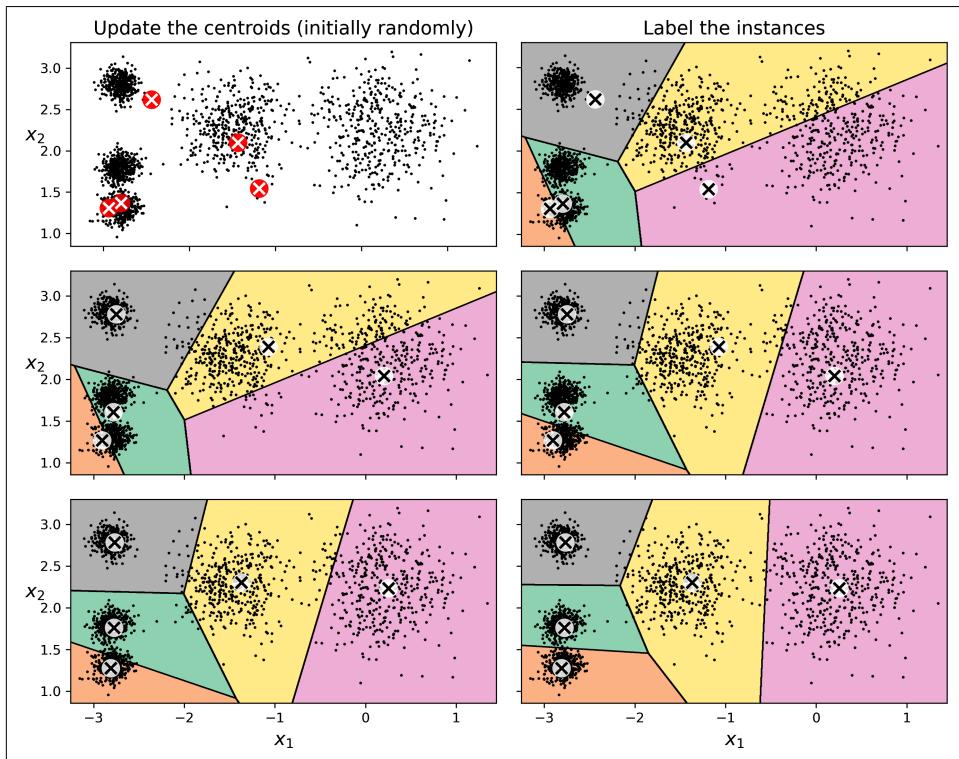


Figure 8-4. The  $k$ -means algorithm

Although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): whether it does or not depends on the centroid initialization. Figure 8-5 shows two suboptimal solutions that the algorithm can converge to if you are not lucky with the random initialization step.

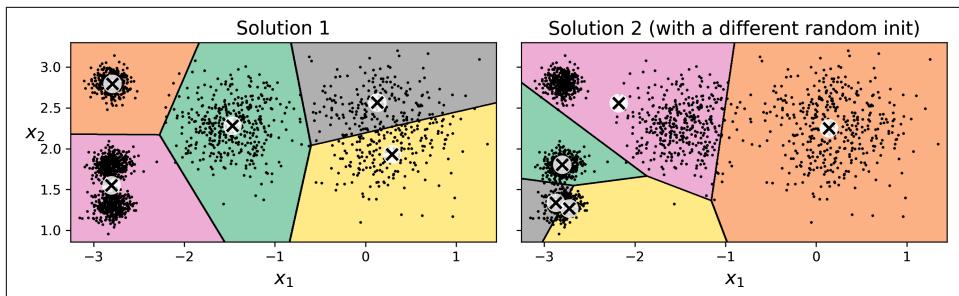


Figure 8-5. Suboptimal solutions due to unlucky centroid initializations

Let's take a look at a few ways you can mitigate this risk by improving the centroid initialization.

### Centroid initialization methods

If you happen to know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set the `init` hyperparameter to a NumPy array containing the list of centroids:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, random_state=42)
kmeans.fit(X)
```

Another solution is to run the algorithm multiple times with different random initializations and keep the best solution. The number of random initializations is controlled by the `n_init` hyperparameter: by default it is equal to 10 when using `init="random"`, which means that the whole algorithm described earlier runs 10 times when you call `fit()`, and Scikit-Learn keeps the best solution. But how exactly does it know which solution is the best? It uses a performance metric! That metric is called the model's *inertia*, which is defined in [Equation 8-1](#).

[Equation 8-1](#). A model's inertia is the sum of all squared distances between each instance  $\mathbf{x}^{(i)}$  and the closest centroid  $\mathbf{c}^{(i)}$  predicted by the model

$$\text{inertia} = \sum_i \| \mathbf{x}^{(i)} - \mathbf{c}^{(i)} \|^2$$

The inertia is roughly equal to 219.6 for the model on the left in [Figure 8-5](#), 600.4 for the model on the right in [Figure 8-5](#), and only 211.6 for the model in [Figure 8-3](#). The `KMeans` class runs the initialization algorithm `n_init` times and keeps the model with the lowest inertia. In this example, the model in [Figure 8-3](#) will be selected (unless we are very unlucky with `n_init` consecutive random initializations). If you are curious, a model's inertia is accessible via the `inertia_` instance variable:

```
>>> kmeans.inertia_
211.59853725816828
```

The `score()` method returns the negative inertia (it's negative because a predictor's `score()` method must always respect Scikit-Learn's "greater is better" rule—if a predictor is better than another, its `score()` method should return a greater score):

```
>>> kmeans.score(X)
-211.59853725816828
```

An important improvement to the  $k$ -means algorithm,  $k$ -means++, was proposed in a [2006 paper](#) by David Arthur and Sergei Vassilvitskii.<sup>3</sup> They introduced a smarter initialization step that tends to select centroids that are distant from one another. This change makes the  $k$ -means algorithm much more likely to locate all important clusters, and less likely to converge to a suboptimal solution (just like spreading out fishing boats increases the chance of locating more schools of fish). The paper showed that the additional computation required for the smarter initialization step is well worth it because it makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution. The  $k$ -means++ initialization algorithm works like this:

1. Take one centroid  $c^{(1)}$ , chosen uniformly at random from the dataset.
2. Take a new centroid  $c^{(i)}$ , choosing an instance  $x^{(i)}$  with probability  $D(x^{(i)})^2 / \sum_{j=1}^m D(x^{(j)})^2$ , where  $D(x^{(i)})$  is the distance between the instance  $x^{(i)}$  and the closest centroid that was already chosen. This probability distribution ensures that instances farther away from already chosen centroids are much more likely to be selected as centroids.
3. Repeat the previous step until all  $k$  centroids have been chosen.

When you set `init="k-means++"` (which is the default), the `KMeans` class actually uses a variant of  $k$ -means++ called *greedy k-means++*: instead of sampling a single centroid at each iteration, it samples multiple and picks the best one. When using this algorithm, `n_init` defaults to 1.

## Accelerated k-means and mini-batch k-means

Another improvement to the  $k$ -means algorithm was proposed in a [2003 paper](#) by Charles Elkan.<sup>4</sup> On some large datasets with many clusters, the algorithm can be accelerated by avoiding many unnecessary distance calculations. Elkan achieved this

---

<sup>3</sup> David Arthur and Sergei Vassilvitskii, "k-Means++: The Advantages of Careful Seeding", *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (2007): 1027–1035.

<sup>4</sup> Charles Elkan, "Using the Triangle Inequality to Accelerate k-Means", *Proceedings of the 20th International Conference on Machine Learning* (2003): 147–153.

by exploiting the triangle inequality (i.e., that a straight line is always the shortest distance between two points<sup>5</sup>) and by keeping track of lower and upper bounds for distances between instances and centroids. However, Elkan's algorithm does not always accelerate training, and sometimes it can even slow down training significantly; it depends on the dataset. Still, if you want to give it a try, set `algorithm="elkan"`.

Yet another important variant of the  $k$ -means algorithm was proposed in a [2010 paper](#) by David Sculley.<sup>6</sup> Instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration. This speeds up the algorithm and makes it possible to cluster huge datasets that do not fit in memory. Scikit-Learn implements this algorithm in the `MiniBatchKMeans` class, which you can use just like the `KMeans` class:

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5, random_state=42)
minibatch_kmeans.fit(X)
```

If the dataset does not fit in memory, the simplest option is to use the `memmap` class, as we did for incremental PCA in [Chapter 7](#). Alternatively, you can pass one mini-batch at a time to the `partial_fit()` method, but this will require much more work, since you will need to perform multiple initializations and select the best one yourself.

## Finding the optimal number of clusters

So far, we've set the number of clusters  $k$  to 5 because it was obvious by looking at the data that this was the correct number of clusters. But in general, it won't be so easy to know how to set  $k$ , and the result might be quite bad if you set it to the wrong value. As you can see in [Figure 8-6](#), for this dataset setting  $k$  to 3 or 8 results in fairly bad models.

You might be thinking that you could just pick the model with the lowest inertia. Unfortunately, it is not that simple. The inertia for  $k = 3$  is about 653.2, which is much higher than for  $k = 5$  (211.7). But with  $k = 8$ , the inertia is just 127.1. The inertia is not a good performance metric when trying to choose  $k$  because it keeps getting lower as we increase  $k$ . Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. Let's plot the inertia as a function of  $k$ . When we do this, the curve often contains an inflection point called the *elbow* (see [Figure 8-7](#)).

---

<sup>5</sup> The triangle inequality is  $AC \leq AB + BC$ , where A, B, and C are three points and AB, AC, and BC are the distances between these points.

<sup>6</sup> David Sculley, "Web-Scale K-Means Clustering", *Proceedings of the 19th International Conference on World Wide Web* (2010): 1177–1178.

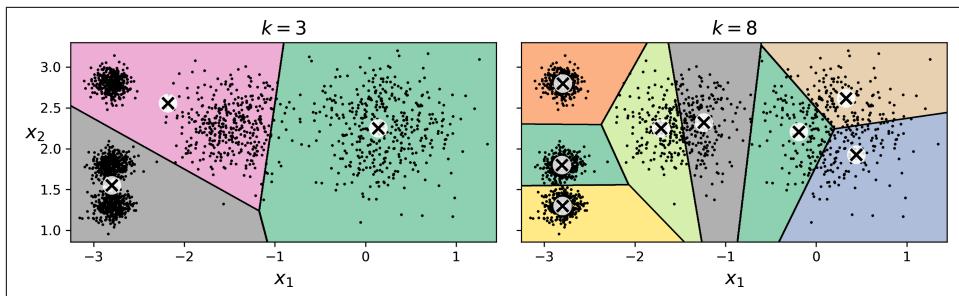


Figure 8-6. Bad choices for the number of clusters: when  $k$  is too small, separate clusters get merged (left), and when  $k$  is too large, some clusters get chopped into multiple pieces (right)

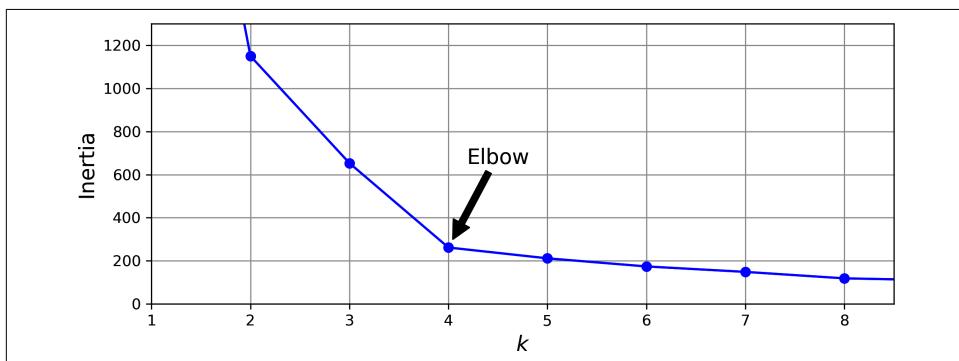


Figure 8-7. Plotting the inertia as a function of the number of clusters  $k$

As you can see, the inertia drops very quickly as we increase  $k$  up to 4, but then it decreases much more slowly as we keep increasing  $k$ . This curve has roughly the shape of an arm, and there is an elbow at  $k = 4$ . So, if we did not know better, we might think 4 was a good choice: any lower value would be dramatic, while any higher value would not help much, and we might just be splitting perfectly good clusters in half for no good reason.

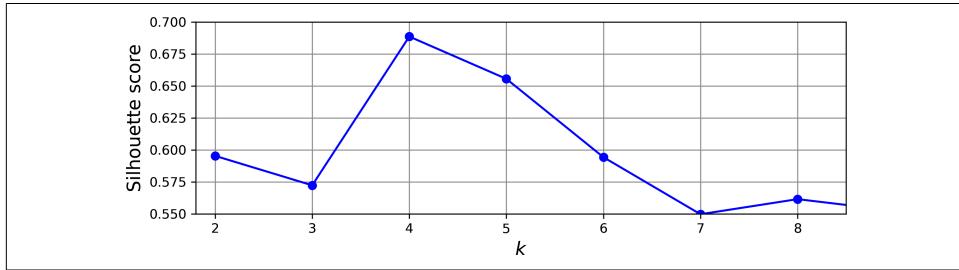
This technique for choosing the best value for the number of clusters is rather coarse. A more precise (but also more computationally expensive) approach is to use the *silhouette score*, which is the mean *silhouette coefficient* over all the instances. An instance's silhouette coefficient is equal to  $(b - a) / \max(a, b)$ , where  $a$  is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and  $b$  is the mean nearest-cluster distance (i.e., the mean distance to the instances of the next closest cluster, defined as the one that minimizes  $b$ , excluding the instance's own cluster). The silhouette coefficient can vary between  $-1$  and  $+1$ . A coefficient close to  $+1$  means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to  $0$  means that it is close to a cluster

boundary; finally, a coefficient close to  $-1$  means that the instance may have been assigned to the wrong cluster.

To compute the silhouette score, you can use Scikit-Learn's `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned:

```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
np.float64(0.655517642572828)
```

Let's compare the silhouette scores for different numbers of clusters (see [Figure 8-8](#)).



*Figure 8-8. Selecting the number of clusters  $k$  using the silhouette score*

As you can see, this visualization is much richer than the previous one: although it confirms that  $k = 4$  is a very good choice, it also highlights the fact that  $k = 5$  is quite good as well, and much better than  $k = 6$  or  $7$ . This was not visible when comparing inertias.

An even more informative visualization is obtained when we plot every instance's silhouette coefficient, sorted by the clusters they are assigned to and by the value of the coefficient. This is called a *silhouette diagram* (see [Figure 8-9](#)). Each diagram contains one knife shape per cluster. The shape's height indicates the number of instances in the cluster, and its width represents the sorted silhouette coefficients of the instances in the cluster (wider is better).

The vertical dashed lines represent the mean silhouette score for each number of clusters. When most of the instances in a cluster have a lower coefficient than this score (i.e., if many of the instances stop short of the dashed line, ending to the left of it), then the cluster is rather bad since this means its instances are much too close to other clusters. Here we can see that when  $k = 3$  or  $6$ , we get bad clusters. But when  $k = 4$  or  $5$ , the clusters look pretty good: most instances extend beyond the dashed line, to the right and closer to  $1.0$ . When  $k = 4$ , the cluster at index  $0$  (at the bottom) is rather big. When  $k = 5$ , all clusters have similar sizes. So, even though the overall silhouette score from  $k = 4$  is slightly greater than for  $k = 5$ , it seems like a good idea to use  $k = 5$  to get clusters of similar sizes.

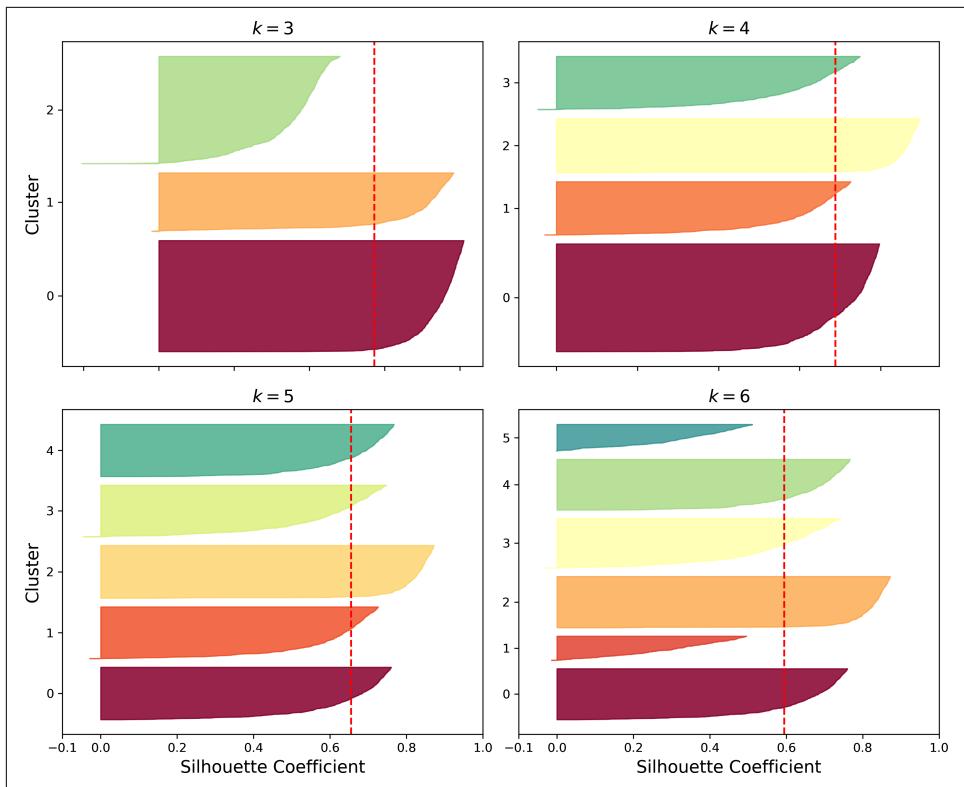


Figure 8-9. Analyzing the silhouette diagrams for various values of  $k$

## Limits of k-Means

Despite its many merits, most notably being fast and scalable,  $k$ -means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid suboptimal solutions, plus you need to specify the number of clusters, which can be quite a hassle. Moreover,  $k$ -means does not behave very well when the clusters have varying sizes, different densities, or nonspherical shapes. For example, Figure 8-10 shows how  $k$ -means clusters a dataset containing three ellipsoidal clusters of different dimensions, densities, and orientations.

As you can see, neither of these solutions is any good. The solution on the left is better, but it still chops off 25% of the middle cluster and assigns it to the cluster on the right. The solution on the right is just terrible, even though its inertia is lower. So, depending on the data, different clustering algorithms may perform better. On these types of elliptical clusters, Gaussian mixture models work great.

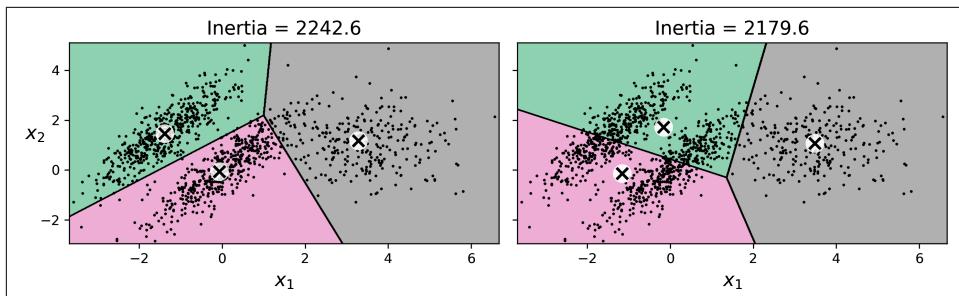


Figure 8-10.  $k$ -means fails to cluster these ellipsoidal blobs properly



It is important to scale the input features (see [Chapter 2](#)) before you run  $k$ -means, or the clusters may be very stretched and  $k$ -means will perform poorly. Scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally helps  $k$ -means.

Now let's look at a few ways we can benefit from clustering. We will use  $k$ -means, but feel free to experiment with other clustering algorithms.

## Using Clustering for Image Segmentation

*Image segmentation* is the task of partitioning an image into multiple segments. There are several variants:

- In *color segmentation*, pixels with a similar color get assigned to the same segment. This is sufficient in many applications. For example, if you want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.
- In *semantic segmentation*, all pixels that are part of the same object type get assigned to the same segment. For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the "pedestrian" segment (there would be one segment containing all the pedestrians).
- In *instance segmentation*, all pixels that are part of the same individual object are assigned to the same segment. In this case there would be a different segment for each pedestrian.

The state of the art in semantic or instance segmentation today is achieved using complex architectures based on convolutional neural networks (see [Chapter 12](#)) or vision transformers (see [Chapter 16](#)). In this chapter we are going to focus on the (much simpler) color segmentation task, using  $k$ -means.

We'll start by importing the Pillow package (successor to the Python Imaging Library, PIL), which we'll then use to load the *ladybug.png* image (see the upper-left image in [Figure 8-11](#)), assuming it's located at `filepath`:

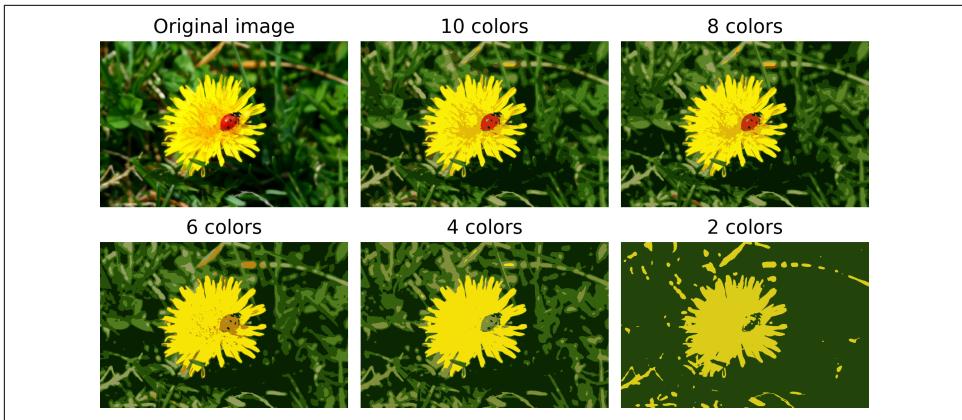
```
>>> import PIL  
>>> image = np.asarray(PIL.Image.open(filepath))  
>>> image.shape  
(533, 800, 3)
```

The image is represented as a 3D array. The first dimension's size is the height; the second is the width; and the third is the number of color channels, in this case red, green, and blue (RGB). In other words, for each pixel there is a 3D vector containing the intensities of red, green, and blue as unsigned 8-bit integers between 0 and 255. Some images may have fewer channels (such as grayscale images, which only have one), and some images may have more channels (such as images with an additional *alpha channel* for transparency, or satellite images, which often contain channels for additional light frequencies, like infrared).

The following code reshapes the array to get a long list of RGB colors, then it clusters these colors using *k*-means with eight clusters. It creates a `segmented_img` array containing the nearest cluster center for each pixel (i.e., the mean color of each pixel's cluster), and lastly it reshapes this array to the original image shape. The third line uses advanced NumPy indexing; for example, if the first 10 labels in `kmeans_.labels_` are equal to 1, then the first 10 colors in `segmented_img` are equal to `kmeans_.cluster_centers_[1]`:

```
X = image.reshape(-1, 3)  
kmeans = KMeans(n_clusters=8, random_state=42).fit(X)  
segmented_img = kmeans.cluster_centers_[kmeans.labels_]  
segmented_img = segmented_img.reshape(image.shape)
```

This outputs the image shown in the upper right of [Figure 8-11](#). You can experiment with various numbers of clusters, as shown in the figure. When you use fewer than eight clusters, notice that the ladybug's flashy red color fails to get a cluster of its own: it gets merged with colors from the environment. This is because *k*-means prefers clusters of similar sizes. The ladybug is small—much smaller than the rest of the image—so even though its color is flashy, *k*-means fails to dedicate a cluster to it.



*Figure 8-11. Image segmentation using k-means with various numbers of color clusters*

That wasn't too hard, was it? Now let's look at another application of clustering.

## Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances. For example, clustering can help choose which additional instances to label (e.g., near the cluster centroids). It can also be used to propagate the most common label in each cluster to the unlabeled instances in that cluster. Let's try these ideas on the digits dataset, which is a simple MNIST-like dataset containing 1,797 grayscale  $8 \times 8$  images representing the digits 0 to 9. First, let's load and split the dataset (it's already shuffled):

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
X_train, y_train = X_digits[:1400], y_digits[:1400]
X_test, y_test = X_digits[1400:], y_digits[1400:]
```

We will pretend we only have labels for 50 instances. To get a baseline performance, let's train a logistic regression model on these 50 labeled instances:

```
from sklearn.linear_model import LogisticRegression

n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

We can then measure the accuracy of this model on the test set (note that the test set must be labeled):

```
>>> log_reg.score(X_test, y_test)  
0.7581863979848866
```

The model's accuracy is just 75.8%. That's not great: indeed, if you try training the model on the full training set, you will find that it will reach about 90.9% accuracy. Let's see how we can do better. First, let's cluster the training set into 50 clusters. Then, for each cluster, we'll find the image closest to the centroid. We'll call these images the *representative images*:

```
k = 50  
kmeans = KMeans(n_clusters=k, random_state=42)  
X_digits_dist = kmeans.fit_transform(X_train)  
representative_digit_idx = X_digits_dist.argmin(axis=0)  
X_representative_digits = X_train[representative_digit_idx]
```

Figure 8-12 shows the 50 representative images.

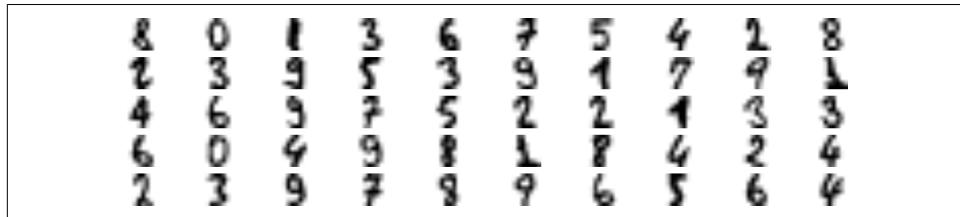


Figure 8-12. Fifty representative digit images (one per cluster)

Let's look at each image and manually label them:

```
y_representative_digits = np.array([8, 0, 1, 3, 6, 7, 5, 4, 2, 8, ..., 6, 4])
```

Now we have a dataset with just 50 labeled instances, but instead of being random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

```
>>> log_reg = LogisticRegression(max_iter=10_000)  
>>> log_reg.fit(X_representative_digits, y_representative_digits)  
>>> log_reg.score(X_test, y_test)  
0.8337531486146096
```

Wow! We jumped from 75.8% accuracy to 83.4%, although we are still only training the model on 50 instances. Since it is often costly and painful to label instances, especially when it has to be done manually by experts, it is a good idea to label representative instances rather than just random instances.

But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called *label propagation*:

```

y_train_propagated = np.empty(len(X_train), dtype=np.int64)
for i in range(k):
    y_train_propagated[kmeans.labels_ == i] = y_representative_digits[i]

```

Now let's train the model again and look at its performance:

```

>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
>>> log_reg.score(X_test, y_test)
0.8690176322418136

```

We got another significant accuracy boost! Let's see if we can do even better by ignoring the 50% of instances that are farthest from their cluster center: this should eliminate some outliers. The following code first computes the distance from each instance to its closest cluster center, then for each cluster it sets the 50% largest distances to  $-1$ . Lastly, it creates a set without these instances marked with a  $-1$  distance:

```

percentile_closest = 50

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]

```

Now let's train the model again on this partially propagated dataset and see what accuracy we get:

```

>>> log_reg = LogisticRegression(max_iter=10_000)
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.8841309823677582

```

Nice! With just 50 labeled instances (only 5 examples per class on average!) we got 88.4% accuracy, pretty close to the performance we got on the fully labeled digits dataset. This is partly thanks to the fact that we dropped some outliers, and partly because the propagated labels are actually pretty good—their accuracy is about 98.9%, as the following code shows:

```

>>> (y_train_partially_propagated == y_train[partially_propagated]).mean()
np.float64(0.9887798036465638)

```



Scikit-Learn also offers two classes that can propagate labels automatically: `LabelSpreading` and `LabelPropagation` in the `sklearn.semi_supervised` package. Both classes construct a similarity matrix between all the instances, and iteratively propagate labels from labeled instances to similar unlabeled instances. There's also a different class called `SelfTrainingClassifier` in the same package: you give it a base classifier (e.g., `RandomForestClassifier`) and it trains it on the labeled instances, then uses it to predict labels for the unlabeled samples. It then updates the training set with the labels it is most confident about, and repeats this process of training and labeling until it cannot add labels anymore. These techniques are not magic bullets, but they can occasionally give your model a little boost.

## Active Learning

To continue improving your model and your training set, the next step could be to do a few rounds of *active learning*, which is when a human expert interacts with the learning algorithm, providing labels for specific instances when the algorithm requests them. There are many different strategies for active learning, but one of the most common ones is called *uncertainty sampling*. Here is how it works:

1. The model is trained on the labeled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
2. The instances for which the model is most uncertain (i.e., where its estimated probability is lowest) are given to the expert for labeling.
3. You iterate this process until the performance improvement stops being worth the labeling effort.

Other active learning strategies include labeling the instances that would result in the largest model change or the largest drop in the model's validation error, or the instances that different models disagree on (e.g., an SVM and a random forest).

Before we move on to Gaussian mixture models, let's take a look at DBSCAN, another popular clustering algorithm that illustrates a very different approach based on local density estimation. This approach allows the algorithm to identify clusters of arbitrary shapes.

## DBSCAN

The *density-based spatial clustering of applications with noise* (DBSCAN) algorithm defines clusters as continuous regions of high density. Here is how it works:

- For each instance, the algorithm counts how many instances are located within a small distance  $\epsilon$  (epsilon) from it. This region is called the instance's  $\epsilon$ -neighborhood.
- If an instance has at least `min_samples` instances in its  $\epsilon$ -neighborhood (including itself), then it is considered a *core instance*. In other words, core instances are those that are located in dense regions.
- All instances in the neighborhood of a core instance belong to the same cluster. This neighborhood may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.
- Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

This algorithm works well if all the clusters are well separated by low-density regions. The DBSCAN class in Scikit-Learn is as simple to use as you might expect. Let's test it on the moons dataset, introduced in [Chapter 5](#):

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
dbSCAN = DBSCAN(eps=0.05, min_samples=5)
dbSCAN.fit(X)
```

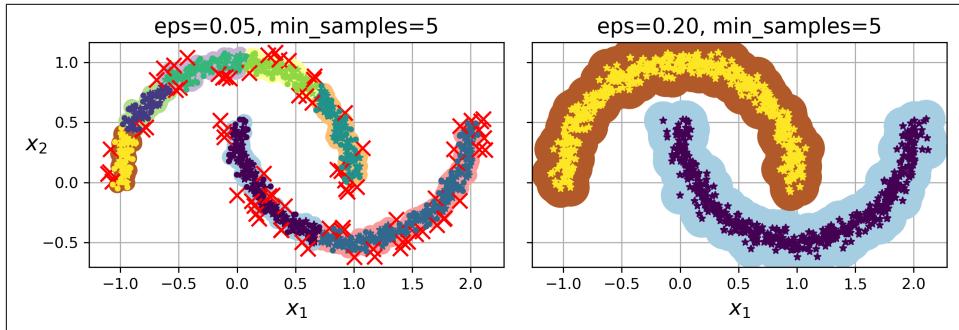
The labels of all the instances are now available in the `labels_` instance variable:

```
>>> dbSCAN.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5, [...], 3,  3,  4,  2,  6,  3])
```

Notice that some instances have a cluster index equal to  $-1$ , which means that they are considered as anomalies by the algorithm. The indices of the core instances are available in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```
>>> dbSCAN.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, [...], 993, 995, 997, 998, 999])
>>> dbSCAN.components_
array([[ -0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       [...],
       [ 0.79419406,  0.60777171]])
```

This clustering is represented in the lefthand plot of [Figure 8-13](#). As you can see, it identified quite a lot of anomalies, plus seven different clusters. How disappointing! Fortunately, if we widen each instance's neighborhood by increasing `eps` to 0.2, we get the clustering on the right, which looks perfect. Let's continue with this model.



*Figure 8-13. DBSCAN clustering using two different neighborhood radiiuses*

Surprisingly, the `DBSCAN` class does not have a `predict()` method, although it has a `fit_predict()` method. In other words, it cannot predict which cluster a new instance belongs to. This decision was made because different classification algorithms can be better for different tasks, so the authors decided to let the user choose which one to use. Moreover, it's not hard to implement. For example, let's train a `KNeighborsClassifier`:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbSCAN.components_, dbSCAN.labels_[dbSCAN.core_sample_indices_])
```

Now, given a few new instances, we can predict which clusters they most likely belong to and even estimate a probability for each cluster:

```
>>> X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
>>> knn.predict(X_new)
array([1, 0, 1, 0])
>>> knn.predict_proba(X_new)
array([[0.18, 0.82],
       [1. , 0. ],
       [0.12, 0.88],
       [1. , 0. ]])
```

Note that we only trained the classifier on the core instances, but we could also have chosen to train it on all the instances, or all but the anomalies: this choice depends on the final task.

The decision boundary is represented in [Figure 8-14](#) (the crosses represent the four instances in `X_new`). Notice that since there is no anomaly in the training set, the

classifier always chooses a cluster, even when that cluster is far away. It is fairly straightforward to introduce a maximum distance, in which case the two instances that are far away from both clusters are classified as anomalies. To do this, use the `kneighbors()` method of the `KNeighborsClassifier`. Given a set of instances, it returns the distances and the indices of the  $k$ -nearest neighbors in the training set (two matrices, each with  $k$  columns):

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1
>>> y_pred.ravel()
array([-1,  0,  1, -1])
```

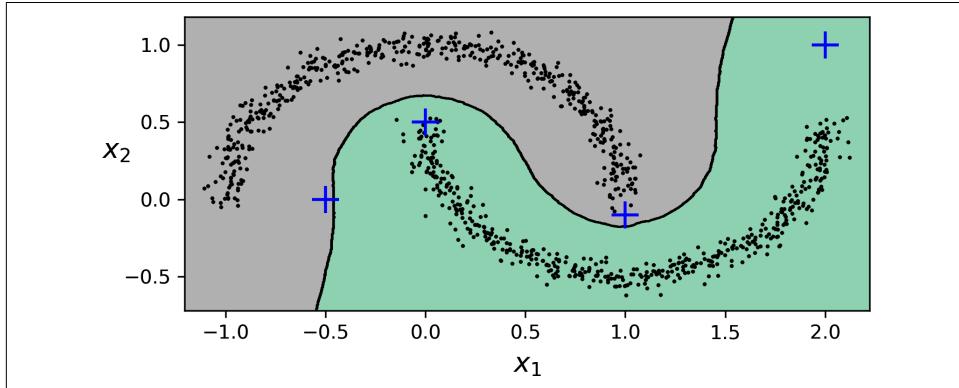


Figure 8-14. Decision boundary between two clusters

In short, DBSCAN is a very simple yet powerful algorithm capable of identifying any number of clusters of any shape. It is robust to outliers, and it has just two hyperparameters (`eps` and `min_samples`). If the density varies significantly across the clusters, however, or if there's no sufficiently low-density region around some clusters, DBSCAN can struggle to capture all the clusters properly. Moreover, its computational complexity is roughly  $O(m^2n)$ , so it does not scale well to large datasets.



You may also want to try *hierarchical DBSCAN* (HDBSCAN), using `sklearn.cluster.HDBSCAN`: it is often better than DBSCAN at finding clusters of varying densities.

## Other Clustering Algorithms

Scikit-Learn implements several more clustering algorithms that you should take a look at. I cannot cover them all in detail here, but here is a brief overview:

### *Agglomerative clustering*

A hierarchy of clusters is built from the bottom up. Think of many tiny bubbles floating on water and gradually attaching to each other until there's one big group of bubbles. Similarly, at each iteration, agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If you drew a tree with a branch for every pair of clusters that merged, you would get a binary tree of clusters, where the leaves are the individual instances. This approach can capture clusters of various shapes; it also produces a flexible and informative cluster tree instead of forcing you to choose a particular cluster scale, and it can be used with any pairwise distance. It can scale nicely to large numbers of instances if you provide a connectivity matrix, which is a sparse  $m \times m$  matrix that indicates which pairs of instances are neighbors (e.g., returned by `sklearn.neighbors.kneighbors_graph()`). Without a connectivity matrix, the algorithm does not scale well to large datasets.

### *BIRCH*

The balanced iterative reducing and clustering using hierarchies (BIRCH) algorithm was designed specifically for very large datasets, and it can be faster than batch  $k$ -means, with similar results, as long as the number of features is not too large (<20). During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster, without having to store all the instances in the tree: this approach allows it to use limited memory while handling huge datasets.

### *Mean-shift*

This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains). Mean-shift shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster. Mean-shift has some of the same features as DBSCAN, like how it can find any number of clusters of any shape, it has very few hyperparameters (just one—the radius of the circles, called the *bandwidth*), and it relies on local density estimation. But unlike DBSCAN, mean-shift tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is  $O(m^2n)$ , so it is not suited for large datasets.

### *Affinity propagation*

In this algorithm, instances repeatedly exchange messages between one another until every instance has elected another instance (or itself) to represent it. These elected instances are called *exemplars*. Each exemplar and all the instances that

elected it from one cluster. In real-life politics, you typically want to vote for a candidate whose opinions are similar to yours, but you also want them to win the election, so you might choose a candidate you don't fully agree with, but who is more popular. You typically evaluate popularity through polls. Affinity propagation works in a similar way, and it tends to choose exemplars located near the center of clusters, similar to  $k$ -means. But unlike with  $k$ -means, you don't have to pick a number of clusters ahead of time: it is determined during training. Moreover, affinity propagation can deal nicely with clusters of different sizes. Sadly, this algorithm has a computational complexity of  $O(m^2)$ , so it is not suited for large datasets.

### *Spectral clustering*

This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces the matrix's dimensionality), then it uses another clustering algorithm in this low-dimensional space (Scikit-Learn's implementation uses  $k$ -means). Spectral clustering can capture complex cluster structures, and it can also be used to cut graphs (e.g., to identify clusters of friends on a social network). It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

Now let's dive into Gaussian mixture models, which can be used for density estimation, clustering, and anomaly detection.

## Gaussian Mixtures

A *Gaussian mixture model* (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid. Each cluster can have a different ellipsoidal shape, size, density, and orientation, just like in [Figure 8-10](#).<sup>7</sup> When you observe an instance, you know it was generated from one of the Gaussian distributions, but you are not told which one, and you do not know what the parameters of these distributions are.

There are several GMM variants. In the simplest variant, implemented in the `GaussianMixture` class, you must know in advance the number  $k$  of Gaussian distributions. The dataset  $\mathbf{X}$  is assumed to have been generated through the following probabilistic process:

---

<sup>7</sup> In contrast, as we saw earlier,  $k$ -means implicitly assumes that clusters all have a similar size and density, and are all roughly round.

- For each instance, a cluster is picked randomly from among  $k$  clusters. The probability of choosing the  $j^{\text{th}}$  cluster is the cluster's weight  $\phi^{(j)}$ .<sup>8</sup> The index of the cluster chosen for the  $i^{\text{th}}$  instance is denoted  $z^{(i)}$ .
- If the  $i^{\text{th}}$  instance was assigned to the  $j^{\text{th}}$  cluster (i.e.,  $z^{(i)} = j$ ), then the location  $\mathbf{x}^{(i)}$  of this instance is sampled randomly from the Gaussian distribution with mean  $\mu^{(j)}$  and covariance matrix  $\Sigma^{(j)}$ . This is denoted  $\mathbf{x}^{(i)} \sim \mathcal{N}(\mu^{(j)}, \Sigma^{(j)})$ .

So what can you do with such a model? Well, given the dataset  $\mathbf{X}$ , you typically want to start by estimating the weights  $\phi$  and all the distribution parameters  $\mu^{(1)}$  to  $\mu^{(k)}$  and  $\Sigma^{(1)}$  to  $\Sigma^{(k)}$ . Scikit-Learn's `GaussianMixture` class makes this super easy:

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

Let's look at the parameters that the algorithm estimated:

```
>>> gm.weights_
array([0.40005972, 0.20961444, 0.39032584])
>>> gm.means_
array([[ -1.40764129,   1.42712848],
       [  3.39947665,   1.05931088],
       [  0.05145113,   0.07534576]])
>>> gm.covariances_
array([[[ 0.63478217,   0.72970097],
        [ 0.72970097,   1.16094925]],

       [[ 1.14740131, -0.03271106],
        [-0.03271106,  0.95498333]],

       [[ 0.68825143,   0.79617956],
        [ 0.79617956,   1.21242183]]])
```

Great, it worked fine! Indeed, two of the three clusters were generated with 500 instances each, while the third cluster only contains 250 instances. So the true cluster weights are 0.4, 0.4, and 0.2, respectively, and that's roughly what the algorithm found (in a different order). Similarly, the true means and covariance matrices are quite close to those found by the algorithm. But how? This class relies on the *expectation-maximization* (EM) algorithm, which has many similarities with the  $k$ -means algorithm: it also initializes the cluster parameters randomly, then it repeats two steps until convergence, first assigning instances to clusters (this is called the *expectation step*) and then updating the clusters (this is called the *maximization step*). Sounds familiar, right? In the context of clustering, you can think of EM as a generalization of  $k$ -means that not only finds the cluster centers ( $\mu^{(1)}$  to  $\mu^{(k)}$ ), but also their size,

---

<sup>8</sup> Phi ( $\phi$  or  $\varphi$ ) is the 21st letter of the Greek alphabet.

shape, and orientation ( $\Sigma^{(1)}$  to  $\Sigma^{(k)}$ ), as well as their relative weights ( $\phi^{(1)}$  to  $\phi^{(k)}$ ). Unlike  $k$ -means, though, EM uses soft cluster assignments, not hard assignments. For each instance, during the expectation step, the algorithm estimates the probability that it belongs to each cluster (based on the current cluster parameters). Then, during the maximization step, each cluster is updated using *all* the instances in the dataset, with each instance weighted by the estimated probability that it belongs to that cluster. These probabilities are called the *responsibilities* of the clusters for the instances. During the maximization step, each cluster's update will mostly be impacted by the instances it is most responsible for.



Unfortunately, just like  $k$ -means, EM can end up converging to poor solutions, so it needs to be run several times, keeping only the best solution. This is why we set `n_init` to 10. Be careful: by default `n_init` is set to 1.

You can check whether the algorithm converged and how many iterations it took:

```
>>> gm.converged_
True
>>> gm.n_iter_
4
```

Now that you have an estimate of the location, size, orientation, and relative weight of each cluster, the model can easily assign each instance to the most likely cluster (hard clustering) or estimate the probability that it belongs to a particular cluster (soft clustering). Just use the `predict()` method for hard clustering, or the `predict_proba()` method for soft clustering:

```
>>> gm.predict(X)
array([2, 2, 0, ..., 1, 1, 1])
>>> gm.predict_proba(X).round(3)
array([[0.    , 0.023, 0.977],
       [0.001, 0.016, 0.983],
       [1.    , 0.    , 0.    ],
       ...,
       [0.    , 1.    , 0.    ],
       [0.    , 1.    , 0.    ],
       [0.    , 1.    , 0.    ]])
```

A Gaussian mixture model is a *generative model*, meaning you can sample new instances from it (note that they are ordered by cluster index):

```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[-2.32491052,  1.04752548],
       [-1.16654983,  1.62795173],
       [ 1.84860618,  2.07374016],
       [ 3.98304484,  1.49869936],
```

```

[ 3.8163406 ,  0.53038367],
[ 0.38079484, -0.56239369])
>>> y_new
array([0, 0, 1, 1, 1, 2])

```

It is also possible to estimate the density of the model at any given location. This is achieved using the `score_samples()` method: for each instance it is given, this method estimates the log of the *probability density function* (PDF) at that location. The greater the score, the higher the density:

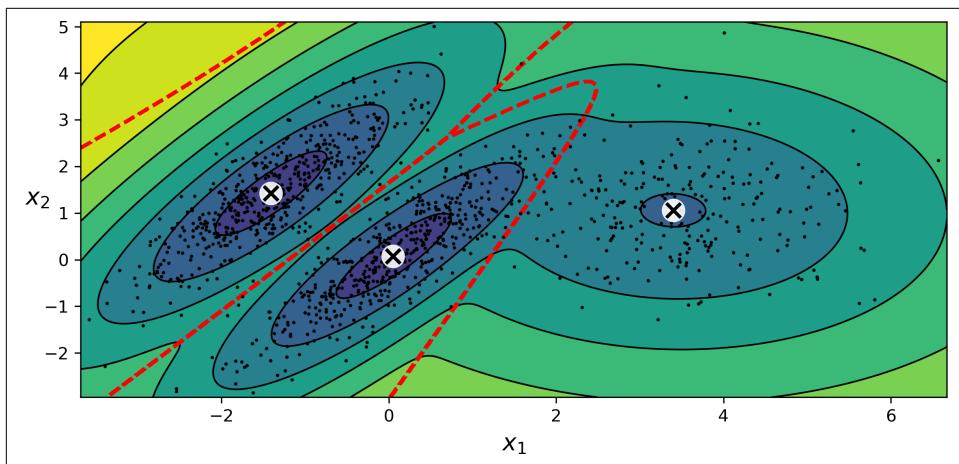
```

>>> gm.score_samples(X).round(2)
array([-2.61, -3.57, -3.33, ... , -3.51, -4.4 , -3.81])

```

If you compute the exponential of these scores, you get the value of the PDF at the location of the given instances. These are not probabilities, but probability *densities*: they can take on any positive value, not just a value between 0 and 1. To estimate the probability that an instance will fall within a particular region, you would have to integrate the PDF over that region (if you do so over the entire space of possible instance locations, the result will be 1).

[Figure 8-15](#) shows the cluster means, the decision boundaries (dashed lines), and the density contours of this model.



*Figure 8-15. Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model*

Nice! The algorithm clearly found an excellent solution. Of course, we made its task easy by generating the data using a set of 2D Gaussian distributions (unfortunately, real-life data is not always so Gaussian and low-dimensional). We also gave the algorithm the correct number of clusters. When there are many dimensions, or many clusters, or few instances, EM can struggle to converge to the optimal solution. You might need to reduce the difficulty of the task by limiting the number of parameters

that the algorithm has to learn. One way to do this is to limit the range of shapes and orientations that the clusters can have. This can be achieved by imposing constraints on the covariance matrices. To do this, set the `covariance_type` hyperparameter to one of the following values:

#### "spherical"

All clusters must be spherical, but they can have different diameters (i.e., different variances).

#### "diag"

Clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the coordinate axes (i.e., the covariance matrices must be diagonal).

#### "tied"

All clusters must have the same ellipsoidal shape, size, and orientation (i.e., all clusters share the same covariance matrix).

By default, `covariance_type` is equal to "full", which means that each cluster can take on any shape, size, and orientation (it has its own unconstrained covariance matrix). [Figure 8-16](#) plots the solutions found by the EM algorithm when `covariance_type` is set to "tied" or "spherical".

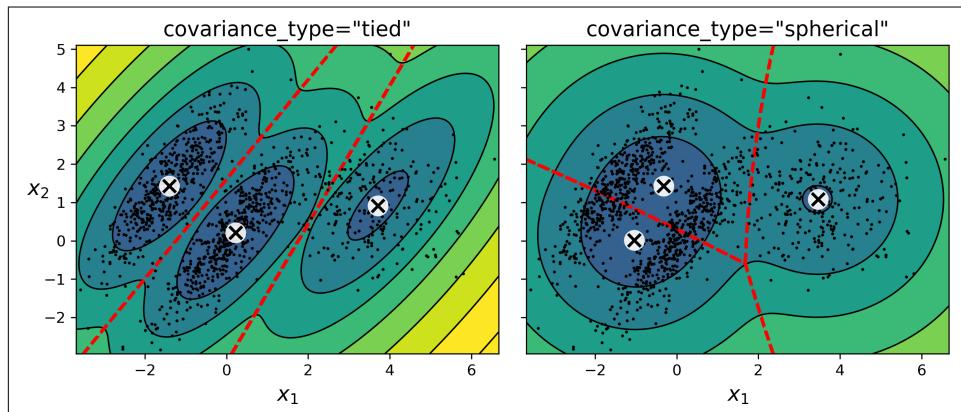


Figure 8-16. Gaussian mixtures for tied clusters (left) and spherical clusters (right)



The computational complexity of training a `GaussianMixture` model depends on the number of instances  $m$ , the number of dimensions  $n$ , the number of clusters  $k$ , and the constraints on the covariance matrices. If `covariance_type` is "spherical" or "diag", it is  $O(kmn)$ , assuming the data has a clustering structure. If `covariance_type` is "tied" or "full", it is  $O(kmn^2 + kn^3)$ , so it will not scale to large numbers of features.

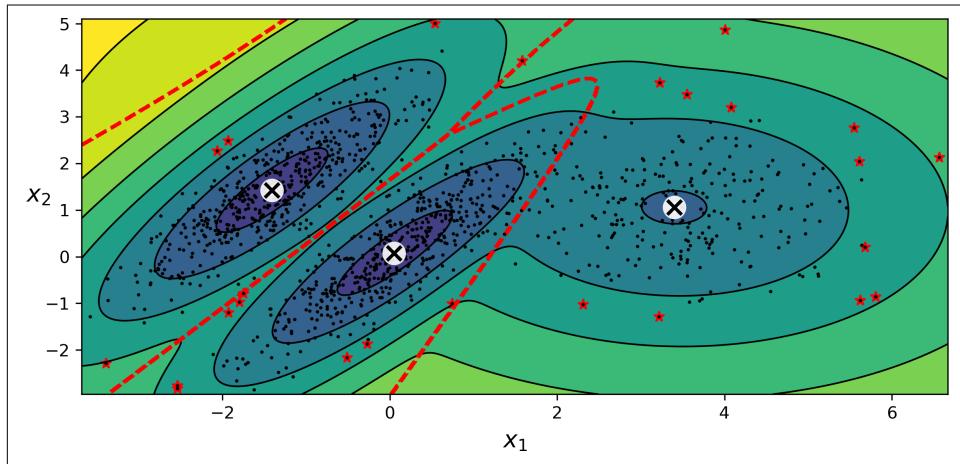
Gaussian mixture models can also be used for anomaly detection. We'll see how in the next section.

## Using Gaussian Mixtures for Anomaly Detection

Using a Gaussian mixture model for anomaly detection is quite simple: any instance located in a low-density region can be considered an anomaly. You must define what density threshold you want to use. For example, in a manufacturing company that tries to detect defective products, the ratio of defective products is usually well known. Say it is equal to 2%. You then set the density threshold to be the value that results in having 2% of the instances located in areas below that threshold density. If you notice that you get too many false positives (i.e., perfectly good products that are flagged as defective), you can lower the threshold. Conversely, if you have too many false negatives (i.e., defective products that the system does not flag as defective), you can increase the threshold. This is the usual precision/recall trade-off (see [Chapter 3](#)). Here is how you would identify the outliers using the second percentile lowest density as the threshold (i.e., approximately 2% of the instances will be flagged as anomalies):

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 2)
anomalies = X[densities < density_threshold]
```

[Figure 8-17](#) represents these anomalies as stars.



*Figure 8-17. Anomaly detection using a Gaussian mixture model*

A closely related task is *novelty detection*: it differs from anomaly detection in that the algorithm is assumed to be trained on a “clean” dataset, uncontaminated by outliers, whereas anomaly detection does not make this assumption. Indeed, outlier detection is often used to clean up a dataset.



Gaussian mixture models try to fit all the data, including the outliers; if you have too many of them this will bias the model's view of "normality", and some outliers may wrongly be considered as normal. If this happens, you can try to fit the model once, use it to detect and remove the most extreme outliers, then fit the model again on the cleaned-up dataset. Another approach is to use robust covariance estimation methods (see the `EllipticEnvelope` class).

Just like  $k$ -means, the `GaussianMixture` algorithm requires you to specify the number of clusters. So how can you find that number?

## Selecting the Number of Clusters

With  $k$ -means, you can use the inertia or the silhouette score to select the appropriate number of clusters. But with Gaussian mixtures, it is not possible to use these metrics because they are not reliable when the clusters are not spherical or have different sizes. Instead, you can try to find the model that minimizes a *theoretical information criterion*, such as the *Bayesian information criterion* (BIC) or the *Akaike information criterion* (AIC), defined in [Equation 8-2](#).

*Equation 8-2. Bayesian information criterion (BIC) and Akaike information criterion (AIC)*

$$BIC = \log(m)p - 2\log(\hat{\mathcal{L}})$$

$$AIC = 2p - 2\log(\hat{\mathcal{L}})$$

In these equations:

- $m$  is the number of instances, as always.
- $p$  is the number of parameters learned by the model.
- $\hat{\mathcal{L}}$  is the maximized value of the *likelihood function* of the model.

Both the BIC and the AIC penalize models that have more parameters to learn (e.g., more clusters) and reward models that fit the data well. They often end up selecting the same model. When they differ, the model selected by the BIC tends to be simpler (fewer parameters) than the one selected by the AIC, but tends to not fit the data quite as well (this is especially true for larger datasets).

## Likelihood Function

The terms “probability” and “likelihood” are often used interchangeably in everyday language, but they have very different meanings in statistics. Given a statistical model with some parameters  $\theta$ , the word “probability” is used to describe how plausible a future outcome  $x$  is (knowing the parameter values  $\theta$ ), while the word “likelihood” is used to describe how plausible a particular set of parameter values  $\theta$  are, after the outcome  $x$  is known.

Consider a 1D mixture model of two Gaussian distributions centered at  $-4$  and  $+1$ . For simplicity, this toy model has a single parameter  $\theta$  that controls the standard deviations of both distributions. The top-left contour plot in Figure 8-18 shows the entire model  $f(x; \theta)$  as a function of both  $x$  and  $\theta$ . To estimate the probability distribution of a future outcome  $x$ , you need to set the model parameter  $\theta$ . For example, if you set  $\theta$  to 1.3 (the horizontal line), you get the probability density function  $f(x; \theta = 1.3)$  shown in the lower-left plot. Say you want to estimate the probability that  $x$  will fall between  $-2$  and  $+2$ . You must calculate the integral of the PDF on this range (i.e., the surface of the shaded region). But what if you don’t know  $\theta$ , and if instead you have observed a single instance  $x = 2.5$  (the vertical line in the upper-left plot)? In this case, you get the likelihood function  $\mathcal{L}(\theta|x = 2.5) = f(x = 2.5; \theta)$ , represented in the upper-right plot.

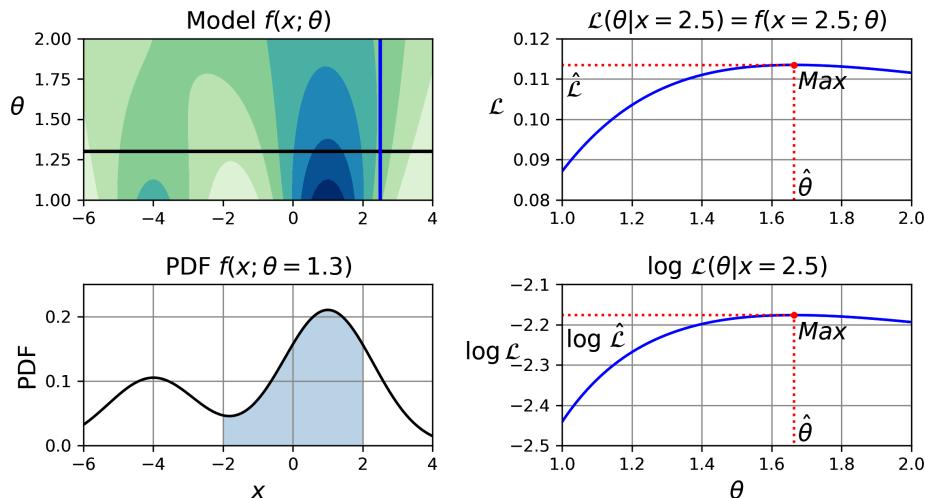


Figure 8-18. A model’s parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right), and a log likelihood function (lower right)

In short, the PDF is a function of  $x$  (with  $\theta$  fixed), while the likelihood function is a function of  $\theta$  (with  $x$  fixed). It is important to understand that the likelihood function is *not* a probability distribution: if you integrate a probability distribution over all

possible values of  $x$ , you always get 1, but if you integrate the likelihood function over all possible values of  $\theta$ , the result can be any positive value.

Given a dataset  $X$ , a common task is to try to estimate the most likely values for the model parameters. To do this, you must find the values that maximize the likelihood function, given  $X$ . In this example, if you have observed a single instance  $x = 2.5$ , the *maximum likelihood estimate* (MLE) of  $\theta$  is  $\hat{\theta} \approx 1.66$ . If a prior probability distribution  $g$  over  $\theta$  exists, it is possible to take it into account by maximizing  $\mathcal{L}(\theta|x)g(\theta)$  rather than just maximizing  $\mathcal{L}(\theta|x)$ . This is called *maximum a-posteriori* (MAP) estimation. Since MAP constrains the parameter values, you can think of it as a regularized version of MLE.

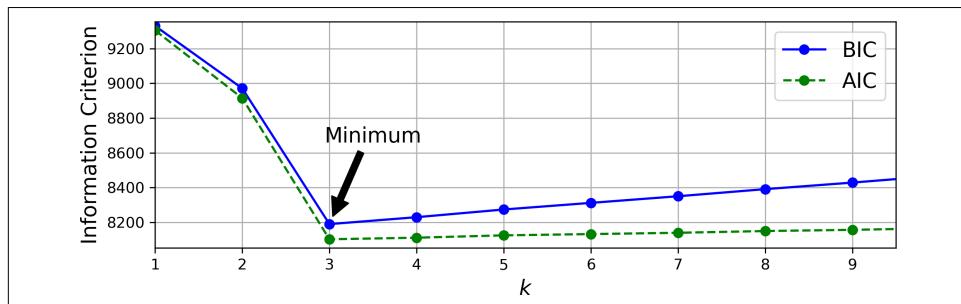
Notice that maximizing the likelihood function is equivalent to maximizing its logarithm (represented in the lower-right plot in [Figure 8-18](#)). Indeed, the logarithm is a strictly increasing function, so if  $\theta$  maximizes the log likelihood, it also maximizes the likelihood. It turns out that it is generally easier to maximize the log likelihood. For example, if you observed several independent instances  $x^{(1)}$  to  $x^{(m)}$ , you would need to find the value of  $\theta$  that maximizes the product of the individual likelihood functions. But it is equivalent, and much simpler, to maximize the sum (not the product) of the log likelihood functions, thanks to the magic of the logarithm which converts products into sums:  $\log(ab) = \log(a) + \log(b)$ .

Once you have estimated  $\hat{\theta}$ , the value of  $\theta$  that maximizes the likelihood function, then you are ready to compute  $\hat{\mathcal{L}} = \mathcal{L}(\hat{\theta}, X)$ , which is the value used to compute the AIC and BIC; you can think of it as a measure of how well the model fits the data.

To compute the BIC and AIC, call the `bic()` and `aic()` methods:

```
>>> gm.bic(X)
np.float64(8189.733705221636)
>>> gm.aic(X)
np.float64(8102.508425106598)
```

[Figure 8-19](#) shows the BIC for different numbers of clusters  $k$ . As you can see, both the BIC and the AIC are lowest when  $k = 3$ , so it is most likely the best choice.



*Figure 8-19. AIC and BIC for different numbers of clusters  $k$*

## Bayesian Gaussian Mixture Models

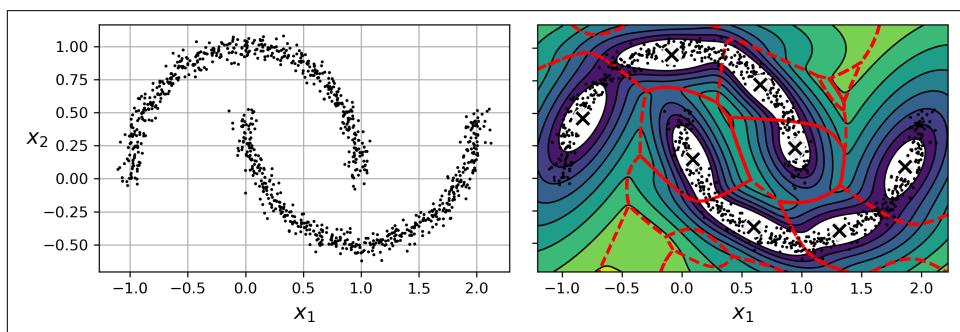
Rather than manually searching for the optimal number of clusters, you can use the `BayesianGaussianMixture` class, which is capable of giving weights equal (or close) to zero to unnecessary clusters. Set the number of clusters `n_components` to a value that you have good reason to believe is greater than the optimal number of clusters (this assumes some minimal knowledge about the problem at hand), and the algorithm will eliminate the unnecessary clusters automatically. For example, let's set the number of clusters to 10 and see what happens:

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10, max_iter=500,
...                                 random_state=42)
...
>>> bgm.fit(X)
>>> bgm.weights_.round(2)
array([0.4 , 0.21, 0.39, 0. , 0. , 0. , 0. , 0. , 0. , 0. ])
```

Perfect: the algorithm automatically detected that only three clusters are needed, and the resulting clusters are almost identical to the ones in [Figure 8-15](#).

A final note about Gaussian mixture models: although they work great on clusters with ellipsoidal shapes, they don't do so well with clusters of very different shapes. For example, let's see what happens if we use a Bayesian Gaussian mixture model to cluster the moons dataset (see [Figure 8-20](#)).

Oops! The algorithm desperately searched for ellipsoids, so it found eight different clusters instead of two. The density estimation is not too bad, so this model could perhaps be used for anomaly detection, but it failed to identify the two moons. To conclude this chapter, let's take a quick look at a few algorithms capable of dealing with arbitrarily shaped clusters.



*Figure 8-20. Fitting a Gaussian mixture to nonellipsoidal clusters*

## Other Algorithms for Anomaly and Novelty Detection

Scikit-Learn implements other algorithms dedicated to anomaly detection or novelty detection:

### *Fast-MCD (minimum covariance determinant)*

Implemented by the `EllipticEnvelope` class, this algorithm is useful for outlier detection, in particular to clean up a dataset. It assumes that the normal instances (inliers) are generated from a single Gaussian distribution (not a mixture). It also assumes that the dataset is contaminated with outliers that were not generated from this Gaussian distribution. When the algorithm estimates the parameters of the Gaussian distribution (i.e., the shape of the elliptic envelope around the inliers), it is careful to ignore the instances that are most likely outliers. This technique gives a better estimation of the elliptic envelope and thus makes the algorithm better at identifying the outliers.

### *Isolation forest*

This is an efficient algorithm for outlier detection, especially in high-dimensional datasets. The algorithm builds a random forest in which each decision tree is grown randomly: at each node, it picks a feature randomly, then it picks a random threshold value (between the min and max values) to split the dataset in two. The dataset gradually gets chopped into pieces this way, until all instances end up isolated from the other instances. Anomalies are usually far from other instances, so on average (across all the decision trees) they tend to get isolated in fewer steps than normal instances.

### *Local outlier factor (LOF)*

This algorithm is also good for outlier detection. It compares the density of instances around a given instance to the density around its neighbors. An anomaly is often more isolated than its  $k$ -nearest neighbors.

### *One-class SVM*

This algorithm is better suited for novelty detection. Recall that a kernelized SVM classifier separates two classes by first (implicitly) mapping all the instances to a high-dimensional space, then separating the two classes using a linear SVM classifier within this high-dimensional space (see the online chapter on SVMs at <https://homl.info>). Since we just have one class of instances, the one-class SVM algorithm instead tries to separate the instances in high-dimensional space from the origin. In the original space, this will correspond to finding a small region that encompasses all the instances. If a new instance does not fall within this region, it is an anomaly. There are a few hyperparameters to tweak: the usual ones for a kernelized SVM, plus a margin hyperparameter that corresponds to the probability of a new instance being mistakenly considered as novel when it is

in fact normal. It works great, especially with high-dimensional datasets, but like all SVMs it does not scale to large datasets.

*PCA and other dimensionality reduction techniques with an `inverse_transform()` method*

If you compare the reconstruction error of a normal instance with the reconstruction error of an anomaly, the latter will usually be much larger. This is a simple and often quite efficient anomaly detection approach (see this chapter's exercises for an example).

## Exercises

1. How would you define clustering? Can you name a few clustering algorithms?
2. What are some of the main applications of clustering algorithms?
3. Describe two techniques to select the right number of clusters when using  $k$ -means.
4. What is label propagation? Why would you implement it, and how?
5. Can you name two clustering algorithms that can scale to large datasets? And two that look for regions of high density?
6. Can you think of a use case where active learning would be useful? How would you implement it?
7. What is the difference between anomaly detection and novelty detection?
8. What is a Gaussian mixture? What tasks can you use it for?
9. Can you name two techniques to find the right number of clusters when using a Gaussian mixture model?
10. The classic Olivetti faces dataset contains 400 grayscale  $64 \times 64$ -pixel images of faces. Each image is flattened to a 1D vector of size 4,096. Forty different people were photographed (10 times each), and the usual task is to train a model that can predict which person is represented in each picture. Load the dataset using the `sklearn.datasets.fetch_olivetti_faces()` function, then split it into a training set, a validation set, and a test set (note that the dataset is already scaled between 0 and 1). Since the dataset is quite small, you will probably want to use stratified sampling to ensure that there are the same number of images per person in each set. Next, cluster the images using  $k$ -means, and ensure that you have a good number of clusters (using one of the techniques discussed in this chapter). Visualize the clusters: do you see similar faces in each cluster?
11. Continuing with the Olivetti faces dataset, train a classifier to predict which person is represented in each picture, and evaluate it on the validation set. Next, use  $k$ -means as a dimensionality reduction tool, and train a classifier on the

reduced set. Search for the number of clusters that allows the classifier to get the best performance: what performance can you reach? What if you append the features from the reduced set to the original features (again, searching for the best number of clusters)?

12. Train a Gaussian mixture model on the Olivetti faces dataset. To speed up the algorithm, you should probably reduce the dataset's dimensionality (e.g., use PCA, preserving 99% of the variance). Use the model to generate some new faces (using the `sample()` method), and visualize them (if you used PCA, you will need to use its `inverse_transform()` method). Try to modify some images (e.g., rotate, flip, darken) and see if the model can detect the anomalies (i.e., compare the output of the `score_samples()` method for normal images and for anomalies).
13. Some dimensionality reduction techniques can also be used for anomaly detection. For example, take the Olivetti faces dataset and reduce it with PCA, preserving 99% of the variance. Then compute the reconstruction error for each image. Next, take some of the modified images you built in the previous exercise and look at their reconstruction error: notice how much larger it is. If you plot a reconstructed image, you will see why: it tries to reconstruct a normal face.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.



## PART II

---

# Neural Networks and Deep Learning



---

# Introduction to Artificial Neural Networks

Birds inspired us to fly, burdock plants inspired Velcro, and nature has inspired countless more inventions. It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the logic that sparked *artificial neural networks* (ANNs), machine learning models inspired by the networks of biological neurons found in our brains. However, although planes were inspired by birds, they don't have to flap their wings to fly. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether (e.g., by saying "units" rather than "neurons"), lest we restrict our creativity to biologically plausible systems.<sup>1</sup>

ANNs are at the very core of deep learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex machine learning tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple's Siri or Google Assistant) and chatbots (e.g., ChatGPT or Claude), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning how proteins fold (DeepMind's AlphaFold).

This chapter introduces artificial neural networks, starting with a quick tour of the very first ANN architectures and leading up to multilayer perceptrons (MLPs), which are heavily used today (many other architectures will be explored in the following chapters). In this chapter, we will implement simple MLPs using Scikit-Learn to get our feet wet, and in the next chapter we will switch to PyTorch, as it is a much more flexible and efficient library for neural nets.

---

<sup>1</sup> You can get the best of both worlds by being open to biological inspirations without being afraid to create biologically unrealistic models, as long as they work well.

Now let's go back in time to the origins of artificial neural networks.

## From Biological to Artificial Neurons

Surprisingly, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their [landmark paper](#),<sup>2</sup> “A Logical Calculus of Ideas Immanent in Nervous Activity”, McCulloch and Pitts presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using *propositional logic*. This was the first artificial neural network architecture. Since then many other architectures have been invented, as you will see.

The early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear in the 1960s that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere, and ANNs entered a long winter. In the early 1980s, new architectures were invented and better training techniques were developed, sparking a revival of interest in *connectionism*, the study of neural networks. But progress was slow, and by the 1990s other powerful machine learning techniques had been invented, such as support vector machines. These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold.

We are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? Well, here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore’s law (the number of components in integrated circuits has doubled about every 2 years over the last 50 years), but also thanks to the gaming industry, which has stimulated the production of powerful *graphical processing units* (GPUs) by the millions: GPU cards were initially designed to accelerate graphics, but it turns out that neural networks perform similar computations (such as large matrix multiplications), so they can also be accelerated using GPUs. Moreover, cloud platforms have made this power accessible to everyone.

---

<sup>2</sup> Warren S. McCulloch and Walter Pitts, “A Logical Calculus of the Ideas Immanent in Nervous Activity”, *The Bulletin of Mathematical Biology* 5, no. 4 (1943): 115–113.

- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have had a huge positive impact.
- Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is not a big problem in practice, especially for larger neural networks: the local optima often perform almost as well as the global optimum.
- The invention of the Transformer architecture in 2017 (see [Chapter 15](#)) has been a game changer: it can process and generate all sorts of data (e.g., text, images, audio) unlike earlier, more specialized, architectures, and it performs great across a wide variety of tasks from robotics to protein folding. Moreover, it scales rather well, which has made it possible to train very large *foundation models* that can be reused across many different tasks, possibly with a bit of fine-tuning (that's transfer learning), or just by prompting the model in the right way (that's *in-context learning*, or ICL). For instance, you can give it a few examples of the task at hand (that's *few-shot learning*, or FSL), or ask it to reason step-by-step (that's *chain-of-thought* prompting, or CoT). It's a new world!
- ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products. AI is no longer just powering products in the shadows: since chatbots such as ChatGPT were released, the general public is now directly interacting daily with AI assistants, and the big tech companies are competing fiercely to grab this gigantic market: the pace of innovation is wild.

## Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron (represented in [Figure 9-1](#)). It is an unusual-looking cell mostly found in animal brains. It's composed of a *cell body* containing the nucleus and most of the cell's complex components, many branching extensions called *dendrites*, plus one very long extension called the *axon*. The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called *telodendria*, and at the tip of these branches are minuscule structures called *synaptic terminals* (or simply *synapses*), which are connected to the dendrites or cell bodies of other neurons.<sup>3</sup> Biological neurons produce short electrical impulses called *action potentials* (APs, or just *signals*), which travel along

---

<sup>3</sup> They are not actually attached, just so close that they can very quickly exchange chemical signals.

the axons and make the synapses release chemical signals called *neurotransmitters*. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).

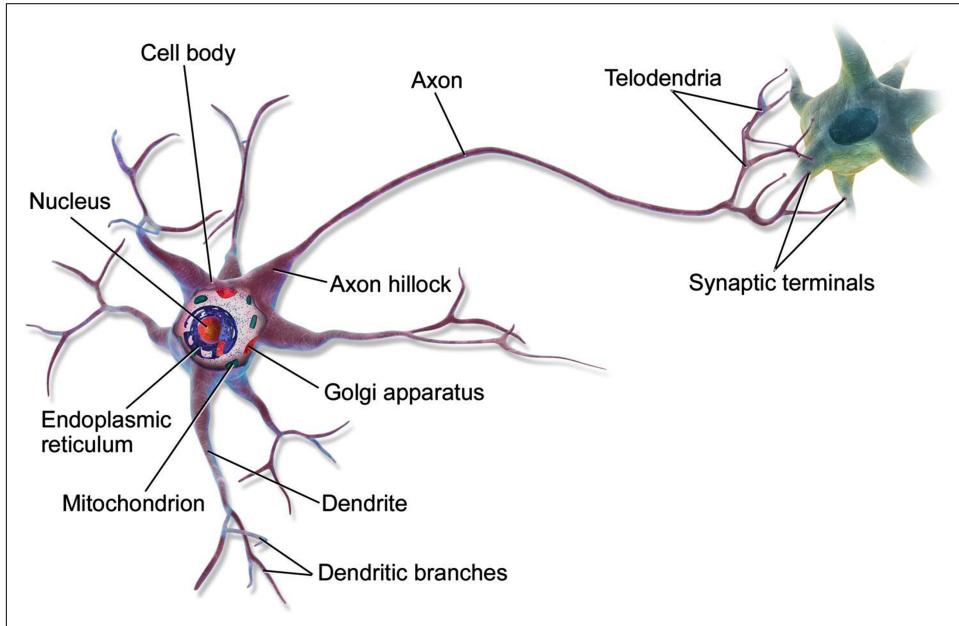


Figure 9-1. A biological neuron<sup>4</sup>

Thus, individual biological neurons seem to behave in a simple way, but they're organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural networks (BNNs)<sup>5</sup> is the subject of active research, but some parts of the brain have been mapped. These efforts show that neurons are often organized in consecutive layers, especially in the cerebral cortex (the outer layer of the brain), as shown in Figure 9-2.

<sup>4</sup> Image by Bruce Blaus (Creative Commons 3.0). Reproduced from <https://en.wikipedia.org/wiki/Neuron>.

<sup>5</sup> In the context of machine learning, the phrase "neural networks" generally refers to ANNs, not BNNs.

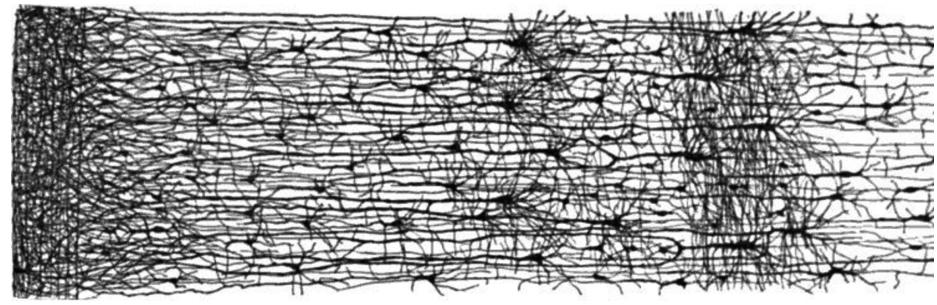


Figure 9-2. Multiple layers in a biological neural network (human cortex)<sup>6</sup>

## Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an *artificial neuron*: it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active. In their paper, McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that can compute any logical proposition you want. To see how such a network works, let's build a few ANNs that perform various logical computations (see Figure 9-3), assuming that a neuron is activated when at least two of its input connections are active.

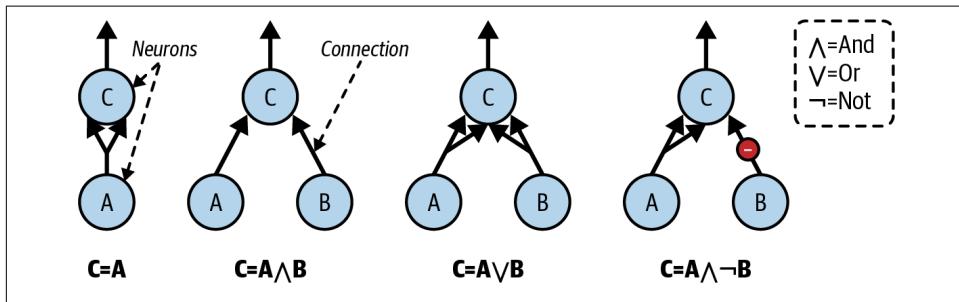


Figure 9-3. ANNs performing simple logical computations

Let's see what these networks do:

- The first network on the left is the identity function: if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A); but if neuron A is off, then neuron C is off as well.

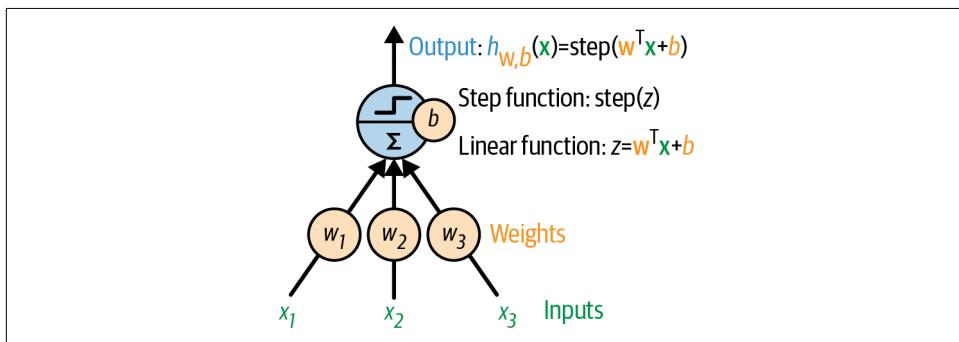
<sup>6</sup> Drawing of a cortical lamination by S. Ramon y Cajal (public domain). Reproduced from [https://en.wikipedia.org/wiki/Cerebral\\_cortex](https://en.wikipedia.org/wiki/Cerebral_cortex).

- The second network performs a logical AND: neuron C is activated only when both neurons A and B are activated (a single input signal is not enough to activate neuron C).
- The third network performs a logical OR: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

You can imagine how these networks can be combined to compute complex logical expressions (see the exercises at the end of the chapter for an example).

## The Perceptron

The *perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron (see [Figure 9-4](#)) called a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU first computes a linear function of its inputs:  $z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b = \mathbf{w}^T \mathbf{x} + b$ . Then it applies a *step function* to the result:  $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$ . So it's almost like logistic regression, except it uses a step function instead of the logistic function.<sup>7</sup> Just like in logistic regression, the model parameters are the input weights  $\mathbf{w}$  and the bias term  $b$ .



*Figure 9-4. TLU: an artificial neuron that computes a weighted sum of its inputs  $\mathbf{w}^T \mathbf{x}$ , plus a bias term  $b$ , then applies a step function*

<sup>7</sup> Logistic regression and the logistic function were introduced in [Chapter 4](#), along with several other concepts that we will heavily rely on in this chapter, including softmax, cross-entropy, gradient descent, early stopping, and more, so please make sure to read it first.

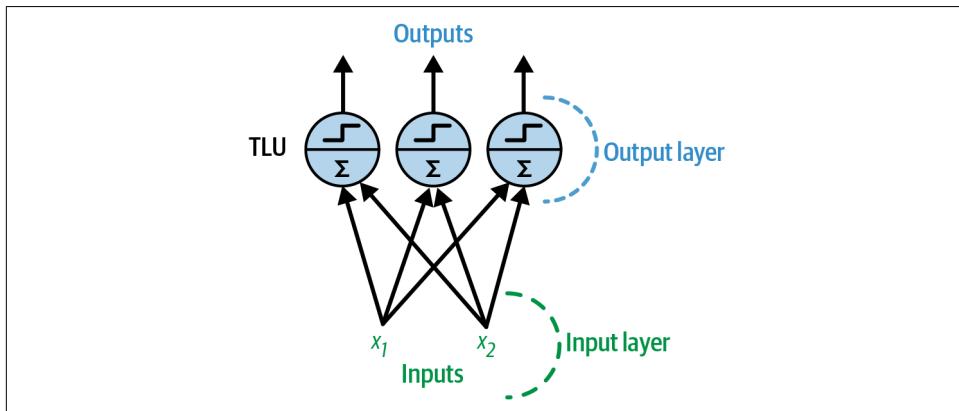
The most common step function used in perceptrons is the *Heaviside step function* (see [Equation 9-1](#)). Sometimes the sign function is used instead.

*Equation 9-1. Common step functions used in perceptrons (assuming threshold = 0)*

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

A single TLU can be used for simple linear binary classification. It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class. This may remind you of logistic regression ([Chapter 4](#)) or linear SVM classification (see the online chapter on SVMs at <https://homl.info>). You could, for example, use a single TLU to classify iris flowers based on petal length and width. Training such a TLU would require finding the right values for  $w_1$ ,  $w_2$ , and  $b$  (the training algorithm is discussed shortly).

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. Such a layer is called a *fully connected layer*, or a *dense layer*. The inputs constitute the *input layer*. And since the layer of TLUs produces the final outputs, it is called the *output layer*. For example, a perceptron with two inputs and three outputs is represented in [Figure 9-5](#).



*Figure 9-5. Architecture of a perceptron with two inputs and three output neurons*

This perceptron can classify instances simultaneously into three different binary classes, which makes it a multilabel classifier. It may also be used for multiclass classification.

Thanks to the magic of linear algebra, [Equation 9-2](#) can be used to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

*Equation 9-2. Computing the outputs of a fully connected layer*

$$\hat{\mathbf{Y}} = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

In this equation:

- $\hat{\mathbf{Y}}$  is the output matrix. It has one row per instance and one column per neuron.
- $\mathbf{X}$  is the input matrix. It has one row per instance and one column per input feature.
- The weight matrix  $\mathbf{W}$  contains all the connection weights. It has one row per input feature and one column per neuron.<sup>8</sup>
- The bias vector  $\mathbf{b}$  contains all the bias terms: one per neuron.
- The function  $\phi$  is called the *activation function*: when the artificial neurons are TLUs, it is a step function (we will discuss other activation functions shortly).



In mathematics, the sum of a matrix and a vector is undefined. However, in data science, we allow “broadcasting”: adding a vector to a matrix means adding it to every row in the matrix. So,  $\mathbf{X}\mathbf{W} + \mathbf{b}$  first multiplies  $\mathbf{X}$  by  $\mathbf{W}$ —which results in a matrix with one row per instance and one column per output—then adds the vector  $\mathbf{b}$  to every row of that matrix, which adds each bias term to the corresponding output, for every instance. Moreover,  $\phi$  is then applied itemwise to each item in the resulting matrix.

So, how is a perceptron trained? The perceptron training algorithm proposed by Rosenblatt was largely inspired by *Hebb's rule*. In his 1949 book, *The Organization of Behavior* (Wiley), Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. Siegrid Löwel later summarized Hebb's idea in the catchy phrase, “Cells that fire together, wire together”; that is, the connection weight between two neurons tends to increase when they fire simultaneously. This rule later became known as Hebb's rule (or *Hebbian learning*). Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction; the perceptron learning rule reinforces connections that help reduce the error. More specifically, the

---

<sup>8</sup> In some libraries, such as PyTorch, the weight matrix is transposed, so there's one row per neuron, and one column per input feature.

perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction. The rule is shown in [Equation 9-3](#).

*Equation 9-3. Perceptron learning rule (weight update)*

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

In this equation:

- $w_{i,j}$  is the connection weight between the  $i^{\text{th}}$  input and the  $j^{\text{th}}$  neuron.
- $x_i$  is the  $i^{\text{th}}$  input value of the current training instance.
- $\hat{y}_j$  is the output of the  $j^{\text{th}}$  output neuron for the current training instance.
- $y_j$  is the target output of the  $j^{\text{th}}$  output neuron for the current training instance.
- $\eta$  is the learning rate (see [Chapter 4](#)).

The decision boundary of each output neuron is linear, so perceptrons are incapable of learning complex patterns (just like logistic regression classifiers). However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm will converge to a solution.<sup>9</sup> This is called the *perceptron convergence theorem*.

Scikit-Learn provides a `Perceptron` class that can be used pretty much as you would expect—for example, on the iris dataset (introduced in [Chapter 4](#)):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0) # Iris setosa

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new) # predicts True and False for these 2 flowers
```

You may have noticed that the perceptron learning algorithm strongly resembles stochastic gradient descent (introduced in [Chapter 4](#)). In fact, Scikit-Learn's `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters:

---

<sup>9</sup> Note that this solution is not unique: when data points are linearly separable, there is an infinity of hyperplanes that can separate them.

```
loss="perceptron", learning_rate="constant", eta0=1 (the learning rate), and  
penalty=None (no regularization).
```



Contrary to logistic regression classifiers, perceptrons do not output a class probability. This is one reason to prefer logistic regression over perceptrons. Moreover, perceptrons do not use any regularization by default, and training stops as soon as there are no more prediction errors on the training set, so the model typically does not generalize as well as logistic regression or a linear SVM classifier. However, perceptrons may train a bit faster.

In their 1969 monograph, *Perceptrons*, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of perceptrons—in particular, the fact that they are incapable of solving some trivial problems (e.g., the *exclusive OR* (XOR) classification problem; see the left side of Figure 9-6). This is true of any other linear classification model (such as logistic regression classifiers), but researchers had expected much more from perceptrons, and some were so disappointed that they dropped neural networks altogether in favor of more formal approaches such as logic, problem solving, and search. The lack of practical applications also didn't help.

It turns out that some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons. The resulting ANN is called a *multilayer perceptron* (MLP).

## The Multilayer Perceptron and Backpropagation

An MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the righthand side of Figure 9-6: with inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1. Try verifying that this network indeed solves the XOR problem!<sup>10</sup>

An MLP is composed of one input layer, one or more layers of artificial neurons (originally TLUs) called *hidden layers*, and one final layer of artificial neurons called the *output layer* (see Figure 9-7). The layers close to the input layer are usually called the *lower layers*, and the ones close to the outputs are usually called the *upper layers*.

---

<sup>10</sup> For example, when the inputs are (0, 1) the lower-left neuron computes  $0 \times 1 + 1 \times 1 - 3 / 2 = -1 / 2$ , which is negative, so it outputs 0. The lower-right neuron computes  $0 \times 1 + 1 \times 1 - 1 / 2 = 1 / 2$ , which is positive, so it outputs 1. The output neuron receives the outputs of the first two neurons as its inputs, so it computes  $0 \times (-1) + 1 \times 1 - 1 / 2 = 1 / 2$ . This is positive, so it outputs 1.

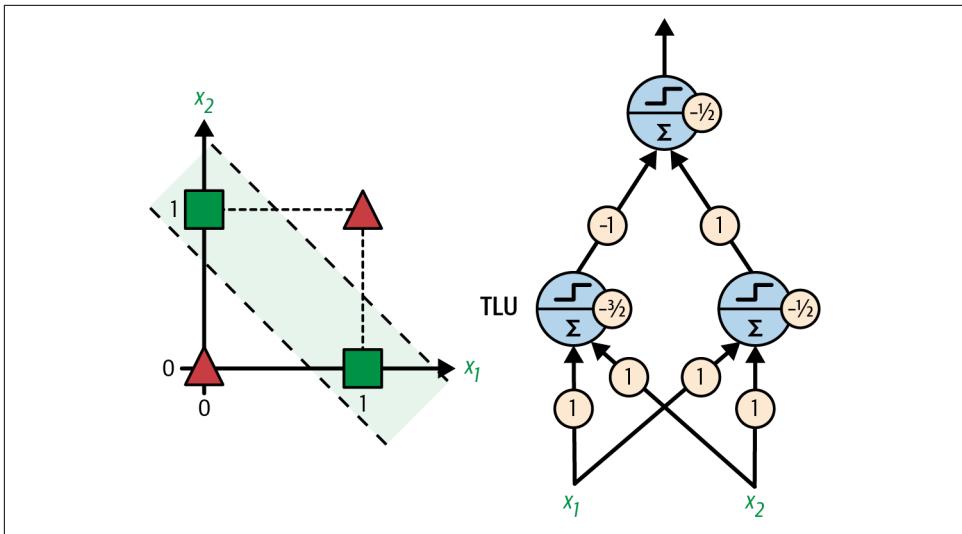


Figure 9-6. XOR classification problem and an MLP that solves it

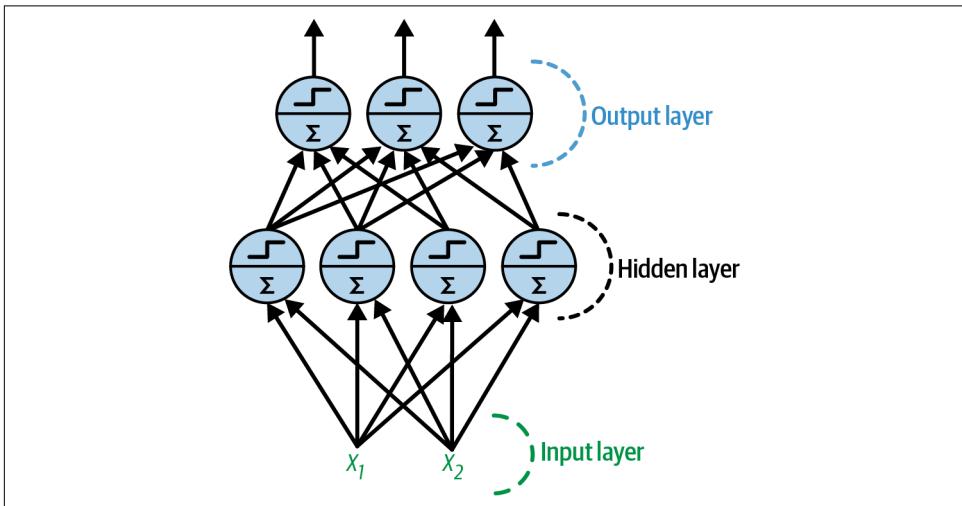


Figure 9-7. Architecture of a multilayer perceptron with two inputs, one hidden layer of four neurons, and three output neurons



The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

When an ANN contains a deep stack of hidden layers,<sup>11</sup> it is called a *deep neural network* (DNN). The field of deep learning studies DNNs, and more generally it is interested in models containing deep stacks of computations. Even so, many people talk about deep learning whenever neural networks are involved (even shallow ones).

For many years researchers struggled to find a way to train MLPs, without success. In the early 1960s several researchers discussed the possibility of using gradient descent to train neural networks, but as we saw in [Chapter 4](#), this requires computing the gradients of the model's error with regard to the model parameters; it wasn't clear at the time how to do this efficiently with such a complex model containing so many parameters, especially with the computers they had back then.

Then, in 1970, a researcher named Seppo Linnainmaa introduced in his master's thesis a technique to compute all the gradients automatically and efficiently. This algorithm is now called *reverse-mode automatic differentiation* (or *reverse-mode autodiff* for short). In just two passes through the network (one forward, one backward), it is able to compute the gradients of the neural network's error with regard to every single model parameter. In other words, it can find out how each connection weight and each bias should be tweaked in order to reduce the neural network's error. These gradients can then be used to perform a gradient descent step. If you repeat this process of computing the gradients automatically and taking a gradient descent step, the neural network's error will gradually drop until it eventually reaches a minimum. This combination of reverse-mode autodiff and gradient descent is now called *backpropagation* (or *backprop* for short).

Here's an analogy: imagine you are learning to shoot a basketball into the hoop. You throw the ball (that's the forward pass), and you observe that it went far off to the right side (that's the error computation), then you consider how you can change your body position to throw the ball a bit less to the right next time (that's the backward pass): you realize that your arm will need to rotate a bit counterclockwise, and probably your whole upper body as well, which in turn means that your feet should turn too (notice how we're going down the "layers"). Once you've thought it through, you actually move your body: that's the gradient descent step. The smaller the errors, the smaller the adjustments. As you repeat the whole process many times, the error gradually gets smaller, and after a few hours of practice, you manage to get the ball through the hoop every time. Good job!

---

<sup>11</sup> In the 1990s, an ANN with more than two hidden layers was considered deep. Nowadays, it is common to see ANNs with dozens of layers, or even hundreds, so the definition of "deep" is quite fuzzy.



There are various autodiff techniques, with different pros and cons. Reverse-mode autodiff is well suited when the function to differentiate has many variables (e.g., connection weights and biases) and few outputs (e.g., one loss). If you want to learn more about autodiff, check out [Appendix A](#).

Backpropagation can actually be applied to all sorts of computational graphs, not just neural networks: indeed, Linnainmaa's master's thesis was not about neural nets at all, it was more general. It was several more years before backprop started to be used to train neural networks, but it still wasn't mainstream. Then, in 1985, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a [paper](#)<sup>12</sup> analyzing how backpropagation allows neural networks to learn useful internal representations. Their results were so impressive that backpropagation was quickly popularized in the field. Over 40 years later, it is still by far the most popular training technique for neural networks.

Let's run through how backpropagation works again in a bit more detail:

- It handles one mini-batch at a time, and goes through the full training set multiple times. If each mini-batch contains 32 instances, and each instance has 100 features, then the mini-batch will be represented as a matrix with 32 rows and 100 columns. Each pass through the training set is called an *epoch*.
- For each mini-batch, the algorithm computes the output of all the neurons in the first hidden layer using [Equation 9-2](#). If the layer has 50 neurons, then its output is a matrix with one row per sample in the mini-batch (e.g., 32), and 50 columns (i.e., one per neuron). This matrix is then passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the *forward pass*: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.
- Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).
- Then it computes how much each output layer parameter contributed to the error. This is done analytically by applying the *chain rule* (one of the most fundamental rules in calculus), which makes this step fast and precise. The result is one gradient per parameter.

---

<sup>12</sup> David Rumelhart et al., "Learning Internal Representations by Error Propagation" (Defense Technical Information Center technical report, September 1985).

- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until it reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights and biases in the network by propagating the error gradient backward through the network (hence the name of the algorithm).
- Finally, the algorithm performs a gradient descent step to tweak all the connection weights and bias terms in the network, using the error gradients it just computed.



It is important to initialize all the hidden layers' connection weights randomly, or else training will fail. For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical. In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart. If instead you randomly initialize the weights, you *break the symmetry* and allow backpropagation to train a diverse team of neurons.

In short, backpropagation makes predictions for a mini-batch (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each parameter (reverse pass), and finally tweaks the connection weights and biases to reduce the error (gradient descent step).

In order for backprop to work properly, Rumelhart and his colleagues made a key change to the MLP's architecture: they replaced the step function with the logistic function,  $\sigma(z) = 1 / (1 + \exp(-z))$ , also called the *sigmoid* function. This was essential because the step function contains only flat segments, so there is no gradient to work with (gradient descent cannot move on a flat surface), while the sigmoid function has a well-defined nonzero derivative everywhere, allowing gradient descent to make some progress at every step. In fact, the backpropagation algorithm works well with many other activation functions, not just the sigmoid function. Here are two other popular choices:

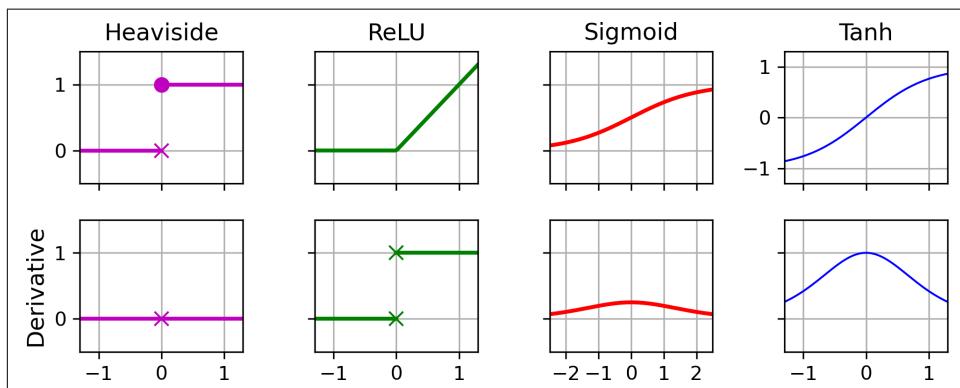
*The hyperbolic tangent function:  $\tanh(z) = 2\sigma(2z) - 1$*

Just like the sigmoid function, this activation function is S-shaped, continuous, and differentiable, but its output value ranges from  $-1$  to  $1$  (instead of  $0$  to  $1$  in the case of the sigmoid function). That range tends to make each layer's output more or less centered around  $0$  at the beginning of training, which often helps speed up convergence.

*The rectified linear unit function:  $\text{ReLU}(z) = \max(0, z)$*

The ReLU function is continuous but unfortunately not differentiable at  $z = 0$  (the slope changes abruptly, which can make gradient descent bounce around), and its derivative is 0 for  $z < 0$ . In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default for most architectures (except the Transformer architecture, as we will see in [Chapter 15](#)).<sup>13</sup> Importantly, the fact that it does not have a maximum output value helps reduce some issues during gradient descent (we will come back to this in [Chapter 11](#)).

These popular activation functions and their derivatives are represented in [Figure 9-8](#). But wait! Why do we need activation functions in the first place? Well, if you chain several linear transformations, all you get is a linear transformation. For example, if  $f(x) = 2x + 3$  and  $g(x) = 5x - 1$ , then chaining these two linear functions gives you another linear function:  $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$ . So if you don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that. Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.



*Figure 9-8. Activation functions (left) and their derivatives (right)*

OK! You know where neural nets came from, what the MLP architecture looks like, and how it computes its outputs. You've also learned about the backpropagation algorithm. It's time to see MLPs in action!

<sup>13</sup> Biological neurons seem to implement a roughly sigmoid (*S*-shaped) activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that ReLU generally works better in ANNs. This is one of the cases where the biological analogy was perhaps misleading.

# Building and Training MLPs with Scikit-Learn

MLPs can tackle a wide range of tasks, but the most common are regression and classification. Scikit-Learn can help with both of these. Let's start with regression.

## Regression MLPs

How would you build an MLP for a regression task? Well, if you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value. For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So, you end up with four output neurons.

Scikit-Learn includes an `MLPRegressor` class, so let's use it to build an MLP with three hidden layers composed of 50 neurons each, and train it on the California housing dataset. For simplicity, we will use Scikit-Learn's `fetch_california_housing()` function to load the data. This dataset is simpler than the one we used in [Chapter 2](#), since it contains only numerical features (there is no `ocean_proximity` feature), and there are no missing values. The targets are also scaled down: each unit represents \$100,000. Let's start by importing everything we will need:

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import root_mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
```

Next, let's fetch the California housing dataset and split it into a training set and a test set:

```
housing = fetch_california_housing()
X_train, X_test, y_train, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
```

Now let's create an `MLPRegressor` model with 3 hidden layers composed of 50 neurons each. The first hidden layer's input size (i.e., the number of rows in its weights matrix) and the output layer's output size (i.e., the number of columns in its weights matrix) will adjust automatically to the dimensionality of the inputs and targets, respectively, when training starts. The model uses the ReLU activation function in all hidden layers, and no activation function at all on the output layer. We also set `verbose=True` to get details on the model's progress during training:

```
mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], early_stopping=True,
                       verbose=True, random_state=42)
```

Since neural nets can have a *lot* of parameters, they have a tendency to overfit the training set. To reduce this risk, one option is to use early stopping (introduced in [Chapter 4](#)): when we set `early_stopping=True`, the `MLPRegressor` class automatically sets aside 10% of the training data and uses it to evaluate the model at each epoch (you can adjust the validation set's size by setting `validation_fraction`). If the validation score stops improving for 10 epochs, training automatically stops (you can tweak this number of epochs by setting `n_iter_no_change`).

Now let's create a pipeline to standardize the input features before sending them to the `MLPRegressor`. This is very important because gradient descent does not converge very well when the features have very different scales, as we saw in [Chapter 4](#). We can then train the model! The `MLPRegressor` class uses a variant of gradient descent called *Adam* (see [Chapter 11](#)) to minimize the mean squared error. It also uses a tiny bit of  $\ell_2$  regularization (you can control its strength via the `alpha` hyperparameter, which defaults to 0.0001):

```
>>> pipeline = make_pipeline(StandardScaler(), mlp_reg)
>>> pipeline.fit(X_train, y_train)
Iteration 1, loss = 0.85190332
Validation score: 0.534299
Iteration 2, loss = 0.28288639
Validation score: 0.651094
[...]
Iteration 45, loss = 0.12960481
Validation score: 0.788517
Validation score did not improve more than tol=0.000100 for 10 consecutive
epochs. Stopping.
```

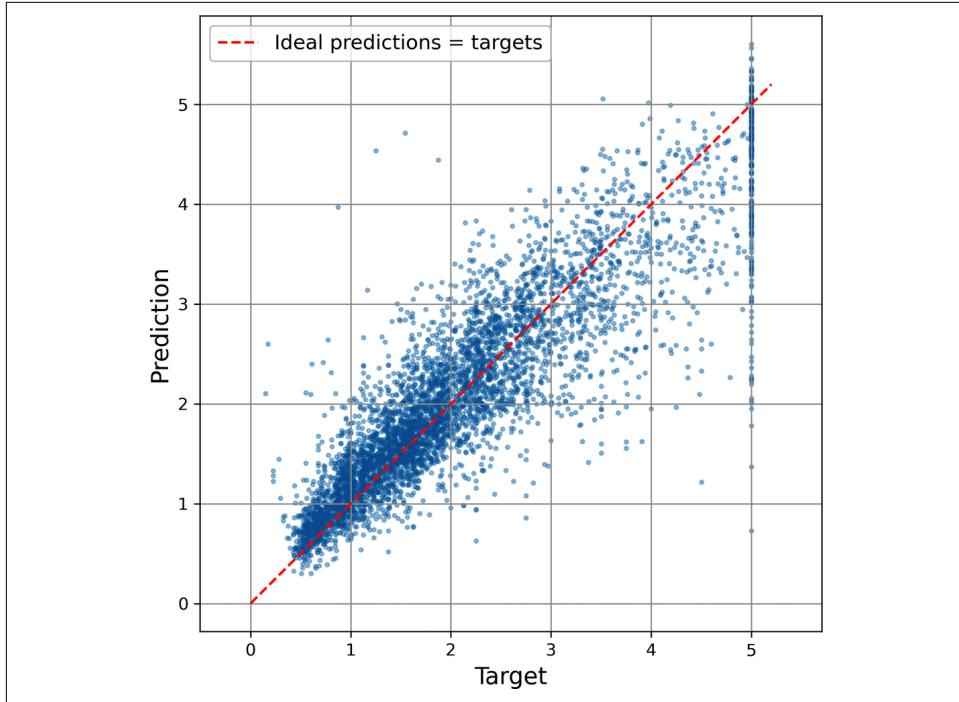
And there you go, you just trained your very first MLP! It required 45 epochs, and as you can see, the training loss went down at each epoch. This loss corresponds to [Equation 4-9](#) divided by 2, so you must multiply it by 2 to get the MSE (although not exactly because the loss includes the  $\ell_2$  regularization term). The validation score generally went up at each epoch. Like every regressor in Scikit-Learn, `MLPRegressor` uses the  $R^2$  score by default for evaluation—that's what the `score()` method returns. As we saw in [Chapter 2](#), the  $R^2$  score measures the ratio of the variance that is explained by the model. In this case, it reaches close to 80% on the validation set, which is fairly good for this task:

```
>>> mlp_reg.best_validation_score_
0.791536125425778
```

Let's evaluate the RMSE on the test set:

```
>>> y_pred = pipeline.predict(X_test)
>>> rmse = root_mean_squared_error(y_test, y_pred)
>>> rmse
0.5327699946812925
```

We get a test RMSE of about 0.53, which is comparable to what you would get with a random forest classifier. Not too bad for a first try! [Figure 9-9](#) plots the model’s predictions versus the targets (on the test set). The dashed red line represents the ideal predictions (i.e., equal to the targets): most of the predictions are close to the targets, but there are still quite a few errors, especially for larger targets.



*Figure 9-9. MLP regressor’s predictions versus the targets*

Note that this MLP does not use any activation function for the output layer, so it’s free to output any value it wants. This is generally fine, but if you want to guarantee that the output is always positive, then you should use the ReLU activation function on the output layer, or the *softplus* activation function, which is a smooth variant of ReLU:  $\text{softplus}(z) = \log(1 + \exp(z))$ . Softplus is close to 0 when  $z$  is negative, and close to  $z$  when  $z$  is positive. Finally, if you want to guarantee that the predictions always fall within a given range of values, then you should use the sigmoid function or the hyperbolic tangent, and scale the targets to the appropriate range: 0 to 1 for sigmoid and -1 to 1 for tanh. Sadly, the `MLPRegressor` class does not support activation functions in the output layer.



Scikit-Learn does not offer GPU acceleration, and its neural net features are fairly limited. This is why we will switch to PyTorch starting in [Chapter 10](#). That said, it is quite convenient to be able to build and train a standard MLP in just a few lines of code using Scikit-Learn: it lets you tackle many complex tasks very quickly.

In general, the mean squared error is the right loss to use for a regression tasks, but if you have a lot of outliers in the training set, you may sometimes prefer to use the mean absolute error instead, or preferably the *Huber loss*, which is a combination of both: it is quadratic when the error is smaller than a threshold  $\delta$  (typically 1), but linear when the error is larger than  $\delta$ . The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error. Unfortunately, `MLPRegressor` only supports the MSE loss.

[Table 9-1](#) summarizes the typical architecture of a regression MLP.

*Table 9-1. Typical regression MLP architecture*

Hyperparameter	Typical value
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per target dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Huber if outliers

All right, MLPs can tackle regression tasks. What else can they do?

## Classification MLPs

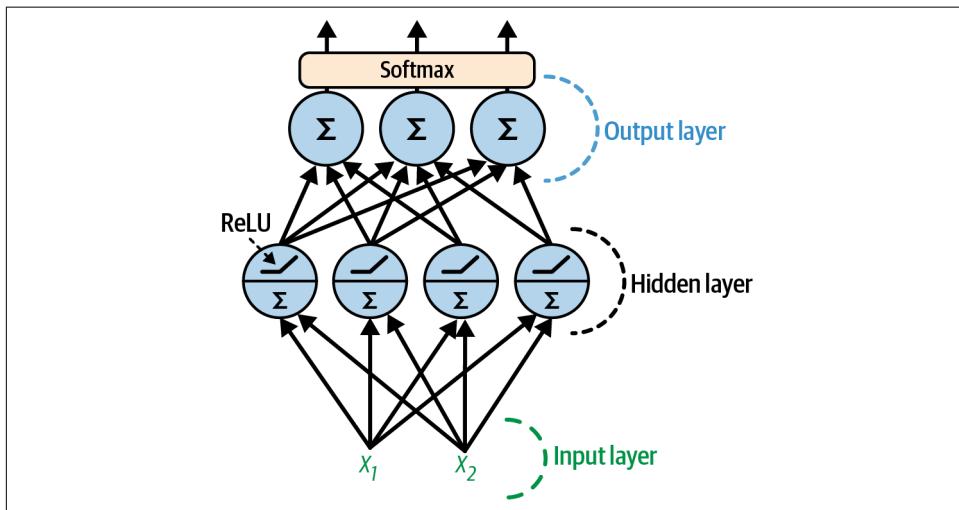
MLPs can also be used for classification problems. For a binary classification problem, you just need a single output neuron using the sigmoid activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. The estimated probability of the negative class is equal to one minus that number.

MLPs can also easily handle multilabel binary classification tasks (see [Chapter 3](#)). For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need two output neurons, both using the sigmoid activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the

output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have nonurgent ham, urgent ham, nonurgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see [Figure 9-10](#)). The softmax function (introduced in [Chapter 4](#)) will ensure that all the estimated probabilities are between 0 and 1, and that they add up to 1, since the classes are exclusive. As we saw in [Chapter 3](#), this is called multiclass classification.

Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (or *x-entropy* or log loss for short, see [Chapter 4](#)) is generally a good choice.



*Figure 9-10. A modern MLP (including ReLU and softmax) for classification*

[Table 9-2](#) summarizes the typical architecture of a classification MLP.

*Table 9-2. Typical classification MLP architecture*

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

As you might expect, Scikit-Learn offers an `MLPClassifier` class in the `sklearn.neural_network` package, which you can use for binary or multiclass classification. It is almost identical to the `MLPRegressor` class, except that its output layer uses the softmax activation function, and it minimizes the cross-entropy loss rather than the MSE. Moreover, the `score()` method returns the model's accuracy rather than the  $R^2$  score. Let's try it out.

We could tackle the iris dataset, but that task is too simple for a neural net: a linear model would do just as well and wouldn't risk overfitting. So let's instead tackle a more complex task: Fashion MNIST. This is a drop-in replacement of MNIST (introduced in [Chapter 3](#)). It has the exact same format as MNIST (70,000 grayscale images of  $28 \times 28$  pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits, so each class is much more diverse, and the problem turns out to be significantly more challenging than MNIST. For example, a simple linear model reaches about 92% accuracy on MNIST, but only about 83% on Fashion MNIST. Let's see if we can do better with an MLP.

First, let's load the dataset using the `fetch_openml()` function, very much like we did for MNIST in [Chapter 3](#). Note that the targets are represented as strings '`0`', '`1`', ..., '`9`', so we convert them to integers:

```
from sklearn.datasets import fetch_openml

fashion_mnist = fetch_openml(name="Fashion-MNIST", as_frame=False)
targets = fashion_mnist.target.astype(int)
```

The data is already shuffled, so we just take the first 60,000 images for training, and the last 10,000 for testing:

```
X_train, y_train = fashion_mnist.data[:60_000], targets[:60_000]
X_test, y_test = fashion_mnist.data[60_000:], targets[60_000:]
```

Each image is represented as a 1D integer array containing 784 pixel intensities ranging from 0 to 255. You can use the `plt.imshow()` function to plot an image, but first you need to reshape it to `[28, 28]`:

```
import matplotlib.pyplot as plt

X_sample = X_train[0].reshape(28, 28) # first image in the training set
plt.imshow(X_sample, cmap="binary")
plt.show()
```

If you run this code, you should see the ankle boot represented in the top-right corner of [Figure 9-11](#).



Figure 9-11. First four samples from each class in Fashion MNIST

With MNIST, when the label is equal to 5, it means that the image represents the handwritten digit 5. Easy. For Fashion MNIST, however, we need the list of class names to know what we are dealing with. Scikit-Learn does not provide it, so let's create it:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

We can now confirm that the first image in the training set represents an ankle boot:

```
>>> class_names[y_train[0]]
'Ankle boot'
```

We're ready to build the classification MLP:

```
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler

mlp_clf = MLPClassifier(hidden_layer_sizes=[300, 100], verbose=True,
                        early_stopping=True, random_state=42)
pipeline = make_pipeline(MinMaxScaler(), mlp_clf)
pipeline.fit(X_train, y_train)
accuracy = pipeline.score(X_test, y_test)
```

This code is very similar to the regression code we used earlier, but there are a few differences:

- Of course, it's a classification task so we use an `MLPClassifier` rather than an `MLPRegressor`.
- We use just two hidden layers with 300 and 100 neurons, respectively. You can try a different number of hidden layers, and change the number of neurons as well if you want.
- We also use a `MinMaxScaler` instead of a `StandardScaler`. We need it to shrink the pixel intensities down to the 0–1 range rather than 0–255: having features

in this range usually works better with the default hyperparameters used by `MLPClassifier`, such as its default learning rate and weight initialization scale. You might wonder why we didn't use a `StandardScaler`? Well some pixels don't vary much across images; for example, the pixels around the edges are almost always white. If we used the `StandardScaler`, these pixels would get scaled up to have the same variance as every other pixel: as a result, we would give more importance to these pixels than they probably deserve. Using the `MinMaxScaler` often works better than the `StandardScaler` for images (but your mileage may vary).

- Lastly, the `score()` function returns the model's accuracy.

If you run this code, you will find that the model reaches about 89.7% accuracy on the validation set during training (the exact value is given by `mlp_clf.best_validation_score_`), but it starts overfitting a bit toward the end, so it ends up at just 89.2% accuracy. When we evaluate the model on the test set, we get 87.1%, which is not bad for this task, although we can do better with other neural net architectures such as convolutional neural networks ([Chapter 12](#)).

You probably noticed that training was quite slow. That's because the hidden layers have a *lot* of parameters, so there are many computations to run at each iteration. For example, the first hidden layer has  $784 \times 300$  connection weights, plus 300 bias terms, which adds up to 235,500 parameters! All these parameters give the model quite a lot of flexibility to fit the training data, but it also means that there's a high risk of overfitting, especially when you do not have a lot of training data. In this case, you may want to use regularization techniques such as early stopping and  $\ell_2$  regularization.

Once the model is trained, you can use it to classify new images:

```
>>> X_new = X_test[:15] # let's pretend these are 15 new images
>>> mlp_clf.predict(X_new)
array([9, 2, 1, 1, 6, 1, 4, 6, 5, 7, 4, 5, 8, 3, 4])
```

All these predictions are correct, except for the one at index 12, which should be a 7 (sneaker) instead of a 8 (bag). You might want to know how confident the model was about these predictions, especially the bad one. For this, you can use `model.predict_proba()` instead of `model.predict()`, like we did in [Chapter 3](#):

```
>>> y_proba = mlp_clf.predict_proba(X_new)
>>> y_proba[12]
array([0., 0., 0., 0., 0., 0., 0., 1., 0.])
```

Hmm, that's not great: the model is telling us that it's 100% confident that the image represents a bag (index 8). So not only is the model wrong, it's 100% confident that it's right. In fact, across all 10,000 images in the test set, there are only 16 images that the model is less than 99.9% confident about, despite the fact that its accuracy is about

90%. That's why you should always treat estimated probabilities with a grain of salt: neural nets have a strong tendency to be overconfident, especially if they are trained for a bit too long.



The targets for classification tasks can be class indices (e.g., 3) or class probabilities, typically one-hot vectors (e.g., [0, 0, 0, 1, 0, 0, 0, 0, 0]). But if your model tends to be overconfident, you can try the *label smoothing* technique:<sup>14</sup> reduce the target class's probability slightly (e.g., from 1 down to 0.9) and distribute the rest evenly across the other classes (e.g., [0.1/9, 0.1/9, 0.1/9, 0.9, 0.1/9, 0.1/9, 0.1/9, 0.1/9, 0.1/9]).

Still, getting 90% accuracy on Fashion MNIST is pretty good. You could get even better performance by fine-tuning the hyperparameters, for example using `RandomizedSearchCV`, as we did in [Chapter 2](#). However, the search space is quite large, so it helps to know roughly where to look.

## Hyperparameter Tuning Guidelines

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network architecture, but even in a basic MLP you can change the number of layers, the number of neurons and the type of activation function to use in each layer, the weight initialization logic, the type of optimizer to use, its learning rate, the batch size, and more. What are some good values for these hyperparameters?

### Number of Hidden Layers

For many problems, you can begin with a single hidden layer and get reasonable results. An MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons. But deep networks have a much higher *parameter efficiency* than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets, allowing them to reach much better performance with the same amount of training data. This is because their layered structure enables them to reuse and compose features across multiple levels: for example, the first layer in a face classifier may learn to recognize low-level features such as dots, arcs, or straight lines; while the second layer may learn to combine these low-level features into higher-level features such as squares or circles; and the third layer may learn to combine these higher-level features into a mouth, an eye, or a nose; and the top layer would then be able to use these top-level features to classify faces.

---

<sup>14</sup> C. Szegedy et al., “Rethinking the Inception Architecture for Computer Vision”, CVPR 2016: 2818–2826.

Not only does this hierarchical architecture help DNNs converge faster to a good solution, but it also improves their ability to generalize to new datasets. For example, if you have already trained a model to recognize faces in pictures and you now want to train a new neural network to recognize hairstyles, you can kickstart the training by reusing the lower layers of the first network. Instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the values of the weights and biases of the lower layers of the first network. This way the network will not have to learn from scratch all the low-level structures that occur in most pictures; it will only have to learn the higher-level structures (e.g., hairstyles). This is called *transfer learning*.

In summary, for many problems you can start with just one or two hidden layers, and the neural network will work pretty well. For instance, you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total number of neurons, in roughly the same amount of training time. For more complex problems, you can ramp up the number of hidden layers until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers (or even hundreds, but not fully connected ones, as you will see in [Chapter 12](#)), and they need a huge amount of training data. You will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task. Training will then be a lot faster and require much less data.

## Number of Neurons per Hidden Layer

The number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, the MNIST task requires  $28 \times 28 = 784$  inputs and 10 output neurons.

As for the hidden layers, it used to be common to size them to form a pyramid, with fewer and fewer neurons at each layer—the rationale being that many low-level features can coalesce into far fewer high-level features. A typical neural network for MNIST might have 3 hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned because it seems that using the same number of neurons in all hidden layers performs just as well in most cases, or even better; plus, there is only one hyperparameter to tune, instead of one per layer. That said, depending on the dataset, it can sometimes help to make the first hidden layer a bit larger than the others.

Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. Alternatively, you can try building a model with slightly more layers and neurons than you actually need, then use early stopping

and other regularization techniques to prevent it from overfitting too much. Vincent Vanhoucke, a Waymo researcher and former Googler, has dubbed this the “stretch pants” approach: instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size. With this approach, you avoid bottleneck layers that could ruin your model. Indeed, if a layer has too few neurons, it will lack the computational capacity to model complex relationships, and it may not even have enough representational power to preserve all the useful information from the inputs. For example, if you apply PCA (introduced in [Chapter 7](#)) to the Fashion MNIST training set, you will find that you need 187 dimensions to preserve 95% of the variance in the data. So if you set the number of neurons in the first hidden layer to some greater number, say 200, you can be confident that this layer will not be a bottleneck. However, you don’t want to add too many neurons, or else the model will have too many parameters to optimize, and it will take more time and data to train.



In general, you will get more bang for your buck by increasing the number of layers rather than the number of neurons per layer.

That said, bottleneck layers are not always a bad thing. For example, limiting the dimensionality of the first hidden layers forces the neural net to keep only the most important dimensions, which can eliminate some of the noise in the data (but don’t go too far!). Also, having a bottleneck layer near the output layer can force the neural net to learn good representations of the data in the previous layers (i.e., packing more useful information in less space), which can help the neural net generalize, and can also be useful in and of itself for *representation learning*. We will get back to that in [Chapter 18](#).

## Learning Rate

The learning rate is a hugely important hyperparameter. In general, the optimal learning rate is about half of the maximum learning rate (i.e., the learning rate above which the training algorithm diverges, as we saw in [Chapter 4](#)). One way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate (e.g.,  $10^{-5}$ ) and gradually increasing it up to a very large value (e.g., 10). This is done by multiplying the learning rate by a constant factor at each iteration (e.g., by  $(10 / 10^{-5})^{1/500}$  to go from  $10^{-5}$  to 10 in 500 iterations). If you plot the loss as a function of the learning rate (using a log scale for the learning rate), you should see it dropping at first. But after a while, the learning rate will be too large, so the loss will shoot back up: the optimal learning rate is often a bit lower than the point at which the loss starts to climb (typically about 10 times lower than the turning

point). You can then reinitialize your model and train it normally using this good learning rate.



To change the learning rate during training when using Scikit-Learn, you must set the MLP's `warm_start` hyperparameter to `True`, and fit the model one batch at a time using `partial_fit()`, much like we did with the `SGDRegressor` in [Chapter 4](#). Simply update the learning rate at each iteration.

## Batch Size

The batch size can have a significant impact on your model's performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently (as we will see in [Chapter 10](#)), so the training algorithm will see more instances per second. Therefore, many researchers and practitioners recommend using the largest batch size that can fit in VRAM (video RAM, i.e., the GPU's memory). There's a catch, though: large batch sizes can sometimes lead to training instabilities, especially with smaller models and at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size. Yann LeCun once tweeted "Friends don't let friends use mini-batches larger than 32", citing a [2018 paper](#)<sup>15</sup> by Dominic Masters and Carlo Luschi which concluded that using small batches (from 2 to 32) was preferable because small batches led to better models in less training time.

However, other research points in the opposite direction. For example, in 2017, papers by [Elad Hoffer et al.](#)<sup>16</sup> and [Priya Goyal et al.](#)<sup>17</sup> showed that it is possible to use very large batch sizes (up to 8,192), along with various techniques such as warming up the learning rate (i.e., starting training with a small learning rate, then ramping it up), to obtain very short training times, without any generalization gap.

So one strategy is to use a large batch size, possibly with learning rate warmup, and if training is unstable or the final performance is disappointing, then try using a smaller batch size instead.

---

<sup>15</sup> Dominic Masters and Carlo Luschi, "Revisiting Small Batch Training for Deep Neural Networks", arXiv preprint arXiv:1804.07612 (2018).

<sup>16</sup> Elad Hoffer et al., "Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks", *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 1729–1739.

<sup>17</sup> Priya Goyal et al., "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv preprint arXiv:1706.02677 (2017).

## Other Hyperparameters

Here are two more hyperparameters you can tune if you have the computation budget and the time:

### *Optimizer*

Choosing a better optimizer than plain old mini-batch gradient descent (and tuning its hyperparameters) can help speed up training and sometimes reach better performance.

### *Activation function*

We discussed how to choose the activation function earlier in this chapter: in general, the ReLU activation function is a good default for all hidden layers. In some cases, replacing ReLU with another function can help.



The optimal learning rate depends on the other hyperparameters—especially the batch size—so if you modify any hyperparameter, make sure to tune the learning rate again.

For more best practices regarding tuning neural network hyperparameters, check out the excellent [2018 paper<sup>18</sup>](#) by Leslie Smith. The [Deep Learning Tuning Playbook](#) by Google researchers is also well worth reading. The free e-book [Machine Learning Yearning by Andrew Ng](#) also contains a wealth of practical advice.

Lastly, I highly recommend you go through exercise 1 at the end of this chapter. You will use a nice web interface to play with various neural network architectures and visualize their outputs. This will be very useful to better understand MLPs and grow a good intuition for the effects of each hyperparameter (number of layers and neurons, activation functions, and more).

This concludes our introduction to artificial neural networks and their implementation with Scikit-Learn. In the next chapter, we will switch to PyTorch, the leading open source library for neural networks, and we will use it to train and run MLPs much faster by exploiting the power of graphical processing units (GPUs). We will also start building more complex models, with multiple inputs and outputs.

---

<sup>18</sup> Leslie N. Smith, “A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay”, arXiv preprint arXiv:1803.09820 (2018).

# Exercises

1. This [neural network playground](#) is a great tool to build your intuitions without writing any code (it was built by the TensorFlow team, but there's nothing TensorFlow-specific about it; in fact, it doesn't even use TensorFlow). In this exercise, you will train several binary classifiers in just a few clicks, and tweak the model's architecture and its hyperparameters to gain some intuition on how neural networks work and what their hyperparameters do. Take some time to explore the following:
  - a. The patterns learned by a neural net. Try training the default neural network by clicking the Run button (top left). Notice how it quickly finds a good solution for the classification task. The neurons in the first hidden layer have learned simple patterns, while the neurons in the second hidden layer have learned to combine the simple patterns of the first hidden layer into more complex patterns. In general, the more layers there are, the more complex the patterns can be.
  - b. Activation functions. Try replacing the tanh activation function with a ReLU activation function, and train the network again. Notice that it finds a solution even faster, but this time the boundaries are linear. This is due to the shape of the ReLU function.
  - c. The risk of local minima. Modify the network architecture to have just one hidden layer with three neurons. Train it multiple times (to reset the network weights, click the Reset button next to the Play button). Notice that the training time varies a lot, and sometimes it even gets stuck in a local minimum.
  - d. What happens when neural nets are too small. Remove one neuron to keep just two. Notice that the neural network is now incapable of finding a good solution, even if you try multiple times. The model has too few parameters and systematically underfits the training set.
  - e. What happens when neural nets are large enough. Set the number of neurons to eight, and train the network several times. Notice that it is now consistently fast and never gets stuck. This highlights an important finding in neural network theory: large neural networks rarely get stuck in local minima, and even when they do, these local optima are often almost as good as the global optimum. However, they can still get stuck on long plateaus for a long time.
  - f. The risk of vanishing gradients in deep networks. Select the spiral dataset (the bottom-right dataset under "DATA"), and change the network architecture to have four hidden layers with eight neurons each. Notice that training takes much longer and often gets stuck on plateaus for long periods of time. Also notice that the neurons in the highest layers (on the right) tend to evolve faster than the neurons in the lowest layers (on the left). This problem, called

the *vanishing gradients* problem, can be alleviated with better weight initialization and other techniques, better optimizers (such as AdaGrad or Adam), or batch normalization (discussed in [Chapter 11](#)).

- g. Go further. Take an hour or so to play around with other parameters and get a feel for what they do to build an intuitive understanding about neural networks.
2. Draw an ANN using the original artificial neurons (like the ones in [Figure 9-3](#)) that computes  $A \oplus B$  (where  $\oplus$  represents the XOR operation). Hint:  $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$ .
3. Why is it generally preferable to use a logistic regression classifier rather than a classic perceptron (i.e., a single layer of threshold logic units trained using the perceptron training algorithm)? How can you tweak a perceptron to make it equivalent to a logistic regression classifier?
4. Why was the sigmoid activation function a key ingredient in training the first MLPs?
5. Name three popular activation functions. Can you draw them?
6. Suppose you have an MLP composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.
  - a. What is the shape of the input matrix  $\mathbf{X}$ ?
  - b. What are the shapes of the hidden layer's weight matrix  $\mathbf{W}_h$  and bias vector  $\mathbf{b}_h$ ?
  - c. What are the shapes of the output layer's weight matrix  $\mathbf{W}_o$  and bias vector  $\mathbf{b}_o$ ?
  - d. What is the shape of the network's output matrix  $\mathbf{Y}$ ?
  - e. Write the equation that computes the network's output matrix  $\mathbf{Y}$  as a function of  $\mathbf{X}$ ,  $\mathbf{W}_h$ ,  $\mathbf{b}_h$ ,  $\mathbf{W}_o$ , and  $\mathbf{b}_o$ .
7. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST, how many neurons do you need in the output layer, and which activation function should you use? What about for getting your network to predict housing prices, as in [Chapter 2](#)?
8. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
9. Can you list all the hyperparameters you can tweak in a basic MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem?

10. Train a deep MLP on the CoverType dataset. You can load it using `sklearn.datasets.fetch_covtype()`. See if you can get over 93% accuracy on the test set by fine-tuning the hyperparameters manually and/or using `RandomizedSearchCV`.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.



# Building Neural Networks with PyTorch

PyTorch is a powerful open source deep learning library developed by Facebook’s AI Research lab (FAIR, now called Meta AI). It is the Python successor of the Torch library, originally written in the Lua programming language. With PyTorch, you can build all sorts of neural network models and train them at scale using GPUs (or other hardware accelerators, as we will see). In many ways it is similar to NumPy, except it also supports hardware acceleration and autodiff (see [Chapter 9](#)), and includes optimizers and ready-to-use neural net components.

When PyTorch was released in 2016, Google’s TensorFlow library was by far the most popular: it was fast, it scaled well, and it could be deployed across many platforms. But its programming model was complex and static, making it difficult to use and debug. In contrast, PyTorch was designed from the ground up to provide a more flexible, Pythonic approach to building neural networks. In particular, as you will see, it uses dynamic computation graphs (also known as define-by-run), making it intuitive and easy to debug. PyTorch is also beautifully coded and documented, and focuses on its core task: making it easy to build and train high-performance neural networks. Last but not least, it leans strongly into the open source culture and benefits from an enthusiastic and dedicated community, and a rich ecosystem. In September 2022, PyTorch’s governance was even transferred to the PyTorch Foundation, a subsidiary of the Linux Foundation. All these qualities resonated well with researchers: PyTorch quickly became the most used framework in academia, and once a majority of deep learning papers were based on PyTorch, a large part of the industry was gradually converted as well.<sup>1</sup>

---

<sup>1</sup> To be fair, most of TensorFlow’s usability issues were fixed in version 2, and Google also launched JAX, which is well designed and extremely fast, so PyTorch still has some healthy competition. The good news is that the APIs of all these libraries have converged quite a bit, so switching from one to the other is much easier than it used to be.

In this chapter, you will learn how to train, evaluate, fine-tune, optimize, and save neural nets with PyTorch. We will start by getting familiar with the core building blocks of PyTorch, namely tensors and autograd, next we will test the waters by building and training a simple linear regression model, and then we will upgrade this model to a multilayer neural network, first for regression, then for classification. Along the way, we will see how to build custom neural networks with multiple inputs or outputs. Finally, we will discuss how to automatically fine-tune hyperparameters using the Optuna library, and how to optimize and export your models. Hop on board, we're diving into deep learning!



Colab runtimes come with a recent version of PyTorch preinstalled. However, if you prefer to install it on your own machine, please see the installation instructions at <https://hml.info/install-p>: this involves installing Python, many libraries, and a GPU driver (if you have one).

## PyTorch Fundamentals

The core data structure of PyTorch is the *tensor*.<sup>2</sup> It's a multidimensional array with a shape and a data type, used for numerical computations. Isn't that exactly like a NumPy array? Well, yes, it is! But a tensor also has two extra features: it can live on a GPU (or other hardware accelerators, as we will see), and it supports auto-differentiation. Every neural network we will build from now on will input and output tensors (much like Scikit-Learn models input and output NumPy arrays). So let's start by looking at how to create and manipulate tensors.

### PyTorch Tensors

First, let's import the PyTorch library:

```
>>> import torch
```

Next you can create a PyTorch tensor much like you would create a NumPy array. For example, let's create a  $2 \times 3$  array:

```
>>> X = torch.tensor([[1.0, 4.0, 7.0], [2.0, 3.0, 6.0]])
>>> X
tensor([[1., 4., 7.],
        [2., 3., 6.]])
```

---

<sup>2</sup> There are things called tensors in mathematics and physics, but ML tensors are simpler: they're really just multidimensional arrays for numerical computations, plus a few extra features.

Just like a NumPy array, a tensor can contain floats, integers, booleans, or complex numbers—just one data type per tensor. If you initialize a tensor with values of different types, then the most general one will be selected (i.e., complex > float > integer > bool). You can also select the data type explicitly when creating the tensor, for example `dtype=torch.float16` for 16-bit floats. Note that tensors of strings or objects are not supported.

You can get a tensor's shape and data type like this:

```
>>> X.shape  
torch.Size([2, 3])  
>>> X.dtype  
torch.float32
```

Indexing works just like for NumPy arrays:

```
>>> X[0, 1]  
tensor(4.)  
>>> X[:, 1]  
tensor([4., 3.])
```

You can also run all sorts of computations on tensors, and the API is conveniently similar to NumPy's: for example, there's `torch.abs()`, `torch.cos()`, `torch.exp()`, `torch.max()`, `torch.mean()`, `torch.sqrt()`, and so on. PyTorch tensors also have methods for most of these operations, so you can write `X.exp()` instead of `torch.exp(X)`. Let's try a few operations:

```
>>> 10 * (X + 1.0) # itemwise addition and multiplication  
tensor([[20., 50., 80.],  
       [30., 40., 70.]])  
>>> X.exp() # itemwise exponential  
tensor([[ 2.7183,  54.5981, 1096.6332],  
       [ 7.3891, 20.0855, 403.4288]])  
>>> X.mean()  
tensor(3.8333)  
>>> X.max(dim=0) # max values along dimension 0 (i.e., max value per column)  
torch.return_types.max(values=tensor([2., 4., 7.]), indices=tensor([1, 0, 0]))  
>>> X @ X.T # matrix transpose and matrix multiplication  
tensor([[66., 56.],  
       [56., 49.]])
```



PyTorch prefers the argument name `dim` in operations such as `max()`, but it also supports `axis` (as in NumPy or Pandas).

You can also convert a tensor to a NumPy array using the `numpy()` method, and create a tensor from a NumPy array:

```
>>> import numpy as np
>>> X.numpy()
array([[1., 4., 7.],
       [2., 3., 6.]], dtype=float32)
>>> torch.tensor(np.array([[1., 4., 7.], [2., 3., 6.]]))
tensor([[1., 4., 7.],
       [2., 3., 6.]], dtype=torch.float64)
```

Notice that the default precision for floats is 32 bits in PyTorch, whereas it's 64 bits in NumPy. It's generally better to use 32 bits in deep learning because this takes half the RAM and speeds up computations, and neural nets do not actually need the extra precision offered by 64-bit floats. So when calling the `torch.tensor()` function to convert a NumPy array to a tensor, it's best to specify `dtype=torch.float32`. Alternatively, you can use `torch.FloatTensor()` which automatically converts the array to 32 bits:

```
>>> torch.FloatTensor(np.array([[1., 4., 7.], [2., 3., 6.]]))
tensor([[1., 4., 7.],
       [2., 3., 6.]])
```



Both `torch.tensor()` and `torch.FloatTensor()` make a copy of the given NumPy array. If you prefer, you can use `torch.from_numpy()` which creates a tensor on the CPU that just uses the NumPy array's data directly, without copying it. But beware: modifying the NumPy array will also modify the tensor, and vice versa.

You can also modify a tensor in place using indexing and slicing, as with a NumPy array:

```
>>> X[:, 1] = -99
>>> X
tensor([[ 1., -99.,  7.],
       [ 2., -99.,  6.]])
```

PyTorch's API provides many in-place operations, such as `abs_()`, `sqrt_()`, and `zero_()`, which modify the input tensor directly: they can sometimes save some memory and speed up your models. For example, the `relu_()` method applies the ReLU activation function in place by replacing all negative values with 0s:

```
>>> X.relu_()
>>> X
tensor([[ 1.,  0.,  7.],
       [ 2.,  0.,  6.]])
```



PyTorch's in-place operations are easy to spot at a glance because their name always ends with an underscore. With very few exceptions (e.g., `zero_()`), removing the underscore gives you the regular operation (e.g., `abs_()` is in place, `abs()` is not).

We will cover many more operations as we go, but now let's look at how to use hardware acceleration to make computations much faster.

## Hardware Acceleration

PyTorch tensors can be copied easily to the GPU, assuming your machine has a compatible GPU, and you have the required libraries installed. On Colab, all you need to do is ensure that you are using a GPU runtime: for this, go to the Runtime menu and select “Change runtime type”, then make sure a GPU is selected (e.g., an Nvidia T4 GPU). The GPU runtime will automatically have the appropriate PyTorch library installed—compiled with GPU support—as well as the appropriate GPU drivers and related libraries (e.g., Nvidia’s CUDA and cuDNN libraries).<sup>3</sup> If you prefer to run the code on your own machine, you will need to ensure that you have all the drivers and libraries required. Please follow the instructions at <https://hml.info/install-p>.

PyTorch has excellent support for Nvidia GPUs, as well as several other hardware accelerators:

- Apple’s *Metal Performance Shaders* (MPS) to accelerate computations on Apple silicon such as the M1, M2, and later chips, as well as some Intel Macs with a compatible GPU.
- AMD Instinct accelerators and AMD Radeon GPUs, through the ROCm software stack, or via DirectML on Windows.
- Intel GPUs and CPUs on Linux and Windows via Intel’s oneAPI.
- Google TPUs via the `torch_xla` library.

Let’s check whether PyTorch can access an Nvidia GPU or Apple’s MPS, otherwise let’s fall back to the CPU:

```
if torch.cuda.is_available():
    device = "cuda"
elif torch.backends.mps.is_available():
    device = "mps"
else:
    device = "cpu"
```

---

<sup>3</sup> CUDA is Nvidia’s proprietary platform to run code on its CUDA-compatible GPUs, and cuDNN is a library built on CUDA to accelerate various deep neural network architectures.



Deep learning generally requires a *lot* of compute power, especially once we start diving into computer vision and natural language processing, in the following chapters. You will need a reasonably powerful machine, but most importantly you will need a hardware accelerator (or several). If you don't have one, you can try using Colab or Kaggle; they offer runtimes with free GPUs. Or consider using other cloud services. Otherwise, prepare to be very, very patient.

On a Colab GPU runtime, `device` will be equal to "cuda". Now let's create a tensor on that GPU. To do that, one option is to create the tensor on the CPU, then copy it to the GPU using the `to()` method:

```
>>> M = torch.tensor([[1., 2., 3.], [4., 5., 6.]])  
>>> M = M.to(device)
```



The `cpu()` and `cuda()` methods are short for `to("cpu")` and `to("cuda")`, respectively.

You can always tell which device a tensor lives on by looking at its `device` attribute:

```
>>> M.device  
device(type='cuda', index=0)  
---
```

Alternatively, we can create the tensor directly on the GPU using the `device` argument:

```
>>> M = torch.tensor([[1., 2., 3.], [4., 5., 6.]], device=device)
```



If you have multiple Nvidia GPUs, you can refer to the desired GPU by appending the GPU index: "cuda:0" (or just "cuda") for GPU #0, "cuda:1" for GPU #1, and so on.

Once the tensor is on the GPU, we can run operations on it normally, and they will all take place on the GPU:

```
>>> R = M @ M.T # run some operations on the GPU  
>>> R  
tensor([[14., 32.],  
       [32., 77.]], device='cuda:0')
```

Note that the result  $R$  also lives on the GPU. This means we can perform multiple operations on the GPU without having to transfer data back and forth between the CPU and the GPU. This is crucial in deep learning because data transfer between devices can often become a performance bottleneck.

How much does a GPU accelerate the computations? Well it depends on the GPU, of course: the more expensive ones are dozens of times faster than the cheap ones. But speed alone is not the only important factor: the data throughput is also crucial, as we just saw. If your model is compute heavy (e.g., a very deep neural net), the GPU's speed and amount of RAM will typically matter most, but if it is a shallower model, then pumping the training data into the GPU might become the bottleneck. Let's run a little test to compare the speed of a matrix multiplication running on the CPU versus the GPU:<sup>4</sup>

```
>>> M = torch.rand((1000, 1000)) # on the CPU
>>> %timeit M @ M.T
16.1 ms ± 2.17 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> M = torch.rand((1000, 1000), device="cuda") # on the GPU
>>> %timeit M @ M.T
549 µs ± 3.99 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Wow! The GPU gave us a  $29\times$  speed boost! And that's just using the free Nvidia T4 GPU on Colab; imagine the speedup we could get using a more powerful GPU. Now try playing around with the matrix size: you will notice that the speedup is much less impressive on smaller matrices (e.g., it's just  $2\times$  for  $100 \times 100$  matrices). That's because GPUs work by breaking large operations into smaller operations and running them in parallel across thousands of cores. If the task is small, it cannot be broken up into that many pieces, and the performance gain is therefore smaller. In fact, when running many tiny tasks, it can sometimes be faster to just run the operations on the CPU.

All right, now that we've seen what tensors are and how to use them on the CPU or the GPU, let's look at PyTorch's auto-differentiation feature.

## Autograd

PyTorch comes with an efficient implementation of reverse-mode auto-differentiation (introduced in [Chapter 9](#) and detailed in [Appendix A](#)), called *autograd*, which stands for automatic gradients. It is quite easy to use. For example, consider a simple function,  $f(x) = x^2$ . Differential calculus tells us that the derivative of this function is  $f'(x) = 2x$ . If we evaluate  $f(5)$  and  $f'(5)$ , we get 25 and 10, respectively. Let's see if PyTorch agrees:

---

<sup>4</sup> The `%timeit` magic command only works in Jupyter notebooks and Colab, as well as in the iPython shell; in a regular Python shell or program, you can use the `timeit.timeit()` function instead.

```

>>> x = torch.tensor(5.0, requires_grad=True)
>>> f = x ** 2
>>> f
tensor(25., grad_fn=<PowBackward0>)
>>> f.backward()
>>> x.grad
tensor(10.)

```

Great, we got the correct results: `f` is 25, and `x.grad` is 10! Note that the `backward()` function automatically computed the gradient  $f'(x)$  at the same point  $x = 5.0$ . Let's go through this code line by line:

- First, we created a tensor `x`, equal to 5.0, and we told PyTorch that it's a variable (not a constant) by specifying `requires_grad=True`. Knowing this, PyTorch will automatically keep track of all operations involving `x`: this is needed because PyTorch must capture the computation graph in order to run backprop on it and obtain the derivative of `f` with regard to `x`. In this computation graph, the tensor `x` is a *leaf node*.
- Then we compute `f = x ** 2`. The result is a tensor equal to 25.0, the square of 5.0. But wait, there's more to it: `f` also carries a `grad_fn` attribute which represents the operation that created this tensor (`**`, power, hence the name `PowBackward0`), and which tells PyTorch how to backpropagate the gradients through this particular operation. This `grad_fn` attribute is how PyTorch keeps track of the computation graph.
- Next, we call `f.backward()`: this backpropagates the gradients through the computation graph, starting with `f`, and all the way back to the leaf nodes (just `x` in this case).
- Lastly, we can just read the `x` tensor's `grad` attribute, which was computed during backprop: this gives us the derivative of `f` with regard to `x`. Ta-da!

PyTorch creates a new computation graph on the fly during each forward pass, as the operations are executed. This allows PyTorch to support very dynamic models containing loops and conditionals.



The way PyTorch accumulates gradients in each variable's `grad` attribute can be surprising at first, especially coming from TensorFlow or JAX. In these frameworks, computing the gradients of `f` with regard to `x` just returns the gradients, without affecting `x`. In PyTorch, if you call `backward()` on a tensor, it will accumulate the gradients in every variable that was used to compute it. So if you call `backward()` on two tensors `t1` and `t2` that both used the same variable `v`, then `v.grad` will be the sum of their gradients.

After computing the gradients, you generally want to perform a gradient descent step by subtracting a fraction of the gradients from the model variables (at least when training a neural network). In our simple example, running gradient descent will gradually push  $x$  toward 0, since that's the value that minimizes  $f(x) = x^2$ . To do a gradient descent step, you must temporarily disable gradient tracking since you don't want to track the gradient descent step itself in the computation graph (in fact, PyTorch would raise an exception if you tried to run an in-place operation on a tracked variable). This can be done by placing the gradient descent step inside a `torch.no_grad()` context, like this:

```
learning_rate = 0.1
with torch.no_grad():
    x -= learning_rate * x.grad # gradient descent step
```

The variable  $x$  gets decremented by  $0.1 * 10.0 = 1.0$ , down from 5.0 to 4.0.

Another way to avoid gradient computation is to use the variable's `detach()` method: this creates a new tensor detached from the computation graph, with `requires_grad=False`, but still pointing to the same data in memory. You can then update this detached tensor:

```
x_detached = x.detach()
x_detached -= learning_rate * x.grad
```

Since `x_detached` and  $x$  share the same memory, modifying `x_detached` also modifies  $x$ .

The `detach()` method can be handy when you need to run some computation on a tensor without affecting the gradients (e.g., for evaluation or logging), or when you need fine-grained control over which operations should contribute to gradient computation. Using `no_grad()` is generally preferred when performing inference or doing a gradient descent step, as it provides a convenient context-wide method to disable gradient tracking.

Lastly, before you repeat the whole process (forward pass + backward pass + gradient descent step), it's essential to zero out the gradients of every model parameter (you don't need a `no_grad()` context for this since the gradient tensor has `requires_grad=False`):

```
x.grad.zero_()
```



If you forget to zero out the gradients at each training iteration, the `backward()` method will just accumulate them, causing incorrect gradient descent updates. Since there won't be any explicit error, just low performance (and perhaps infinite or NaN values), this issue may be hard to debug.

Putting everything together, the whole training loop looks like this:

```
learning_rate = 0.1
x = torch.tensor(5.0, requires_grad=True)
for iteration in range(100):
    f = x ** 2 # forward pass
    f.backward() # backward pass
    with torch.no_grad():
        x -= learning_rate * x.grad # gradient descent step

    x.grad.zero_() # reset the gradients
```

If you want to use in-place operations to save memory and speed up your models a bit by avoiding unnecessary copy operations, you have to be careful: in-place operations don't always play nicely with autograd. Firstly, as we saw earlier, you cannot apply an in-place operation to a leaf node (i.e., a tensor with `requires_grad=True`), as PyTorch wouldn't know where to store the computation graph. For example `x.cos_()` or `x += 1` would cause a `RuntimeError`. Secondly, consider the following code, which computes  $z(t) = \exp(t) + 1$  at  $t = 2$  and then tries to compute the gradients:

```
t = torch.tensor(2.0, requires_grad=True)
z = t.exp() # this is an intermediate result
z += 1 # this is an in-place operation
z.backward() # ▲ RuntimeError!
```

Oh no! Although `z` is computed correctly, the last line causes a `RuntimeError`, complaining that "one of the variables needed for gradient computation has been modified by an in-place operation". Indeed, the intermediate result `z = t.exp()` was lost when we ran the in-place operation `z += 1`, so when the backward pass reached the exponential operation, the gradients could not be computed. A simple fix is to replace `z += 1` with `z = z + 1`. It looks similar, but it's no longer an in-place operation: a new tensor is created and assigned to the same variable, but the original tensor is unchanged and recorded in the computation graph of the final tensor.

Surprisingly, if you replace `exp()` with `cos()` in the previous code example, the gradients will be computed correctly: no error! Why is that? Well, the outcome depends on the way each operation is implemented:

- Some operations—such as `exp()`, `relu()`, `rsqrt()`, `sigmoid()`, `sqrt()`, `tan()`, and `tanh()`—save their outputs in the computation graph during the forward pass, then use these outputs to compute the gradients during the backward pass.<sup>5</sup>

---

<sup>5</sup> For example, since the derivative of  $\exp(x)$  is equal to  $\exp(x)$ , it makes a lot of sense to store the output of this operation in the computation graph during the forward pass, then use this output during the backward pass to get the gradients: no need to store additional data, and no need to recompute  $\exp(x)$ .

This means that you must not modify such an operation’s output in place, or you will get an error during the backward pass (as we just saw).

- Other operations—such as `abs()`, `cos()`, `log()`, `sin()`, `square()`, and `var()`—save their inputs instead of their output.<sup>6</sup> Such an operation doesn’t care if you modify its output in place, but you must not modify its inputs in place before the backward pass (e.g., to compute something else based on the same inputs).
- Some operations—such as `max()`, `min()`, `norm()`, `prod()`, `sgn()`, and `std()`—save both the inputs and the outputs, so you must not modify either of them in place before the backward pass.
- Lastly, a few operations—such as `ceil()`, `floor()`, `mean()`, `round()`, and `sum()`—save neither their inputs nor their outputs.<sup>7</sup> You can safely modify them in place.



Implement your models first without any in-place operations, then if you need to save some memory or speed up your model a bit, you can try converting some of the most costly operations to their in-place counterparts. Just make sure that your model still outputs the same result for a given input, and also make sure you don’t modify in place a tensor needed for backprop (you will get a `RuntimeError` in this case).

OK, let’s step back a bit. We’ve discussed all the fundamentals of PyTorch: how to create tensors and use them to perform all sorts of computations, how to accelerate the computations with a GPU, and how to use autograd to compute gradients for gradient descent. Great! Now let’s apply what we’ve learned so far by building and training a simple linear regression model with PyTorch.

## Implementing Linear Regression

We will start by implementing linear regression using tensors and autograd directly, then we will simplify the code using PyTorch’s high-level API, and also add GPU support.

---

<sup>6</sup> For example, the derivative of  $\text{abs}(x)$  is  $-1$  when  $x < 0$  and  $+1$  when  $x > 0$ . If this operation saved its output in the computation graph, the backward pass would be unable to know whether  $x$  was positive or negative (since  $\text{abs}(x)$  is always positive), so it wouldn’t be able to compute the gradients. This is why this operation must save its input instead.

<sup>7</sup> For example, the derivative of `floor(x)` is always zero (at least for noninteger inputs), so the `floor()` operation just saves the shape of the inputs during the forward pass, then during the backward pass it produces gradients of the same shape but full of zeros. For integer inputs, autograd also returns zeros instead of NaN.

## Linear Regression Using Tensors and Autograd

Let's tackle the same California housing dataset as in [Chapter 9](#). I will assume you have already downloaded it using `sklearn.datasets.fetch_california_housing()`, and you have split it into a training set (`X_train` and `y_train`), a validation set (`X_valid` and `y_valid`), and a test set (`X_test` and `y_test`), using `sklearn.model_selection.train_test_split()`. Next, let's convert it to tensors and normalize it. We could use a `StandardScaler` for this, like we did in [Chapter 9](#), but let's just use tensor operations instead, to get a bit of practice:

```
X_train = torch.FloatTensor(X_train)
X_valid = torch.FloatTensor(X_valid)
X_test = torch.FloatTensor(X_test)
means = X_train.mean(dim=0, keepdims=True)
stds = X_train.std(dim=0, keepdims=True)
X_train = (X_train - means) / stds
X_valid = (X_valid - means) / stds
X_test = (X_test - means) / stds
```

Let's also convert the targets to tensors. Since our predictions will be column vectors (i.e., matrices with a single column), we need to ensure that our targets are also column vectors.<sup>8</sup> Unfortunately, the NumPy arrays representing the targets are one-dimensional, so we need to reshape the tensors to column vectors by adding a second dimension of size 1:<sup>9</sup>

```
y_train = torch.FloatTensor(y_train).reshape(-1, 1)
y_valid = torch.FloatTensor(y_valid).reshape(-1, 1)
y_test = torch.FloatTensor(y_test).reshape(-1, 1)
```

Now that the data is ready, let's create the parameters of our linear regression model:

```
torch.manual_seed(42)
n_features = X_train.shape[1] # there are 8 input features
w = torch.randn((n_features, 1), requires_grad=True)
b = torch.tensor(0, requires_grad=True)
```

We now have a weights parameter `w` (a column vector with one weight per input dimension, in this case 8), and a bias parameter `b` (a single scalar). The weights are initialized randomly, while the bias is initialized to zero. We could have initialized the weights to zero as well in this case, but when we get to neural networks it will be important to initialize the weights randomly to break the symmetry between neurons (as explained in [Chapter 9](#)), so we might as well get into the habit now.

---

<sup>8</sup> Column vectors (shape  $[m, 1]$ ) and row vectors (shape  $[1, m]$ ) are often preferred over 1D vectors (shape  $[m]$ ) in machine learning, as they avoid ambiguity in some operations, such as matrix multiplication or broadcasting, and they make the code more consistent whether there's just one feature or more.

<sup>9</sup> Just like in NumPy, the `reshape()` method allows you to specify `-1` for one of the dimensions. This dimension's size is automatically calculated to ensure the new tensor has the same number of cells as the original.



We called `torch.manual_seed()` to ensure that the results are reproducible. However, PyTorch does not guarantee perfectly reproducible results across different releases, platforms, or devices, so if you do not run the code in this chapter with PyTorch 2.8.0 on a Colab runtime with an Nvidia T4 GPU, you may get different results. Moreover, since a GPU splits each operation into multiple chunks and runs them in parallel, the order in which these chunks finish may vary across runs, and this may slightly affect the result due to floating-point precision errors. These minor differences may compound during training, and lead to very different models. To avoid this, you can tell PyTorch to use only deterministic algorithms by calling `torch.use_deterministic_algorithms(True)` and setting `torch.backends.cudnn.benchmark = False`. However, deterministic algorithms are often slower than stochastic ones, and some operations don't have a deterministic version at all, so you will get an error if your code tries to use one.

Next, let's train our model, very much like we did in [Chapter 4](#), except we will use autodiff to compute the gradients rather than using a closed-form equation. For now we will use batch gradient descent (BGD), using the full training set at each training step:

```
learning_rate = 0.4
n_epochs = 20
for epoch in range(n_epochs):
    y_pred = X_train @ w + b
    loss = ((y_pred - y_train) ** 2).mean()
    loss.backward()
    with torch.no_grad():
        b -= learning_rate * b.grad
        w -= learning_rate * w.grad
        b.grad.zero_()
        w.grad.zero_()
    print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}")
```

Let's walk through this code:

- First we define the `learning_rate` hyperparameter. You can experiment with different values to find a value that converges fast and gives a precise result.
- Next, we run 20 epochs. We could implement early stopping to find the right moment to stop and avoid overfitting, like we did in [Chapter 4](#), but we will keep things simple for now.
- Next, we run the forward pass: we compute the predictions `y_pred`, and the mean squared error `loss`.
- Then we run `loss.backward()` to compute the gradients of the loss with regard to every model parameter. This is autograd in action.

- Next, we use the gradients `b.grad` and `w.grad` to perform a gradient descent step. Notice that we're running this code inside a `with torch.no_grad()` context, as discussed earlier.
- Once we've done the gradient descent step, we reset the gradients to zero (very important!).
- Lastly, we print the epoch number and the current loss at each epoch. The `item()` method extracts the value of a scalar tensor.

And that's it; if you run this code, you should see the training loss going down like this:

```
Epoch 1/20, Loss: 16.158456802368164
Epoch 2/20, Loss: 4.8793745040893555
Epoch 3/20, Loss: 2.255225419998169
[...]
Epoch 20/20, Loss: 0.5684100389480591
```

Congratulations, you just trained your first model using PyTorch! You can now use the model to make predictions for some new data `X_new` (which must be represented as a PyTorch tensor). For example, let's make predictions for the first three instances in the test set:

```
>>> X_new = X_test[:3] # pretend these are new instances
>>> with torch.no_grad():
...     y_pred = X_new @ w + b # use the trained parameters to make predictions
...
>>> y_pred
tensor([[0.8916],
       [1.6480],
       [2.6577]])
```



It's best to use a `with torch.no_grad()` context during inference: PyTorch will consume less RAM and run faster since it won't have to keep track of the computation graph.

Implementing linear regression using PyTorch's low-level API wasn't too hard, but using this approach for more complex models would get really messy and difficult. So PyTorch offers a higher-level API to simplify all this. Let's rewrite our model using this higher-level API.

## Linear Regression Using PyTorch's High-Level API

PyTorch provides an implementation of linear regression in the `torch.nn.Linear` class, so let's use it:

```
import torch.nn as nn # by convention, this module is usually imported this way

torch.manual_seed(42) # to get reproducible results
model = nn.Linear(in_features=n_features, out_features=1)
```

The `nn.Linear` class (short for `torch.nn.Linear`) is one of many *modules* provided by PyTorch. Each module is a subclass of the `nn.Module` class. To build a simple linear regression model, a single `nn.Linear` module is all you need. However, for most neural networks you will need to assemble many modules, as we will see later in this chapter, so you can think of modules as math LEGO® bricks. Many modules contain model parameters. For example, the `nn.Linear` module contains a `bias` vector (with one bias term per neuron), and a `weight` matrix (with one row per neuron and one column per input dimension, which is the transpose of the weight matrix we used earlier and in [Equation 9-2](#)). Since our model has a single neuron (because `out_features=1`), the `bias` vector contains a single bias term, and the `weight` matrix contains a single row. These parameters are accessible directly as attributes of the `nn.Linear` module:

```
>>> model.bias
Parameter containing:
tensor([0.3117], requires_grad=True)
>>> model.weight
Parameter containing:
tensor([[ 0.2703,  0.2935, -0.0828,  0.3248, -0.0775,  0.0713, -0.1721,  0.2076]], requires_grad=True)
```

Notice that both parameters were automatically initialized randomly (which is why we used `manual_seed()` to get reproducible results). These parameters are instances of the `torch.nn.Parameter` class, which is a subclass of the `tensor.Tensor` class: this means that you can use them exactly like normal tensors. A module's `parameters()` method returns an iterator over all of the module's attributes of type `Parameter`, as well as all the parameters of all its submodules, recursively (if it has any). It does *not* return regular tensors, even those with `requires_grad=True`. That's the main difference between a regular tensor and a `Parameter`:

```
>>> for param in model.parameters():
...     [...] # do something with each parameter
```

There's also a `named_parameters()` method that returns an iterator over pairs of parameter names and values.

A module can be called just like a regular function. For example, let's make some predictions for the first two instances in the training set (since the model is not trained yet, its parameters are random and the predictions are terrible):

```
>>> model(X_train[:2])
tensor([-0.4718,
       0.1131], grad_fn=<AddmmBackward0>)
```

When we use a module as a function, PyTorch internally calls the module's `forward()` method. In the case of the `nn.Linear` module, the `forward()` method computes  $X @ \text{self.weight.T} + \text{self.bias}$  (where  $X$  is the input). That's just what we need for linear regression!

Notice that the result contains the `grad_fn` attribute, showing that autograd did its job and tracked the computation graph while the model was making its predictions.



If you pass a custom function to a module's `register_forward_hook()` method, it will be called automatically every time the module itself is called. This is particularly handy for logging or debugging. To remove a hook, just call the `remove()` method on the object returned by `register_forward_hook()`. Note that hooks only work if you call the model like a function, not if you call its `forward()` method directly (which is why you should never do that). You can also register functions to run during the backward pass using `register_backward_hook()`.

Now that we have our model, we need to create an optimizer to update the model parameters, and we must also choose a loss function:

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
mse = nn.MSELoss()
```

PyTorch provides a few different optimizers (we will discuss them in the next chapter). Here we're using the simple stochastic gradient descent (SGD) optimizer, which can be used for SGD, mini-batch GD, or batch gradient descent. To initialize it, we must give it the model parameters and the learning rate.

For the loss function, we create an instance of the `nn.MSELoss` class: this is also a module, so we can use it like a function, giving it the predictions and the targets, and it will compute the MSE. The `nn` module contains many other loss functions and other neural net tools, as we will see. Next, let's write a small function to train our model:

```
def train_bgd(model, optimizer, criterion, X_train, y_train, n_epochs):
    for epoch in range(n_epochs):
        y_pred = model(X_train)
        loss = criterion(y_pred, y_train)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}")
```

Compare this training loop with our earlier training loop: it's very similar, but we're now using higher-level constructs rather than working directly with tensors and autograd. Here are a few things to note:

- In PyTorch, the loss function object is commonly referred to as the *criterion*, to distinguish it from the loss value itself (which is computed at each training iteration using the criterion). In this example, it's the `MSELoss` instance.
- The `optimizer.step()` line corresponds to the two lines that updated `b` and `w` in our earlier code.
- And of course the `optimizer.zero_grad()` line corresponds to the two lines that zeroed out `b.grad` and `w.grad`. Notice that we don't need to use `with torch.no_grad()` here since this is done automatically by the optimizer, inside the `step()` and `zero_grad()` functions.



Most people prefer to call `zero_grad()` *before* calling `loss.backward()`, rather than after: this might be a bit safer in case the gradients are nonzero when calling the function, but in general it makes no difference since gradients are automatically initialized to `None`.

Now let's call this function to train our model!

```
>>> train_bgd(model, optimizer, mse, X_train, y_train, n_epochs)
Epoch 1/20, Loss: 4.3378496170043945
Epoch 2/20, Loss: 0.780293345451355
[...]
Epoch 20/20, Loss: 0.5374288558959961
```

All good; the model is trained, and you can now use it to make predictions by simply calling it like a function (preferably inside a `no_grad()` context, as we saw earlier):

```
>>> X_new = X_test[:3] # pretend these are new instances
>>> with torch.no_grad():
...     y_pred = model(X_new) # use the trained model to make predictions
...
>>> y_pred
tensor([[0.8061],
        [1.7116],
        [2.6973]])
```

These predictions are similar to the ones our previous model made, but not exactly the same. That's because the `nn.Linear` module initializes the parameters slightly differently: it uses a uniform random distribution from  $-\frac{\sqrt{2}}{4}$  to  $+\frac{\sqrt{2}}{4}$  for both the weights and the bias term (we will discuss initialization methods in [Chapter 11](#)).

Now that you are familiar with PyTorch's high-level API, you are ready to go beyond linear regression and build a multilayer perceptron (introduced in [Chapter 9](#)).

# Implementing a Regression MLP

PyTorch provides a helpful `nn.Sequential` module that chains multiple modules: when you call this module with some inputs, it feeds these inputs to the first module, then feeds the output of the first module to the second module, and so on. Most neural networks contain stacks of modules, and in fact many neural networks are just one big stack of modules: this makes the `nn.Sequential` module one of the most useful modules in PyTorch. The MLP we want to build is just that: a simple stack of modules—two hidden layers and one output layer. So let's build it using the `nn.Sequential` module:

```
torch.manual_seed(42)
model = nn.Sequential(
    nn.Linear(n_features, 50),
    nn.ReLU(),
    nn.Linear(50, 40),
    nn.ReLU(),
    nn.Linear(40, 1)
)
```

Let's go through each layer:

- The first layer must have the right number of inputs for our data: `n_features` (equal to 8 in our case). However, it can have any number of outputs: let's pick 50 (that's a hyperparameter we can tune).
- Next we have an `nn.ReLU` module, which implements the ReLU activation function for the first hidden layer. This module does not contain any model parameters, and it acts itemwise so the shape of its output is equal to the shape of its input.
- The second hidden layer must have the same number of inputs as the output of the previous layer: in this case, 50. However, it can have any number of outputs. It's common to use the same number of output dimensions in all hidden layers, but in this example I used 40 to make it clear that the output of one layer must match the input of the next layer.
- Then again, an `nn.ReLU` module to implement the second hidden layer's activation function.
- Finally, the output layer must have 40 inputs, but this time its number of outputs is not free: it must match the targets' dimensionality. Since our targets have a single dimension, we must have just one output dimension in the output layer.

Now let's train the model just like we did before:

```
>>> learning_rate = 0.1
>>> optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
>>> mse = nn.MSELoss()
```

```
>>> train_bgd(model, optimizer, mse, X_train, y_train, n_epochs)
Epoch 1/20, Loss: 5.045480251312256
Epoch 2/20, Loss: 2.0523128509521484
[...]
Epoch 20/20, Loss: 0.565444827079773
```

That's it, you can tell your friends you trained your first neural network with PyTorch! However, we are still using batch gradient descent, computing the gradients over the entire training set at each iteration. This works with small datasets, but if we want to be able to scale up to large datasets and large models, we need to switch to mini-batch gradient descent.

## Implementing Mini-Batch Gradient Descent Using DataLoaders

To help implement mini-batch GD, PyTorch provides a class named `DataLoader` in the `torch.utils.data` module. It can efficiently load batches of data of the desired size, and shuffle the data at each epoch if we want it to. The `DataLoader` expects the dataset to be represented as an object with at least two methods: `__len__(self)` to get the number of samples in the dataset, and `__getitem__(self, index)` to load the sample at the given index (including the target).

In our case, the training set is available in the `X_train` and `y_train` tensors, so we first need to wrap these tensors in a dataset object with the required API. To help with this, PyTorch provides a `TensorDataset` class. So let's build a `TensorDataset` to wrap our training set, and a `DataLoader` to pull batches from this dataset. During training, we want the dataset to be shuffled, so we specify `shuffle=True`:

```
from torch.utils.data import TensorDataset, DataLoader

train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

Now that we have a larger model and we have the tools to train it one batch at a time, it's a good time to start using hardware acceleration. It's really quite simple: we just need to move the model to the GPU, which will move all of its parameters to the GPU RAM, and then at the start of each iteration during training we must copy each batch to the GPU. To move the model, we can just use its `to()` method, just like we did with tensors:

```
torch.manual_seed(42)
model = nn.Sequential([...]) # create the model just like earlier
model = model.to(device)
```

We can also create the loss function and optimizer, as earlier (but using a lower learning rate, such as 0.02).



Some optimizers have some internal state, as we will see in [Chapter 11](#). The optimizer will usually allocate its state on the same device as the model parameters, so it's important to create the optimizer *after* you have moved the model to the GPU.

Now let's create a `train()` function to implement mini-batch GD:

```
def train(model, optimizer, criterion, train_loader, n_epochs):
    model.train()
    for epoch in range(n_epochs):
        total_loss = 0.
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            loss = criterion(y_pred, y_batch)
            total_loss += loss.item()
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        mean_loss = total_loss / len(train_loader)
        print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {mean_loss:.4f}")
```

At every epoch, the function iterates through the whole training set, one batch at a time, and processes each batch just like earlier. But what about the very first line: `model.train()`? Well, this switches the model and all of its submodules to *training mode*. For now, this makes no difference at all, but it will be important in [Chapter 11](#) when we start using layers that behave differently during training and evaluation (e.g., `nn.Dropout` or `nn.BatchNorm1d`). Whenever you want to use the model outside of training (e.g., for evaluation, or to make predictions on new instances), you must first switch the model to *evaluation mode* by running `model.eval()`. Note that `model.training` holds a boolean that indicates the current mode.



PyTorch itself does not provide a training loop implementation; you have to build it yourself. As we just saw, it's not that long, and many people enjoy the freedom, clarity, and control this provides. However, if you would prefer to use a well-tested, off-the-shelf training loop with all the bells and whistles you need (such as multi-GPU support), then you can use a library such as PyTorch Lightning, FastAI, Catalyst, or Keras. These libraries are built on top of PyTorch and include a training loop and many other features (Keras supports PyTorch since version 3, and also supports TensorFlow and JAX). Check them out!

Now let's call this `train()` function to train our model on the GPU:

```
>>> train(model, optimizer, mse, train_loader, n_epochs)
Epoch 1/20, Loss: 0.6958
Epoch 2/20, Loss: 0.4480
[...]
Epoch 20/20, Loss: 0.3227
```

It worked great: we actually reached a much lower loss in the same number of epochs! However, you probably noticed that each epoch was much slower. There are two easy tweaks you can make to considerably speed up training:

- If you are using a CUDA device, you should generally set `pin_memory=True` when creating the data loader: this will allocate the data in *page-locked memory* which guarantees a fixed physical memory location in the CPU RAM, and therefore allows direct memory access (DMA) transfers to the GPU, eliminating an extra copy operation that would otherwise be needed. While this could use more CPU RAM since the memory cannot be swapped out to disk, it typically results in significantly faster data transfers and thus faster training. When transferring a tensor to the GPU using its `to()` method, you may also set `non_blocking=True` to avoid blocking the CPU during the data transfer (this only works if `pin_memory=True`).
- The current training loop waits until a batch has been fully processed before it loads the next batch. You can often speed up training by pre-fetching the next batches on the CPU while the GPU is still working on the current batch. For this, set the data loader's `num_workers` argument to the number of processes you want to use for data loading and preprocessing. The optimal number depends on your platform, hardware, and workload, so you should experiment with different values. You can also tweak the number of batches that each worker pre-fetches by setting the data loader's `prefetch_factor` argument. Note that the overhead of spawning and synchronizing workers can often slow down training rather than speed it up (especially on Windows). In this case, you can try setting `persistent_workers=True` to reuse the same workers across epochs.

OK, time to step back a bit: you know the PyTorch fundamentals (tensors and autograd), you can build neural nets using PyTorch's high-level API, and train them using mini-batch gradient descent, with the help of an optimizer, a criterion, and a data loader. The next step is to learn how to evaluate your model.

## Model Evaluation

Let's write a function to evaluate the model. It takes the model and a `DataLoader` for the dataset that we want to evaluate the model on, as well as a function to compute the metric for a given batch, and lastly a function to aggregate the batch metrics (by default, it just computes the mean):

```

def evaluate(model, data_loader, metric_fn, aggregate_fn=torch.mean):
    model.eval()
    metrics = []
    with torch.no_grad():
        for X_batch, y_batch in data_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            metric = metric_fn(y_pred, y_batch)
            metrics.append(metric)
    return aggregate_fn(torch.stack(metrics))

```

Now let's build a `TensorDataset` and a `DataLoader` for our validation set, and pass it to our `evaluate()` function to compute the validation MSE:

```

>>> valid_dataset = TensorDataset(X_valid, y_valid)
>>> valid_loader = DataLoader(valid_dataset, batch_size=32)
>>> valid_mse = evaluate(model, valid_loader, mse)
>>> valid_mse
tensor(0.4080, device='cuda:0')

```

It works fine. But now suppose we want to use the RMSE instead of the MSE (as we saw in [Chapter 2](#), it can be easier to interpret). PyTorch does not have a built-in function for that, but it's easy enough to write:

```

>>> def rmse(y_pred, y_true):
...     return ((y_pred - y_true) ** 2).mean().sqrt()
...
>>> evaluate(model, valid_loader, rmse)
tensor(0.5668, device='cuda:0')

```

But wait a second! The RMSE should be equal to the square root of the MSE; however, when we compute the square root of the MSE that we found earlier, we get a different result:

```

>>> valid_mse.sqrt()
tensor(0.6388, device='cuda:0')

```

The reason is that instead of calculating the RMSE over the whole validation set, we computed it over each batch and then computed the mean of all these batch RMSEs. That's not mathematically equivalent to computing the RMSE over the whole validation set. To solve this, we can use the MSE as our `metric_fn`, and use the `aggregate_fn` to compute the square root of the mean MSE:<sup>10</sup>

```

>>> evaluate(model, valid_loader, mse,
...           aggregate_fn=lambda metrics: torch.sqrt(torch.mean(metrics)))
...
tensor(0.6388, device='cuda:0')

```

<sup>10</sup> The mean of the batch MSEs is equal to the overall MSE since all batches have the same size. Well, except the last batch, which is often smaller, but this makes very little difference.

That's much better!

Rather than implement metrics yourself, you may prefer to use the TorchMetrics library (made by the same team as PyTorch Lightning), which provides many well-tested *streaming metrics*. A streaming metric is an object that keeps track of a given metric, and can be updated one batch at a time. The TorchMetrics library is not preinstalled on Colab, so we have to run `%pip install torchmetrics`, then we can implement the `evaluate_tm()` function, like this:

```
import torchmetrics

def evaluate_tm(model, data_loader, metric):
    model.eval()
    metric.reset() # reset the metric at the beginning
    with torch.no_grad():
        for X_batch, y_batch in data_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            metric.update(y_pred, y_batch) # update it at each iteration
    return metric.compute() # compute the final result at the end
```

Then we can create an RMSE streaming metric, move it to the GPU, and use it to evaluate the validation set:

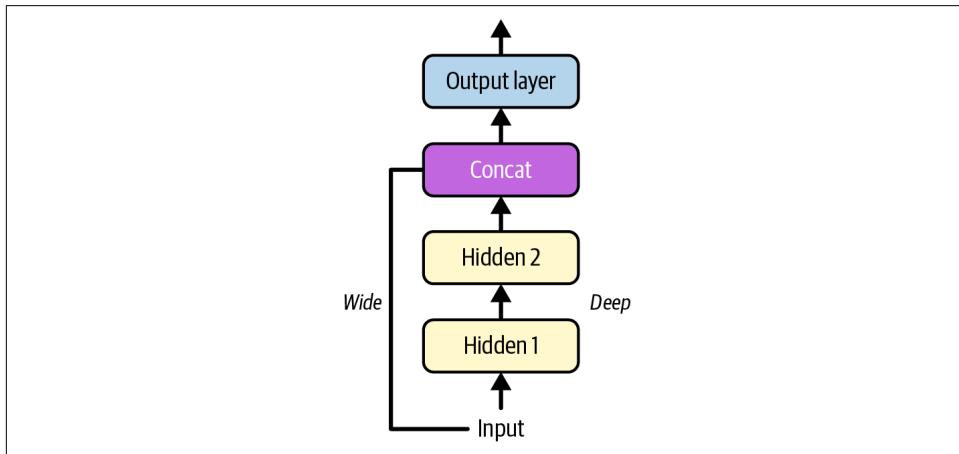
```
>>> rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
>>> evaluate_tm(model, valid_loader, rmse)
tensor(0.6388, device='cuda:0')
```

Sure enough, we get the correct result! Now try updating the `train()` function to evaluate your model's performance during training, both on the training set (during each epoch) and on the validation set (at the end of each epoch). As always, if the performance on the training set is much better than on the validation set, your model is probably overfitting the training set, or there is a bug, such as a data mismatch between the training set and the validation set. This is easier to detect if you plot and analyze the learning curves, much like we did in [Chapter 4](#). For this you can use Matplotlib, or a visualization tool such as TensorBoard (see the notebook for an example).

Now you know how to build, train, and evaluate a regression MLP using PyTorch, and how to use the trained model to make predictions. Great! But so far we have only looked at simple sequential models, composed of a sequence of linear layers and ReLU activation functions. How would you build a more complex, nonsequential model? For this, we will need to build custom modules.

# Building Nonsequential Models Using Custom Modules

One example of a nonsequential neural network is a *Wide & Deep* neural network. This neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al.<sup>11</sup> It connects all or part of the inputs directly to the output layer, as shown in [Figure 10-1](#). This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path). The short path can also be used to provide manually engineered features to the neural network. In contrast, a regular MLP forces all the data to flow through the full stack of layers; thus, simple patterns in the data may end up being distorted by this sequence of transformations.



*Figure 10-1. Wide & Deep neural network*

Let's build such a neural network to tackle the California housing dataset. Because this wide and deep architecture is nonsequential, we have to create a custom module. It's easier than it sounds: just create a class derived from `torch.nn.Module`, then create all the layers you need in the constructor (after calling the base class's `__init__()` method), and define how these layers should be used by the module in the `forward()` method:

```
class WideAndDeep(nn.Module):
    def __init__(self, n_features):
        super().__init__()
        self.deep_stack = nn.Sequential(
            nn.Linear(n_features, 50), nn.ReLU(),
            nn.Linear(50, 40), nn.ReLU(),
```

<sup>11</sup> Heng-Tze Cheng et al., “[Wide & Deep Learning for Recommender Systems](#)”, *Proceedings of the First Workshop on Deep Learning for Recommender Systems* (2016): 7–10.

```

        )
        self.output_layer = nn.Linear(40 + n_features, 1)

    def forward(self, X):
        deep_output = self.deep_stack(X)
        wide_and_deep = torch.concat([X, deep_output], dim=1)
        return self.output_layer(wide_and_deep)

```

Notice that we can use any kind of module inside our custom module: in this example, we use an `nn.Sequential` module to build the “deep” part of our model (it’s actually not that deep; this is just a toy example). It’s the same MLP as earlier, except we separated the output layer because we need to feed it the concatenation of the model’s inputs and the deep part’s outputs. For this same reason, the output layer now has `40 + n_features` inputs instead of just 40.

In the `forward()` method, we just feed the input `X` to the deep stack, concatenate the input and the deep stack’s output, and feed the result to the output layer.



Modules have a `children()` method that returns an iterator over the module’s submodules (nonrecursively). There’s also a `named_children()` method. If your model has a variable number of submodules, you should store them in an `nn.ModuleList` or an `nn.ModuleDict`, which are returned by the `children()` and `named_children()` methods (as opposed to regular Python lists and dicts). Similarly, if your model has a variable number of parameters, you should store them in an `nn.ParameterList` or an `nn.ParameterDict` to ensure they are returned by the `parameters()` and `named_parameters()` methods.

Now we can create an instance of our custom module, move it to the GPU, train it, evaluate it, and use it exactly like our previous models:

```

torch.manual_seed(42)
model = WideAndDeep(n_features).to(device)
learning_rate = 0.002 # the model changed, so did the optimal learning rate
[...] # train, evaluate, and use the model, exactly like earlier

```

But what if you want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path, as illustrated in [Figure 10-2](#)? In this case, one approach is to split the inputs inside the `forward()` method, for example:

```

class WideAndDeepV2(nn.Module):
    [...] # same constructor as earlier, except with adjusted input sizes

    def forward(self, X):
        X_wide = X[:, :5]
        X_deep = X[:, 2:]

```

```

    deep_output = self.deep_stack(X_deep)
    wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
    return self.output_layer(wide_and_deep)

```

This works fine; however, in many cases it's preferable to just let the model take two separate tensors as input. Let's see why and how.

## Building Models with Multiple Inputs

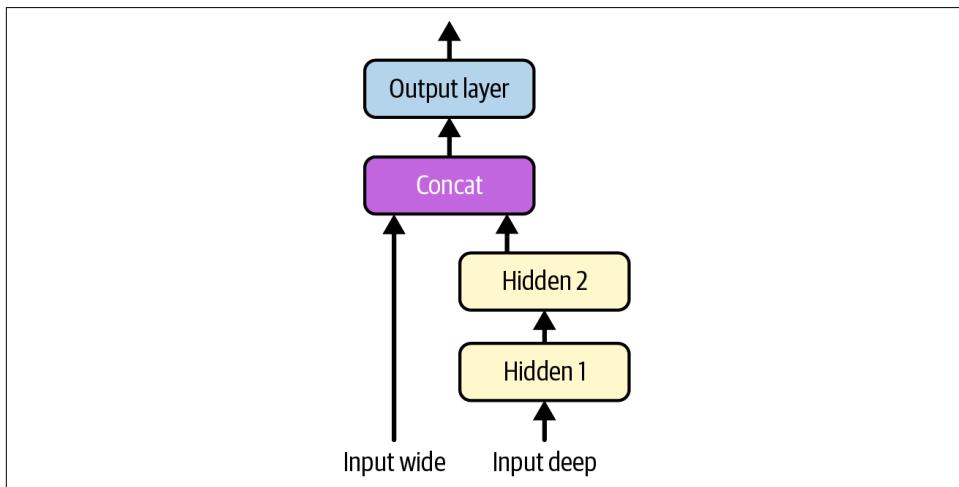
Some models require multiple inputs that cannot easily be combined into a single tensor. For example, the inputs may have a different number of dimensions (e.g., when you want to feed both images and text to the neural network). To make our Wide & Deep model take two separate inputs, as shown in [Figure 10-2](#), we must start by changing the model's `forward()` method:

```

class WideAndDeepV3(nn.Module):
    [...] # same as WideAndDeepV2

    def forward(self, X_wide, X_deep):
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        return self.output_layer(wide_and_deep)

```



*Figure 10-2. Handling multiple inputs*

Next, we need to create datasets that return the wide and deep inputs separately:

```

train_data_wd = TensorDataset(X_train[:, :5], X_train[:, 2:], y_train)
train_loader_wd = DataLoader(train_data_wd, batch_size=32, shuffle=True)
[...] # same for the validation set and test set

```

Since the data loaders now return three tensors instead of two at each iteration, we need to update the main loop in the evaluation and training functions:

```

for X_batch_wide, X_batch_deep, y_batch in data_loader:
    X_batch_wide = X_batch_wide.to(device)
    X_batch_deep = X_batch_deep.to(device)
    y_batch = y_batch.to(device)
    y_pred = model(X_batch_wide, X_batch_deep)
    [...] # the rest of the function is unchanged

```

Alternatively, since the order of the inputs matches the order of the `forward()` method's arguments, we can use Python's `*` operator to unpack all the inputs returned by the `data_loader` and pass them to the model. The advantage of this implementation is that it will work with models that take any number of inputs, not just two, as long as the order is correct:

```

for *X_batch_inputs, y_batch in data_loader:
    X_batch_inputs = [X.to(device) for X in X_batch_inputs]
    y_batch = y_batch.to(device)
    y_pred = model(*X_batch_inputs)
    [...]

```

When your model has many inputs, it's easy to make a mistake and mix up the order of the inputs, which can lead to hard-to-debug issues. To avoid this, it can be a good idea to name each input. For this, you can define a custom dataset that returns a dictionary from input names to input values, like this:

```

class WideAndDeepDataset(torch.utils.data.Dataset):
    def __init__(self, X_wide, X_deep, y):
        self.X_wide = X_wide
        self.X_deep = X_deep
        self.y = y

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        input_dict = {"X_wide": self.X_wide[idx], "X_deep": self.X_deep[idx]}
        return input_dict, self.y[idx]

```

Then create the datasets and data loaders:

```

train_data_named = WideAndDeepDataset(
    X_wide=X_train[:, :5], X_deep=X_train[:, 2:], y=y_train)
train_loader_named = DataLoader(train_data_named, batch_size=32, shuffle=True)
[...] # same for the validation set and test set

```

Once again, we also need to update the main loop in the evaluation and training functions:

```

for inputs, y_batch in data_loader:
    inputs = {name: X.to(device) for name, X in inputs.items()}
    y_batch = y_batch.to(device)
    y_pred = model(X_wide=inputs["X_wide"], X_deep=inputs["X_deep"])
    [...] # the rest of the function is unchanged

```

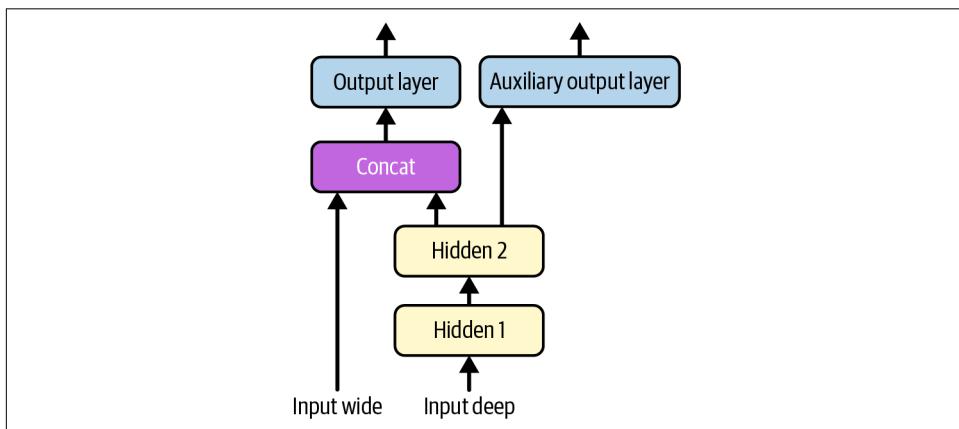
Alternatively, since all the input names match the `forward()` method's argument names, we can use Python's `**` operator to unpack all the tensors in the `inputs` dictionary and pass them as named arguments to the model: `y_pred = model(**inputs)`.

Now that you know how to build sequential and nonsequential models with one or more inputs, let's look at models with multiple outputs.

## Building Models with Multiple Outputs

There are many use cases where you may need a neural net with multiple outputs:

- The task may demand it. For instance, you may want to locate and classify the main object in a picture. This is both a regression task and a classification task.
- Similarly, you may have multiple independent tasks based on the same data. Sure, you could train one neural network per task, but in many cases you will get better results on all tasks by training a single neural network with one output per task. This is because the neural network can learn features in the data that are useful across tasks. For example, you could perform *multitask classification* on pictures of faces, using one output to classify the person's facial expression (smiling, surprised, etc.) and another output to identify whether they are wearing glasses or not.
- Another use case is regularization (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize). For example, you may want to add an auxiliary output in a neural network architecture (see [Figure 10-3](#)) to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.



*Figure 10-3. Handling multiple outputs, in this example to add an auxiliary output for regularization*

Let's add an auxiliary output to our Wide & Deep model to ensure the deep part can make good predictions on its own. Since the deep stack's output dimension is 40, and the targets have a single dimension, we must add an `nn.Linear` layer for the auxiliary output to go from 40 dimensions down to 1. We also need to make the `forward()` method compute the auxiliary output, and return both the main output and the auxiliary output:

```
class WideAndDeepV4(nn.Module):
    def __init__(self, n_features):
        [...] # same as earlier
        self.aux_output_layer = nn.Linear(40, 1)

    def forward(self, X_wide, X_deep):
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        main_output = self.output_layer(wide_and_deep)
        aux_output = self.aux_output_layer(deep_output)
        return main_output, aux_output
```

Next, we need to update the main loop in the training function:

```
for inputs, y_batch in train_loader:
    y_pred, y_pred_aux = model(**inputs)
    main_loss = criterion(y_pred, y_batch)
    aux_loss = criterion(y_pred_aux, y_batch)
    loss = 0.8 * main_loss + 0.2 * aux_loss
    [...] # the rest is unchanged
```

Notice that the model now returns both the main predictions `y_pred` and the auxiliary predictions `y_pred_aux`. In this example, we can use the same targets and the same loss function to compute the main output's loss and the auxiliary output's loss. In other cases, you may have different targets and loss functions for each output, in which case you would need to create a custom dataset to return all the necessary targets. Once we have a loss for each output, we must combine them into a single loss that will be minimized by gradient descent. In general, this final loss is just a weighted sum of all the output losses. In this example, we use a higher weight for the main loss (0.8), because that's what we care about the most, and a lower weight for the auxiliary loss (0.2). This ratio is a regularization hyperparameter that you can tune.

We also need to update the main loop in the evaluation function. However, in this case we can just ignore the auxiliary output, since we only really care about the main output—the auxiliary output is just there for regularization during training:

```
for inputs, y_batch in data_loader:
    y_pred, _ = model(**inputs)
    metric.update(y_pred, y_batch)
    [...] # the rest is unchanged
```

Voilà! You can now build and train all sorts of neural net architectures, combining predefined modules and custom modules in any way you please, and with any

number of inputs and outputs. The flexibility of neural networks is one of their main qualities. But so far we have only tackled a regression task, so let's now turn to classification.

## Building an Image Classifier with PyTorch

As in [Chapter 9](#), we will tackle the Fashion MNIST dataset, so the first thing we need to do is to download the dataset. We could use the `fetch_openml()` function like we did in [Chapter 9](#), but we will show another method instead, using the TorchVision library.

### Using TorchVision to Load the Dataset

The TorchVision library is an important part of the PyTorch ecosystem: it provides many tools for computer vision, including utility functions to download common datasets, such as MNIST or Fashion MNIST, as well as pretrained models for various computer vision tasks (see [Chapter 12](#)), functions to transform images (e.g., crop, rotate, resize, etc.), and more. It is preinstalled on Colab, so let's go ahead and use it to load Fashion MNIST. It is already split into a training set (60,000 images) and a test set (10,000 images), but we'll hold out the last 5,000 images from the training set for validation, using PyTorch's `random_split()` function:

```
import torchvision
import torchvision.transforms.v2 as T

toTensor = T.Compose([T.ToImage(), T.ToDtype(torch.float32, scale=True)])

train_and_valid_data = torchvision.datasets.FashionMNIST(
    root="datasets", train=True, download=True, transform=toTensor)
test_data = torchvision.datasets.FashionMNIST(
    root="datasets", train=False, download=True, transform=toTensor)

torch.manual_seed(42)
train_data, valid_data = torch.utils.data.random_split(
    train_and_valid_data, [55_000, 5_000])
```

After the imports and before loading the datasets, we create a `toTensor` object. What's that about? Well, by default, the `FashionMNIST` class loads images as PIL (Python Image Library) images, with integer pixel values ranging from 0 to 255. But we need PyTorch float tensors instead, with scaled pixel values. Luckily, TorchVision datasets accept a `transform` argument which lets you pass a preprocessing function that will get executed on the fly whenever the data is accessed (there's also a `target_transform` argument if you need to preprocess the targets). TorchVision provides many transform objects that you can use for this (most of these transforms are PyTorch modules).

In this code, we create a `Compose` transform to chain two transforms: a `ToImage` transform followed by a `ToDtype` transform. `ToImage` converts various formats—including PIL images, NumPy arrays, and tensors—to TorchVision’s `Image` class, which is a subclass of `Tensor`. The `ToDtype` transform converts the data type, in this case to 32-bit floats. We also set its `scale` argument to `True` to ensure the values get scaled between 0.0 and 1.0.<sup>12</sup>



Version 1 of TorchVision’s transforms API is still available for backward compatibility and can be imported using `import torchvision.transforms`. However, you should use version 2 (`torchvision.transforms.v2`) instead, since it’s faster and has more features.

Next, we load the dataset: first the training and validation data, then the test data. The `root` argument is the path to the directory where TorchVision will create a subdirectory for the Fashion MNIST dataset. The `train` argument indicates whether you want to load the training set (`True` by default) or the test set. The `download` argument indicates whether to download the dataset if it cannot be found locally (`False` by default). And we also set `transform=toTensor` to use our custom preprocessing pipeline.

As usual, we must create data loaders:

```
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
valid_loader = DataLoader(valid_data, batch_size=32)
test_loader = DataLoader(test_data, batch_size=32)
```

Now let’s look at the first image in the training set:

```
>>> X_sample, y_sample = train_data[0]
>>> X_sample.shape
tensor([1, 28, 28])
>>> X_sample.dtype
tensor.float32
```

In [Chapter 9](#), each image was represented by a 1D array containing 784 pixel intensities, but now each image tensor has 3 dimensions, and its shape is: [1, 28, 28]. The first dimension is the *channel* dimension. For grayscale images, there is a single channel (color images usually have three channels, as we will see in [Chapter 12](#)). The other two dimensions are the height and width dimensions. For example, `X_sample[0, 2, 4]` represents the pixel located in channel 0, row 2, column 4. In Fashion MNIST, a larger value means a darker pixel.

---

<sup>12</sup> TorchVision includes a `ToTensor` transform which does all this, but it’s deprecated so it’s recommended to use this pipeline instead.



PyTorch expects the channel dimension to be first, while many other libraries, such as Matplotlib, PIL, TensorFlow, OpenCV, or Scikit-Image, expect it to be last. Always make sure to move the channel dimension to the right place, depending on the library you are using. `ToImage` already took care of moving the channel dimension to the first position, otherwise we could have used the `torch.permute()` function.

As for the targets, they are integers from 0 to 9, and we can interpret them using the same `class_names` array as in [Chapter 9](#). In fact, many datasets—including `FashionMNIST`—have a `classes` attribute containing the list of class names. For example, here's how we can tell that the sample image represents an ankle boot:

```
>>> train_and_valid_data.classes[y_sample]
'Ankle boot'
```

## Building the Classifier

Let's build a custom module for a classification MLP with two hidden layers:

```
class ImageClassifier(nn.Module):
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_classes):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Flatten(),
            nn.Linear(n_inputs, n_hidden1),
            nn.ReLU(),
            nn.Linear(n_hidden1, n_hidden2),
            nn.ReLU(),
            nn.Linear(n_hidden2, n_classes)
        )

    def forward(self, X):
        return self.mlp(X)

torch.manual_seed(42)
model = ImageClassifier(n_inputs=28 * 28, n_hidden1=300, n_hidden2=100,
                       n_classes=10)
xentropy = nn.CrossEntropyLoss()
```

There are a few things to note in this code:

- First, the model is composed of a single sequence of layers, which is why we used the `nn.Sequential` module. We did not have to create a custom module; we could have written `model = nn.Sequential(...)` instead, but it's generally preferable to wrap your models in custom modules, as it makes your code easier to deploy and reuse, and it's also easier to tune the hyperparameters.

- The model starts with an `nn.Flatten` layer: this layer does not have any parameters, it just reshapes each input sample to a single dimension, which is needed for the `nn.Linear` layers. For example, a batch of 32 Fashion MNIST images has a shape of [32, 1, 28, 28], but after going through the `nn.Flatten` layer, it ends up with a shape of [32, 784] (since  $28 \times 28 = 784$ ).
- The first hidden layer must have the correct number of inputs ( $28 \times 28 = 784$ ), and the output layer must have the correct number of outputs (10, one per class).
- We use a ReLU activation function after each hidden layer, and no activation function at all after the output layer.
- Since this is a multiclass classification task, we use `nn.CrossEntropyLoss`. It accepts either class indices as targets (as in this example), or class probabilities (such as one-hot vectors).



Shape errors are quite common, especially when getting started, so you should familiarize yourself with the error messages: try removing the `nn.Flatten` module, or try messing with the shape of the inputs and/or labels, and see the errors you get.

But wait! Didn't we say in [Chapter 9](#) that we should use the softmax activation function on the output layer for multiclass classification tasks? Well it turns out that PyTorch's `nn.CrossEntropyLoss` computes the cross-entropy loss directly from the logits (i.e., the class scores, introduced in [Chapter 4](#)), rather than from the class probabilities. This bypasses some costly computations during training (e.g., logarithms and exponentials that cancel out), saving both compute and RAM. It's also more numerically stable. However, the downside is that the model must output logits, which means that we will have to call the softmax function manually on the logits whenever we want class probabilities, as we will see shortly.

## Other Classification Losses

For multiclass classification, another option is to add the `nn.LogSoftmax` activation function to the output layer, and then use the `nn.NLLLoss` (negative log-likelihood loss). The model then outputs log probabilities (rather than logits), and the loss computes the cross-entropy based on these log probabilities. Whenever you need actual estimated probabilities, just pass the log probabilities through the exponential function. This approach is a bit slower than using `nn.CrossEntropyLoss`, so it's not used as often, but it can sometimes be useful if you want your model to output log probabilities, or when you wish to tweak the probability distribution before computing the loss.

For binary classification tasks, you must use a single output neuron in the output layer, and use the `nn.BCEWithLogitsLoss` (BCE stands for binary cross-entropy). The model outputs logits, so you must apply the sigmoid function to get estimated probabilities (for the positive class). Alternatively, you can add the `nn.Sigmoid` activation function to the output layer, and use the `nn.BCELoss`: the model will then output estimated probabilities directly (but it's a bit slower and less numerically stable).

For multilabel binary classification, the only difference is that you must have one neuron per label in the output layer.

Now we can train the model as usual (e.g., using the `train()` function with an SGD optimizer). To evaluate the model, we can use the `Accuracy` streaming metric from the `torchmetrics` library, and move it to the GPU:

```
accuracy = torchmetrics.Accuracy(task="multiclass", num_classes=10).to(device)
```



Training the model will take a few minutes with a GPU (or much longer without one). Handling images requires significantly more compute and memory than handling low-dimensional data.

The model reaches around 92.8% accuracy on the training set, and 87.2% accuracy on the validation set (the results might differ a bit depending on the hardware accelerator you use). This means there's a little bit of overfitting going on, so you may want to reduce the number of neurons or add some regularization (see [Chapter 11](#)).

Now that the model is trained, we can use it to make predictions on new images. As an example, let's make predictions for the first batch in the validation set, and look at the results for the first three images:

```
>>> model.eval()
>>> X_new, y_new = next(iter(valid_loader))
>>> X_new = X_new[:3].to(device)
>>> with torch.no_grad():
...     y_pred_logits = model(X_new)
...
>>> y_pred = y_pred_logits.argmax(dim=1) # index of the largest logit
>>> y_pred
tensor([7, 4, 2], device='cuda:0')
>>> [train_and_valid_data.classes[index] for index in y_pred]
['Sneaker', 'Coat', 'Pullover']
```

For each image, the predicted class is the one with the highest logit. In this example, all three predictions are correct!

But what if we want the model's estimated probabilities? For this, we need to compute the softmax of the logits manually, since the model does not include the softmax activation function on the output layer, as we discussed earlier. We could create an `nn.Softmax` module and pass it the logits, but we can also just call the `softmax()` function, which is just one of many functions you will find in the `torch.nn.functional` module (by convention, this module is usually imported as `F`). It doesn't make much difference, it just avoids creating a module instance that we don't need:

```
>>> import torch.nn.functional as F  
>>> y_proba = F.softmax(y_pred_logits, dim=1)  
>>> y_proba.round(decimals=3)  
tensor([[0.000, 0.000, 0.000, 0.000, 0.000, 0.001, 0.000, 0.911, 0.000, 0.088],  
       [0.000, 0.000, 0.004, 0.000, 0.996, 0.000, 0.000, 0.000, 0.000, 0.000],  
       [0.000, 0.000, 0.625, 0.000, 0.335, 0.000, 0.039, 0.000, 0.000, 0.000]],  
       device='cuda:0')
```

Just like in [Chapter 9](#), the model is very confident about the first two predictions: 91.1% and 99.6%, respectively.



If you wish to apply label smoothing during training, just set the `label_smoothing` hyperparameter of the `nn.CrossEntropyLoss` to the amount of smoothing you wish, between 0 and 1 (e.g., 0.05).

It can often be useful to get the model's top  $k$  predictions. For this, we can use the `torch.topk()` function, which returns a tuple containing both the top  $k$  values and their indices:

```
>>> y_top4_logits, y_top4_indices = torch.topk(y_pred_logits, k=4, dim=1)  
>>> y_top4_probas = F.softmax(y_top4_logits, dim=1)  
>>> y_top4_probas.round(decimals=3)  
tensor([[0.9110, 0.0880, 0.0010, 0.0000],  
       [0.9960, 0.0040, 0.0000, 0.0000],  
       [0.6250, 0.3350, 0.0390, 0.0000]], device='cuda:0')  
  
>>> y_top4_indices  
tensor([[7, 9, 5, 8],  
       [4, 2, 6, 0],  
       [2, 4, 6, 0]], device='cuda:0')
```

For the first image, the model's best guess is class 7 (Sneaker) with 91.1% confidence, its second best guess is class 9 (Ankle boot) with 8.8% confidence, and so on.



The Fashion MNIST dataset is balanced, meaning it has the same number of instances of each class. When dealing with an unbalanced dataset, you should generally give more weight to the rare classes and less weight to the frequent ones, or else your model will be biased toward the more frequent classes. You can do this by setting the `weight` argument of the `nn.CrossEntropyLoss`. For example, if there are three classes with 900, 700, and 400 instances, respectively (i.e., 2000 instances in total), then the respective weights should be 2000/900, 2000/700, and 2000/400. It's preferable to normalize these weights to ensure they add up to 1, so in this example you would set `weight=torch.tensor([0.2205, 0.2835, 0.4961])`.

Your PyTorch superpowers are growing: you can now build, train, and evaluate both regression and classification neural nets. The next step is to learn how to fine-tune the model hyperparameters.

## Fine-Tuning Neural Network Hyperparameters with Optuna

We discussed how to manually pick reasonable values for your model's hyperparameters in [Chapter 9](#), but what if you want to go further and automatically search for good hyperparameter values? One option is to convert your PyTorch model to a Scikit-Learn estimator, either by writing your own custom estimator class or by using a wrapper library such as Skorch (<https://skorch.readthedocs.io>), and then use `GridSearchCV` or `RandomizedSearchCV` to fine-tune the hyperparameters, as you did in [Chapter 2](#). However, you will usually get better results by using a dedicated fine-tuning library such as Optuna (<https://optuna.org>), Ray Tune (<https://docs.ray.io>), or Hyperopt (<https://hyperopt.github.io/hyperopt>). These libraries offer several powerful tuning strategies, and they're highly customizable.

Let's look at an example using Optuna. It is not preinstalled on Colab, so we need to install it using `%pip install optuna` (if you prefer to run the code locally, please follow the installation instructions at <https://hml.info/install-p>). Let's tune the learning rate and the number of neurons in the hidden layers (for simplicity, we will use the same number of neurons in both hidden layers). First, we need to define a function that Optuna will call many times to perform hyperparameter tuning: this function must take a `Trial` object and use it to ask Optuna for hyperparameter values, and then use these hyperparameter values to build and train a model. Finally, the function must evaluate the model (typically on the validation set) and return the metric:

```

import optuna

def objective(trial):
    learning_rate = trial.suggest_float("learning_rate", 1e-5, 1e-1, log=True)
    n_hidden = trial.suggest_int("n_hidden", 20, 300)
    model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=n_hidden,
                           n_hidden2=n_hidden, n_classes=10).to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
    [...] # train the model, then evaluate it on the validation set
    return validation_accuracy

```

The `suggest_float()` and `suggest_int()` methods let us ask Optuna for a good hyperparameter value in a given range (Optuna also provides a `suggest_categorical()` method). For the `learning_rate` hyperparameter, we ask for a value between  $10^{-5}$  and  $10^{-1}$ , and since we don't know what the optimal scale is, we add `log=True`: this will make Optuna sample values from a log distribution, which makes it explore all possible scales. If we used the default uniform distribution instead, Optuna would be very unlikely to explore tiny values.

To start hyperparameter tuning, we create a `Study` object and call its `optimize()` method, passing it the objective function we just defined, as well as the number of trials to run (i.e., the number of times Optuna should call the objective function). Since our objective function returns a score—higher is better—we set `direction="maximize"` when creating the study (by default, Optuna tries to *minimize* the objective). To ensure reproducibility, we also set PyTorch's random seed, as well as the random seed used by Optuna's sampler:

```

torch.manual_seed(42)
sampler = optuna.samplers.TPESampler(seed=42)
study = optuna.create_study(direction="maximize", sampler=sampler)
study.optimize(objective, n_trials=5)

```

By default, Optuna uses the *Tree-structured Parzen Estimator* (TPE) algorithm to optimize the hyperparameters: this is a sequential model-based optimization algorithm, meaning it learns from past results to better select promising hyperparameters. In other words, Optuna starts with random hyperparameter values, but it progressively focuses its search on the most promising regions of the hyperparameter space. This allows Optuna to find much better hyperparameters than random search in the same amount of time.



You can add more hyperparameters to the search space, such as the batch size, the type of optimizer, the number of hidden layers, or the type of activation function, but remember that the search space will grow exponentially as you add more hyperparameters, so make sure it's worth the extra search time and compute.

Once Optuna is done, you can look at the best hyperparameters it found, as well as the corresponding validation accuracy:

```
>>> study.best_params
{'learning_rate': 0.08525846269447772, 'n_hidden': 116}
>>> study.best_value
0.8867999911308289
```

This is slightly better than the performance we got earlier. If you increase `n_trials` up to 50 or more, you will get much better results, but of course it will take hours to run. You can also just run `optimize()` repeatedly and stop once you are happy with the performance.

Optuna can also run trials in parallel across multiple machines, which can offer a near linear speed boost. For this, you will need to set up a SQL database (e.g., SQLite or PostgreSQL), and set the `storage` parameter of the `create_study()` function to point to that database. You also need to set the study's name via the `study_name` parameter, and set `load_if_exists=True`. After that, you can copy your hyperparameter tuning script to multiple machines, and run it on each one (if you are using random seeds, make sure they are different on each machine). The scripts will work in parallel, reading and writing the trial results to the database. This has the additional benefit of keeping a full log of all your experiment results.

You may have noticed that we assumed that the `objective()` function had direct access to the training set and validation, presumably via global variables. In general, it's much cleaner to pass them as extra arguments to the `objective()` function, for example, like this:

```
def objective(trial, train_loader, valid_loader):
    [...] # the rest of the function remains the same as above

objective_with_data = lambda trial: objective(
    trial, train_loader=train_loader, valid_loader=valid_loader)
study.optimize(objective_with_data, n_trials=5)
```

To set the extra arguments (the dataset loaders in this case), we just create a lambda function when needed and pass it to the `optimize()` method. Alternatively, you can use the `functools.partial()` function which creates a thin wrapper function around the given callable to provide default values for any number of arguments:

```
from functools import partial

objective_with_data = partial(objective, train_loader=train_loader,
                             valid_loader=valid_loader)
```

It's often possible to quickly tell that a trial is absolutely terrible: for example, when the loss shoots up during the first epoch, or when the model barely improves during the first few epochs. In such a case, it's a good idea to interrupt training early to avoid wasting time and compute. You can simply return the model's current validation

accuracy and hope that Optuna will learn to avoid this region of hyperparameter space. Alternatively, you can interrupt training by raising the `optuna.TrialPruned` exception: this tells Optuna to ignore this trial altogether. In many cases, this leads to a more efficient search because it avoids polluting Optuna’s search algorithm with many noisy model evaluations.

Optuna comes with several Pruner classes that can detect and prune bad trials. For example, the `MedianPruner` will prune trials whose performance is below the median performance, at regular intervals during training. It starts pruning after a given number of trials have completed, controlled by `n_startup_trials` (5 by default). For each trial after that, it lets training start for a few epochs, controlled by `n_warmup_steps` (0 by default); then every few epochs (controlled by `interval_steps`), it ensures that the model’s performance is better than the median performance at the same epoch in past trials. To use this pruner, create an instance and pass it to the `create_study()` method:

```
pruner = optuna.pruners.MedianPruner(n_startup_trials=5, n_warmup_steps=0,
                                         interval_steps=1)
study = optuna.create_study(direction="maximize", sampler=sampler,
                             pruner=pruner)
```

Then in the `objective()` function, add the following code so it runs after each epoch:

```
for epoch in range(n_epochs):
    [...] # train the model for one epoch
    validation_accuracy = [...] # evaluate the model's validation accuracy
    trial.report(validation_accuracy, epoch)
    if trial.should_prune():
        raise optuna.TrialPruned()
```

The `report()` method informs Optuna of the current validation accuracy and epoch, so it can determine whether the trial should be pruned. If `trial.should_prune()` returns `True`, we raise a `TrialPruned` exception.



Optuna has many other features well worth exploring, such as visualization tools, persistence tools for trial results and other artifacts, a dashboard for human-in-the-loop optimization, and many other algorithms for hyperparameter search and trial pruning.

Once you are happy with the hyperparameters, you can train the model on the full training set (i.e., the training set plus the validation set), then evaluate it on the test set. Hopefully, it will perform great! If it does, you will want to save the model, then load it and use it in production: that’s the final topic of this chapter.

# Saving and Loading PyTorch Models

The simplest way to save a PyTorch model is to use the `torch.save()` method, passing it the model and the filepath. The model object is serialized using Python's `pickle` module (which can convert objects into a sequence of bytes), then the result is compressed (zip) and saved to disk. The convention is to use the `.pt` or `.pth` extension for PyTorch files:

```
torch.save(model, "my_fashion_mnist.pt")
```

Simple! Now you can load the model (e.g., in your production code) just as easily:

```
loaded_model = torch.load("my_fashion_mnist.pt", weights_only=False)
```



If your model uses any custom functions or classes (e.g., `ImageClassifier`), then `torch.save()` only saves references to them, not the code itself. Therefore you must ensure that any custom code is loaded in the Python environment before calling `torch.load()`. Also make sure to use the same version of the code to avoid any mismatch issues.

Setting `weights_only=False` ensures that the whole model object is loaded rather than just the model parameters. Then you can use the loaded model for inference. Don't forget to switch to evaluation mode first using the `eval()` method:

```
loaded_model.eval()  
y_pred_logits = loaded_model(X_new)
```

This is nice and easy, but unfortunately this approach has some very serious drawbacks:

- Firstly, `pickle`'s serialization format is notoriously insecure. While `torch.save()` doesn't save custom code, the `pickle` format supports it, so a hacker could inject malicious code in a saved PyTorch model: this code would be run automatically by the `pickle` module when the model is loaded. So always make sure you fully trust the model's source before you load it this way.
- Second, `pickle` is somewhat brittle. It can vary depending on the Python version (e.g., there were big changes between Python 3.7 and 3.8), and it saves specific filepaths to locate code, which can break if the loading environment has a different folder structure.

To avoid these issues, it is recommended to save and load the model weights only, rather than the full model object:

```
torch.save(model.state_dict(), "my_fashion_mnist_weights.pt")
```

The state dictionary returned by the `state_dict()` method is just a Python `OrderedDict` containing an entry for each parameter returned by the `named_parameters()` method. It also contains buffers, if the model has any: a buffer is just a regular tensor that was registered with the model (or any of its submodules) using the `register_buffer()` method. Buffers hold extra data that needs to be stored along with the model, but that is not a model parameter. We will see an example in [Chapter 11](#) with the batch-norm layer.

To load these weights, we must first create a model with the exact same structure, then load the weights using `torch.load()` with `weights_only=True`, and finally call the model's `load_state_dict()` method with the loaded weights:

```
new_model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=300, n_hidden2=100,
                            n_classes=10)
loaded_weights = torch.load("my_fashion_mnist_weights.pt", weights_only=True)
new_model.load_state_dict(loaded_weights)
new_model.eval()
```

The saved model contains only data, and the `load()` function makes sure of that, so this is safe, and also much less likely to break between Python versions or to cause any deployment issue. However, it only works if you are able to create the exact same model architecture before loading the state dictionary. For this, you need to know the number of layers, the number of neurons per layer, and so on. It's a good idea to save this information along with the state dictionary:

```
model_data = {
    "model_state_dict": model.state_dict(),
    "model_hyperparameters": {"n_inputs": 1 * 28 * 28, "n_hidden1": 300, [...]}
}
torch.save(model_data, "my_fashion_mnist_model.pt")
```

You can then load this dictionary, construct the model based on the saved hyperparameters, and load the state dictionary into this model:

```
loaded_data = torch.load("my_fashion_mnist_model.pt", weights_only=True)
new_model = ImageClassifier(**loaded_data["model_hyperparameters"])
new_model.load_state_dict(loaded_data["model_state_dict"])
new_model.eval()
```

If you want to be able to continue training where it left off, you will also need to save the optimizer's state dictionary, its hyperparameters, and any other training information you may need, such as the current epoch and the loss history.



The `safetensors` library by Hugging Face is another popular way to save model weights safely.

There is yet another way to save and load your model: by first converting it to TorchScript. This also makes it possible to speed up your model's inference.

## Compiling and Optimizing a PyTorch Model

PyTorch comes with a very nice feature: it can automatically convert your model's code to *TorchScript*, which you can think of as a statically typed subset of Python. There are two main benefits:

- First, TorchScript code can be compiled and optimized to produce significantly faster models. For example, multiple operations can often be fused into a single, more efficient operation. Operations on constants (e.g.,  $2 * 3$ ) can be replaced with their result (e.g., 6); this is called *constant folding*. Unused code can be pruned, and so on.
- Secondly, TorchScript can be serialized, saved to disk, and then loaded and executed in Python or in a C++ environment using the LibTorch library. This makes it possible to run PyTorch models on a wide range of devices, including embedded devices.

There are two ways to convert a PyTorch model to TorchScript. The first way is called *tracing*. PyTorch just runs your model with some sample data, logs every operation that takes place, and then converts this log to TorchScript. This is done using the `torch.jit.trace()` function:

```
torchscript_model = torch.jit.trace(model, X_new)
```

This generally works well with static models whose `forward()` method doesn't use conditionals or loops. However, if you try to trace a model that includes an `if` or `match` statement, then only the branch that is actually executed will be captured by TorchScript, which is generally not what you want. Similarly, if you use tracing with a model that contains a loop, then the TorchScript code will contain one copy of the operations within that loop for each iteration that was actually executed. Again, not what you generally want.

For such dynamic models, you will probably want to try another approach named *scripting*. In this case, PyTorch actually parses your Python code directly and converts it to TorchScript. This method supports `if` statements and `while` loops properly, as long as the conditions are tensors. It also supports `for` loops when iterating over tensors. However, it only works on a subset of Python. For example, you cannot use global variables, Python generators (`yield`), complex list comprehensions, variable length function arguments (`*args` or `**kwargs`), or `match` statements. Moreover, types must be fixed (a function cannot return an integer in some cases and a float in others), and you can only call other functions if they also respect these rules, so no standard library, no third-party libraries, etc. (see the documentation for the full list

of constraints). This sounds daunting, but for most real-world models, these rules are actually not too hard to respect, and you can save your model like this:

```
torchscript_model = torch.jit.script(model)
```

Regardless of whether you use tracing or scripting to produce your TorchScript model, you can then further optimize it:

```
optimized_model = torch.jit.optimize_for_inference(torchscript_model)
```

TorchScript models can only be used for inference, not for training, since the TorchScript environment doesn't support gradient tracking or parameter updates.

Finally, you can save a TorchScript model using its `save()` method:

```
torchscript_model.save('my_fashion_mnist_torchscript.pt')
```

And then load it using the `torch.jit.load()` function:

```
loaded_torchscript_model = torch.jit.load("my_fashion_mnist_torchscript.pt")
```

One important caveat: TorchScript is no longer under active development—bugs are fixed but no new features are added. It still works fine and it remains one of the best ways to run your PyTorch models in a C++ environment,<sup>13</sup> but since the release of PyTorch 2.0 in March 2023, the PyTorch team has been focusing its efforts on a new set of compilation tools centered around the `torch.compile()` function, which you can use very easily:

```
compiled_model = torch.compile(model)
```

The resulting model can now be used normally, and it will automatically be compiled and optimized when you use it. This is called Just-In-Time (JIT) compilation, as opposed to Ahead-Of-Time (AOT) compilation. Under the hood, `torch.compile()` relies on *TorchDynamo* (or *Dynamo* for short) which hooks directly into Python bytecode to capture the model's computation graph at inference time. Having access to the bytecode allows Dynamo to efficiently and reliably capture the computation graph, properly handling conditionals and loops, while also benefiting from dynamic information that can be used to better optimize the model. The actual compilation and optimization is performed by default by a backend component named *TorchInductor*, which in turn relies on the Triton language to generate highly efficient GPU code (Nvidia only), or on the OpenMP API for CPU optimization. PyTorch 2.x offers a few other optimization backends, including the XLA backend for Google's TPU devices: just set `device="xla"` when calling `torch.compile()`.

With that, you now have all the tools you need to start building and training complex and efficient neural networks. I hope you enjoyed this introduction to PyTorch! We

---

<sup>13</sup> Another popular option is exporting your PyTorch model to the open ONNX standard using `torch.onnx.export()`. The ONNX model can then be used for inference in a wide variety of environments.

covered a lot, but the adventure is only beginning: in the next chapter we will discuss techniques to train very deep nets. After that, we will dive into other popular neural network architectures: convolutional neural networks for image processing, recurrent neural networks for sequential data, transformers for text (and much more), autoencoders for representation learning, and generative adversarial networks and diffusion models to generate data.<sup>14</sup> Then we will visit reinforcement learning to train autonomous agents, and finally, we will learn more about deploying and optimizing your PyTorch models. Let's go!

## Exercises

1. PyTorch is similar to NumPy in many ways, but it offers some extra features. Can you name the most important ones?
2. What is the difference between `torch.exp()` and `torch.exp_()`, or between `torch.relu()` and `torch.relu_()`?
3. What are two ways to create a new tensor on the GPU?
4. What are three ways to perform tensor computations without using autograd?
5. Will the following code cause a `RuntimeError`? What if you replace the second line with `z = t.cos_().exp()`? And what if you replace it with `z = t.exp_().cos_()`?

```
t = torch.tensor(2.0, requires_grad=True)
z = t.cos().exp_()
z.backward()
```

How about the following code, will it cause an error? And what if you replace the third line with `w = v.cos_() * v.sin_()`? Will `w` have the same value in both cases?

```
u = torch.tensor(2.0, requires_grad=True)
v = u + 1
w = v.cos() * v.sin_()
w.backward()
```

6. Suppose you create a `Linear(100, 200)` module. How many neurons does it have? What is the shape of its `weight` and `bias` parameters? What input shape does it expect? What output shape does it produce?
7. What are the main steps of a PyTorch training loop?
8. Why is it recommended to create the optimizer *after* the model is moved to the GPU?

---

<sup>14</sup> A few extra ANN architectures are presented in the online notebook at <https://homl.info/extranns>.

9. What `DataLoader` options should you generally set to speed up training when using a GPU?
10. What are the main classification losses provided by PyTorch, and when should you use each of them?
11. Why is it important to call `model.train()` before training and `model.eval()` before evaluation?
12. What is the difference between `torch.jit.trace()` and `torch.jit.script()`?
13. Use autograd to find the gradient vector of  $f(x, y) = \sin(x^2 y)$  at the point  $(x, y) = (1.2, 3.4)$ .
14. Create a custom `Dense` module that replicates the functionality of an `nn.Linear` module followed by an `nn.ReLU` module. Try implementing it first using the `nn.Linear` and `nn.ReLU` modules, and then reimplement it using `nn.Parameter` and the `relu()` function.
15. Build and train a classification MLP on the CoverType dataset:
  - a. Load the dataset using `sklearn.datasets.fetch_covtype()` and create a custom PyTorch `Dataset` for this data.
  - b. Create data loaders for training, validation, and testing.
  - c. Build a custom MLP module to tackle this classification task. You can optionally use the custom `Dense` module from the previous exercise.
  - d. Train this model on the GPU, and try to reach 93% accuracy on the test set. For this, you will likely have to perform hyperparameter search to find the right number of layers and neurons per layer, a good learning rate and batch size, and so on. You can optionally use Optuna for this.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.



# Training Deep Neural Networks

In [Chapter 10](#) you built, trained, and fine-tuned several artificial neural networks using PyTorch. But they were shallow nets with just a few hidden layers. What if you need to tackle a complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper ANN, perhaps with dozens or even hundreds of layers, each containing hundreds of neurons, linked by hundreds of thousands of connections. Training a deep neural network isn't a walk in the park. Here are some of the problems you could run into:

- You may be faced with the problem of gradients growing ever smaller or larger when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- You might not have enough training data for such a large network, or it might be too costly to label.
- Training may be extremely slow.
- A model with millions of parameters risks severely overfitting the training set, especially if there are not enough training instances or if they are too noisy.

In this chapter we will go through each of these problems and present various techniques to solve them. We will start by exploring the vanishing and exploding gradients problems and some of their most popular solutions, including smart weight initialization, better activation functions, batch-norm, layer-norm, and gradient clipping. Next, we will look at transfer learning and unsupervised pretraining, which can help you tackle complex tasks even when you have little labeled data. Then we will discuss a variety of optimizers that can speed up training large models tremendously. We will also discuss how you can tweak the learning rate during training to speed up training and produce better models. Finally, we will cover a few popular

regularization techniques for large neural networks:  $\ell_1$  and  $\ell_2$  regularization, dropout, Monte Carlo dropout, and max-norm regularization.

With these tools, you will be able to train all sorts of deep nets. Welcome to *deep learning!*

## The Vanishing/Exploding Gradients Problems

As discussed in [Chapter 9](#), the backpropagation algorithm's second phase works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard to each parameter in the network, it uses these gradients to update each parameter with a gradient descent step.

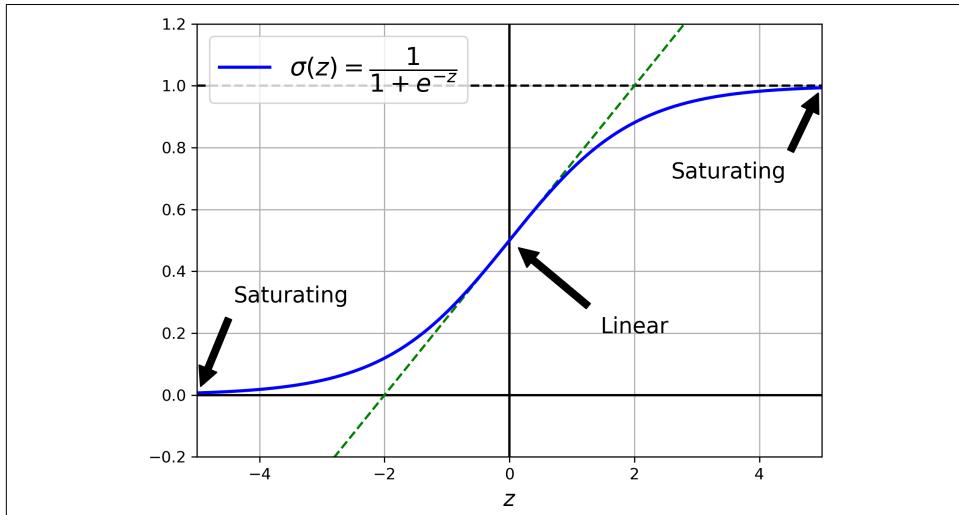
Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the gradient descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution. This is called the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which surfaces most often in recurrent neural networks (see [Chapter 13](#)). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

This unfortunate behavior was empirically observed long ago, and it was one of the reasons deep neural networks were mostly abandoned in the early 2000s. It wasn't clear what caused the gradients to be so unstable when training a DNN, but some light was shed in a [2010 paper](#) by Xavier Glorot and Yoshua Bengio.<sup>1</sup> The authors found a few suspects, including the combination of the popular sigmoid (logistic) activation function and the weight initialization technique that was most popular at the time (i.e., a normal distribution with a mean of 0 and a standard deviation of 1). In short, they showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This saturation is actually made worse by the fact that the sigmoid function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the sigmoid function in deep networks).

---

<sup>1</sup> Xavier Glorot and Yoshua Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks", *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010): 249–256.

Looking at the sigmoid activation function (see [Figure 11-1](#)), you can see that when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0 (i.e., the curve is flat at both extremes). Thus, when backpropagation kicks in it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.



*Figure 11-1. Sigmoid activation function saturation*

## Glorot Initialization and He Initialization

In their paper, Glorot and Bengio proposed a way to significantly alleviate the unstable gradients problem. They pointed out that we need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argued that we need the variance of the outputs of each layer to be equal to the variance of its inputs,<sup>2</sup> and we need the gradients to have equal variance before and after flowing through a layer in the reverse direction (please check out the paper if you are interested in the mathematical details). It is actually not possible to guarantee both unless the layer has an equal number of inputs and outputs (these numbers are

---

<sup>2</sup> Here's an analogy: if you set a microphone amplifier's volume knob too close to zero, people won't hear your voice, but if you set it too close to the max, your voice will be saturated and people won't understand what you are saying. Now imagine a chain of such amplifiers: they all need to be set properly in order for your voice to come out loud and clear at the end of the chain. Your voice has to come out of each amplifier at the same amplitude as it came in.

called the *fan-in* and *fan-out* of the layer), but Glorot and Bengio proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly, as described in [Equation 11-1](#), where  $fan_{avg} = (fan_{in} + fan_{out}) / 2$ . This initialization strategy is called *Xavier initialization* or *Glorot initialization*, after the paper's first author.

*Equation 11-1. Glorot initialization (when using the sigmoid activation function)*

Normal distribution with mean 0 and variance  $\sigma^2 = \frac{1}{fan_{avg}}$

Or a uniform distribution between  $-r$  and  $+r$ , with  $r = \sqrt{\frac{3}{fan_{avg}}}$

If you replace  $fan_{avg}$  with  $fan_{in}$  in [Equation 11-1](#), you get an initialization strategy that Yann LeCun proposed in the 1990s. He called it *LeCun initialization*. Genevieve Orr and Klaus-Robert Müller even recommended it in their 1998 book *Neural Networks: Tricks of the Trade* (Springer). LeCun initialization is equivalent to Glorot initialization when  $fan_{in} = fan_{out}$ . It took over a decade for researchers to realize how important this trick is. Using Glorot initialization can speed up training considerably, and it is one of the tricks that led to the success of deep learning.

Some papers have provided similar strategies for different activation functions, most notably a [2015 paper by Kaiming He et al.](#)<sup>3</sup> These strategies differ only by the scale of the variance and whether they use  $fan_{avg}$  or  $fan_{in}$ , as shown in [Table 11-1](#) (for the uniform distribution, just use  $r = \sqrt{3\sigma^2}$ ). The initialization strategy proposed for the ReLU activation function and its variants is called *He initialization* or *Kaiming initialization*, after the paper's first author. For SELU, use Yann LeCun's initialization method, preferably with a normal distribution. We will cover all these activation functions shortly.

*Table 11-1. Initialization parameters for each type of activation function*

Initialization	Activation functions	$\sigma^2$ (Normal)
Xavier Glorot	None, tanh, sigmoid, softmax	$1 / fan_{avg}$
Kaiming He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish, SwiGLU, ReLU <sup>2</sup>	$2 / fan_{in}$
Yann LeCun	SELU	$1 / fan_{in}$

For historical reasons, PyTorch's `nn.Linear` module initializes its weights using Kaiming uniform initialization, except the weights are scaled down by a factor of

---

<sup>3</sup> Kaiming He et al., "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015): 1026–1034.

$\sqrt{6}$  (and the bias terms are also initialized randomly). Sadly, this is not the optimal scale for any common activation function.<sup>4</sup> One solution is to simply multiply the weights by  $\sqrt{6}$  (i.e.,  $6^{0.5}$ ) just after creating the `nn.Linear` layer to get proper Kaiming initialization. To do this, we can update the parameter's `data` attribute. We will also zero out the biases, as there's no benefit in randomly initializing them:

```
import torch
import torch.nn as nn

layer = nn.Linear(40, 10)
layer.weight.data *= 6 ** 0.5 # Kaiming init (or 3 ** 0.5 for LeCun init)
torch.zeros_(layer.bias.data)
```

This works, but it's clearer and less error-prone to use one of the initialization functions available in the `torch.nn.init` module:

```
nn.init.kaiming_uniform_(layer.weight)
nn.init.zeros_(layer.bias)
```

If you want to apply the same initialization method to the weights of every `nn.Linear` layer in a model, you can do so in the model's constructor, after creating each `nn.Linear` layer. Alternatively, you can write a subclass of the `nn.Linear` class and tweak its constructor to initialize the weights as you wish. But arguably the simplest option is to write a little function that takes a module, checks whether it's an instance of the `nn.Linear` class, and if so, applies the desired initialization function to its weights. You can then apply this function to the model and all of its submodules by passing it to the model's `apply()` method. For example:

```
def use_he_init(module):
    if isinstance(module, nn.Linear):
        nn.init.kaiming_uniform_(module.weight)
        nn.init.zeros_(module.bias)

model = nn.Sequential(nn.Linear(50, 40), nn.ReLU(), nn.Linear(40, 1), nn.ReLU())
model.apply(use_he_init)
```

The `torch.nn.init` module also contains an `orthogonal_()` function which initializes the weights using a random orthogonal matrix, as proposed in a [2014 paper](#) by Andrew Saxe et al.<sup>5</sup> Orthogonal matrices have a number of useful mathematical properties, including the fact that they preserve norms: given an orthogonal matrix  $\mathbf{W}$  and an input vector  $\mathbf{x}$ , the norm of  $\mathbf{Wx}$  is equal to the norm of  $\mathbf{x}$ , and therefore the magnitude of the inputs is preserved in the outputs. When the inputs are

---

<sup>4</sup> A PyTorch issue (#18182) has been open since 2019 to update the weight initialization to use the current best practices.

<sup>5</sup> Andrew Saxe et al., “Exact solutions to the nonlinear dynamics of learning in deep linear neural networks”, ICLR (2014).

standardized, this results in a stable variance through the layer, which prevents the activations and gradients from vanishing or exploding in a deep network (at least at the beginning of training). This initialization technique is much less common than the initialization techniques discussed earlier, but it can work well in recurrent neural nets (Chapter 13) or generative adversarial networks (Chapter 18).

And that's it! Scaling the weights properly will give a deep neural net a much better starting point for training.



In a classifier, it's generally a good idea to scale down the weights of the output layer during initialization (e.g., by a factor of 10). Indeed, this will result in smaller logits at the beginning of training, which means they will be closer together, and hence the estimated probabilities will also be closer together. In other words, it encourages the model to be less confident about its predictions when training starts: this will avoid extreme losses and huge gradients that can often make the model's weights bounce around randomly at the start of training, losing time and potentially preventing the model from learning anything.

## Better Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the problems with unstable gradients were in part due to a poor choice of activation function. Until then most people had assumed that if Mother Nature had chosen to use something pretty close to sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks—in particular, the ReLU activation function, mostly because it does not saturate for positive values, and also because it is very fast to compute.

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively “die”, meaning they stop outputting anything other than 0. In some cases, you may find that half of your network’s neurons are dead, especially if you used a large learning rate. A neuron dies when its weights get tweaked in such a way that the input of the ReLU function (i.e., the weighted sum of the neuron’s inputs plus its bias term) is negative for all instances in the training set. When this happens, it just keeps outputting zeros, and gradient descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative.<sup>6</sup>

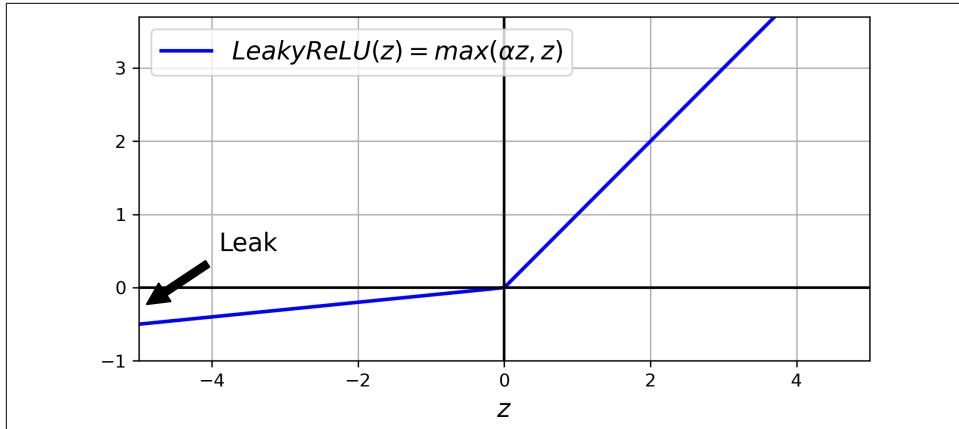
---

<sup>6</sup> A dead neuron may come back to life if its inputs evolve over time and eventually return within a range where the ReLU activation function gets a positive input again. For example, this may happen if gradient descent tweaks the neurons in the layers below the dead neuron.

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*.

### Leaky ReLU

The leaky ReLU activation function is defined as  $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$  (see [Figure 11-2](#)). The hyperparameter  $\alpha$  defines how much the function “leaks”: it is the slope of the function for  $z < 0$ . Having a slope for  $z < 0$  ensures that leaky ReLUs never actually die; they can go into a long coma, but they have a chance to eventually wake up. A [2015 paper](#) by Bing Xu et al.<sup>7</sup> compared several variants of the ReLU activation function, and one of its conclusions was that the leaky variants always outperformed the strict ReLU activation function. In fact, setting  $\alpha = 0.2$  (a huge leak) seemed to result in better performance than  $\alpha = 0.01$  (a small leak). The paper also evaluated the *randomized leaky ReLU* (RReLU), where  $\alpha$  is picked randomly in a given range during training and is fixed to an average value during testing. RReLU also performed fairly well and seemed to act as a regularizer, reducing the risk of overfitting. Finally, the paper evaluated the *parametric leaky ReLU* (PReLU), where  $\alpha$  is authorized to be learned during training: instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameter. PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.



*Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values*

As you might expect, PyTorch includes modules for each of these activation functions: `nn.LeakyReLU`, `nn.RReLU`, and `nn.PReLU`. Just like for other ReLU variants, you should use these along with Kaiming initialization, but the variance should be

<sup>7</sup> Bing Xu et al., “Empirical Evaluation of Rectified Activations in Convolutional Network”, arXiv preprint arXiv:1505.00853 (2015).

slightly smaller due to the negative slope: it should be scaled down by a factor of  $1 + \alpha^2$ . PyTorch supports this: you can pass the  $\alpha$  hyperparameter to the `kaiming_uniform_()` and `kaiming_normal_()` functions, along with `nonlinearity="leaky_relu"` to get the appropriately adjusted Kaiming initialization:

```
alpha = 0.2
model = nn.Sequential(nn.Linear(50, 40), nn.LeakyReLU(negative_slope=alpha))
nn.init.kaiming_uniform_(model[0].weight, alpha, nonlinearity="leaky_relu")
```

ReLU, leaky ReLU, and PReLU all suffer from the fact that they are not smooth functions: their slopes abruptly change at  $z = 0$ . As we saw in [Chapter 4](#) when we discussed lasso, this sort of discontinuity in the derivatives can make gradient descent bounce around the optimum and slow down convergence. So now we will look at some smooth variants of the ReLU activation function, starting with ELU and SELU.

## ELU and SELU

In 2015, a [paper](#) by Djork-Arné Clevert et al.<sup>8</sup> proposed a new activation function, called the *exponential linear unit* (ELU), that outperformed all the ReLU variants in the authors' experiments: training time was reduced, and the neural network performed better on the test set. [Equation 11-2](#) shows this activation function's definition.

*Equation 11-2. ELU activation function*

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

The ELU activation function looks a lot like the ReLU function (see [Figure 11-3](#)), with a few major differences:

- It takes on negative values when  $z < 0$ , which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem. The hyperparameter  $\alpha$  defines the opposite of the value that the ELU function approaches when  $z$  is a large negative number. It is usually set to 1, but you can tweak it like any other hyperparameter.
- It has a nonzero gradient for  $z < 0$ , which avoids the dead neurons problem.
- If  $\alpha$  is equal to 1, then the function is smooth everywhere, including around  $z = 0$ , which helps speed up gradient descent since it does not bounce as much to the left and right of  $z = 0$ .

---

<sup>8</sup> Djork-Arné Clevert et al., “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)”, *Proceedings of the International Conference on Learning Representations*, arXiv preprint (2015).

Using ELU with PyTorch is as easy as using the `nn.ELU` module, along with Kaiming initialization. The main drawback of the ELU activation function is that it is slower to compute than the ReLU function and its variants (due to the use of the exponential function). Its faster convergence rate during training may compensate for that slow computation, but still, at test time an ELU network will be a bit slower than a ReLU network.

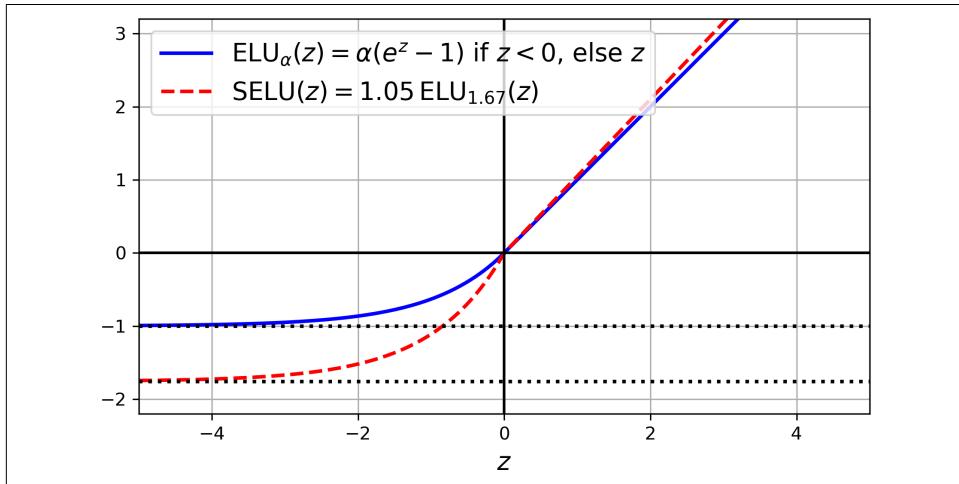


Figure 11-3. ELU and SELU activation functions

Not long after, a [2017 paper](#) by Günter Klambauer et al.<sup>9</sup> introduced the *scaled ELU* (SELU) activation function: as its name suggests, it is a scaled variant of the ELU activation function (about 1.05 times ELU, using  $\alpha \approx 1.67$ ). The authors showed that if you build a neural network composed exclusively of a stack of dense layers (i.e., an MLP), and if all hidden layers use the SELU activation function, then the network will *self-normalize*: the output of each layer will tend to preserve a mean of 0 and a standard deviation of 1 during training, which solves the vanishing/exploding gradients problem. As a result, the SELU activation function may outperform other activation functions for MLPs, especially deep ones. To use it with PyTorch, just use `nn.SELU`. There are, however, a few conditions for self-normalization to happen (see the paper for the mathematical justification):

- The input features must be standardized: mean 0 and standard deviation 1.
- Every hidden layer's weights must be initialized using LeCun normal initialization.

---

<sup>9</sup> Günter Klambauer et al., “Self-Normalizing Neural Networks”, *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 972–981.

- The self-normalizing property is only guaranteed with plain MLPs. If you try to use SELU in other architectures, like recurrent networks (see [Chapter 13](#)) or networks with *skip connections* (i.e., connections that skip layers, such as in Wide & Deep neural networks), it will probably not outperform ELU.
- You cannot use regularization techniques like  $\ell_1$  or  $\ell_2$  regularization, batch-norm, layer-norm, max-norm, or regular dropout (these are discussed later in this chapter).

These are significant constraints, so despite its promises, SELU did not gain a lot of traction. Moreover, other activation functions seem to outperform it quite consistently on most tasks. Let's look at some of the most popular ones.

### GELU, Swish, SwiGLU, Mish, and RELU<sup>2</sup>

The *Gaussian Error Linear Unit* (*GELU*) was introduced in a [2016 paper](#) by Dan Hendrycks and Kevin Gimpel.<sup>10</sup> Once again, you can think of it as a smooth variant of the ReLU activation function. Its definition is given in [Equation 11-3](#), where  $\Phi$  is the standard Gaussian cumulative distribution function (CDF):  $\Phi(z)$  corresponds to the probability that a value sampled randomly from a normal distribution of mean 0 and variance 1 is lower than  $z$ .

*Equation 11-3. GELU activation function*

$$\text{GELU}(z) = z\Phi(z)$$

As you can see in [Figure 11-4](#), GELU resembles ReLU: it approaches 0 when its input  $z$  is very negative, and it approaches  $z$  when  $z$  is very positive. However, whereas all the activation functions we've discussed so far were both convex and monotonic,<sup>11</sup> the GELU activation function is neither: from left to right, it starts by going straight, then it wiggles down, reaches a low point around  $-0.17$  (near  $z \approx -0.75$ ), and finally bounces up and ends up going straight toward the top right. This fairly complex shape and the fact that it has a curvature at every point may explain why it works so well, especially for complex tasks: gradient descent may find it easier to fit complex patterns. In practice, it often outperforms every other activation function discussed so far. However, it is a bit more computationally intensive, and the performance boost it provides is not always sufficient to justify the extra cost. That said, it is possible to show that it is approximately equal to  $z\sigma(1.702 z)$ , where  $\sigma$  is the sigmoid function:

---

<sup>10</sup> Dan Hendrycks and Kevin Gimpel, “Gaussian Error Linear Units (GELUs)”, arXiv preprint arXiv:1606.08415 (2016).

<sup>11</sup> A function is convex if the line segment between any two points on the curve never lies below the curve. A monotonic function only increases, or only decreases.

using this approximation also works very well, and it has the advantage of being much faster to compute.

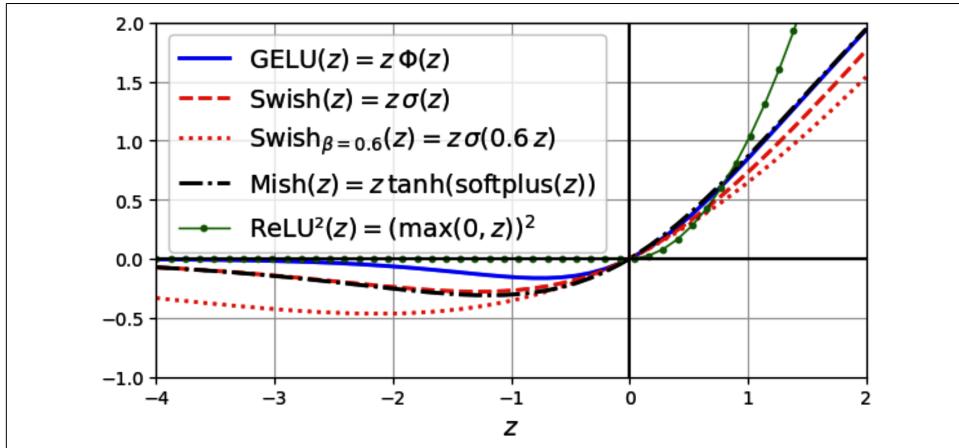


Figure 11-4. GELU, Swish, parametrized Swish, Mish, and  $\text{ReLU}^2$  activation functions

The GELU paper also introduced the *sigmoid linear unit* (SiLU) activation function, which is equal to  $z\sigma(z)$ , but it was outperformed by GELU in the authors' tests. Interestingly, a [2017 paper](#) by Prajit Ramachandran et al.<sup>12</sup> rediscovered the SiLU function by automatically searching for good activation functions. The authors named it *Swish*, and the name caught on. In their paper, Swish outperformed every other function, including GELU. Ramachandran et al. later generalized Swish by adding an extra scalar hyperparameter  $\beta$  to scale the sigmoid function's input. The generalized Swish function is  $\text{Swish}_\beta(z) = z\sigma(\beta z)$ , so GELU is approximately equal to the generalized Swish function using  $\beta = 1.702$ . You can tune  $\beta$  like any other hyperparameter. Alternatively, it's also possible to make  $\beta$  trainable and let gradient descent optimize it (a bit like PReLU): there is typically a single trainable  $\beta$  parameter for the whole model, or just one per layer, to keep the model efficient and avoid overfitting.

A popular Swish variant is *SwiGLU*:<sup>13</sup> the inputs go through the Swish activation function, and in parallel through a linear layer, then both outputs are multiplied itemwise. That's  $\text{SwiGLU}(\mathbf{z}) = \text{Swish}_\beta(\mathbf{z}) \otimes \text{Linear}(\mathbf{z})$ . This is often implemented by doubling the output dimensions of the previous linear layer, then splitting the outputs in two along the feature dimension to get  $\mathbf{z}_1$  and  $\mathbf{z}_2$ , and finally applying:  $\text{SwiGLU}_\beta(\mathbf{z}) = \text{Swish}_\beta(\mathbf{z}_1) \otimes \mathbf{z}_2$ . This is a variant of the *gated linear unit (GLU)*<sup>14</sup>

<sup>12</sup> Prajit Ramachandran et al., "Searching for Activation Functions", arXiv preprint arXiv:1710.05941 (2017).

<sup>13</sup> Noam Shazeer, "GLU Variants Improve Transformer", arXiv preprint arXiv:2002.05202 (2020).

<sup>14</sup> Yann Dauphin et al., "Language Modeling with Gated Convolutional Networks", arXiv preprint arXiv:1612.08083 (2016).

introduced by Facebook researchers in 2016. The itemwise multiplication gives the model more expressive power, allowing it to learn when to turn off (i.e., multiply by 0) or amplify specific features: this is called a *gating mechanism*. SwiGLU is very common in modern transformers (see [Chapter 15](#)).

Another GELU-like activation function is *Mish*, which was introduced in a [2019 paper](#) by Diganta Misra.<sup>15</sup> It is defined as  $\text{mish}(z) = z \tanh(\text{softplus}(z))$ , where  $\text{softplus}(z) = \log(1 + \exp(z))$ . Just like GELU and Swish, it is a smooth, nonconvex, and nonmonotonic variant of ReLU, and once again the author ran many experiments and found that Mish generally outperformed other activation functions—even Swish and GELU, by a tiny margin. [Figure 11-4](#) shows GELU, Swish (both with the default  $\beta = 1$  and with  $\beta = 0.6$ ), and lastly Mish. As you can see, Mish overlaps almost perfectly with Swish when  $z$  is negative, and almost perfectly with GELU when  $z$  is positive.

Lastly, in 2021, Google researchers ran an automated architecture search to improve large transformers, and the search found a very simple yet effective activation function: [ReLU<sup>2</sup>](#).<sup>16</sup> As its name suggests, it's simply ReLU squared:  $\text{ReLU}^2(z) = (\max(0, z))^2$ . It has all the qualities of ReLU (simplicity, computational efficiency, sparse output, no saturation on the positive side) but it also has smooth gradients at  $z = 0$ , and it often outperforms other activation functions, especially for sparse models. However, training can be less stable, in part because of its increased sensitivity to outliers and dying ReLUs.



So, which activation function should you use for the hidden layers of your deep neural networks? ReLU remains a good default for most tasks: it's often just as good as the more sophisticated activation functions, plus it's very fast to compute, and many libraries and hardware accelerators provide ReLU-specific optimizations. However, Swish is probably a better default for complex tasks, and you can even try parametrized Swish with a learnable  $\beta$  parameter for the most complex tasks. Mish and SwiGLU may give you slightly better results, but they require a bit more compute. If you care a lot about runtime latency, then you may prefer leaky ReLU, or parametrized leaky ReLU for complex tasks, or even ReLU<sup>2</sup>, especially for sparse models.

PyTorch supports GELU, Mish, and Swish out of the box (using `nn.GELU`, `nn.Mish`, and `nn.SiLU`, respectively). To implement SwiGLU, double the previous linear layer's

---

<sup>15</sup> Diganta Misra, “Mish: A Self Regularized Non-Monotonic Activation Function”, arXiv preprint arXiv:1908.08681 (2019).

<sup>16</sup> So et al., “Primer: Searching for Efficient Transformers for Language Modeling”, arXiv preprint arXiv:2109.08668 (2021).

output dimension, then use `z1, z2 = z.chunk(2, dim=-1)` to split its output in two, and compute `F.silu(beta * z1) * z2` (where `F` is `torch.nn.functional`). For ReLU<sup>2</sup>, simply compute `F.relu(z).square()`. PyTorch also includes simplified and approximated versions of several activation functions, which are much faster to compute and often more stable during training. These simplified versions have names starting with “Hard”, such as `nn.Hardsigmoid`, `nn.Hardtanh`, and `nn.Hardswish`, and they are often used on mobile devices.

That’s all for activation functions! Now, let’s look at a completely different way to solve the unstable gradients problem: batch normalization.

## Batch Normalization

Although using Kaiming initialization along with ReLU (or any of its variants) can significantly reduce the danger of the vanishing/exploding gradients problems at the beginning of training, it doesn’t guarantee that they won’t come back during training.

In a [2015 paper](#),<sup>17</sup> Sergey Ioffe and Christian Szegedy proposed a technique called *batch normalization* (BN) that addresses these problems. The technique consists of adding an operation in the model just before or after the activation function of each hidden layer. This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the operation lets the model learn the optimal scale and mean of each of the layer’s inputs. In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set (no need for `StandardScaler`); the BN layer will do it for you (well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature).

In order to zero-center and normalize the inputs, the algorithm needs to estimate each input’s mean and standard deviation. It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name “batch normalization”). The whole operation is summarized step by step in [Equation 11-4](#).

---

<sup>17</sup> Sergey Ioffe and Christian Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 448–456.

*Equation 11-4. Batch normalization algorithm*

1.  $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2.  $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$
3.  $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
4.  $\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$

In this algorithm:

- $\mu_B$  is the vector of input means, evaluated over the whole mini-batch  $B$  (it contains one mean per input).
- $m_B$  is the number of instances in the mini-batch.
- $\mathbf{x}^{(i)}$  is the input vector of the batch-norm layer for instance  $i$ .
- $\sigma_B$  is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- $\hat{\mathbf{x}}^{(i)}$  is the vector of zero-centered and normalized inputs for instance  $i$ .
- $\epsilon$  is a tiny number that avoids division by zero and ensures the gradients don't grow too large (typically  $10^{-5}$ ). This is called a *smoothing term*.
- $\gamma$  is the output scale parameter vector for the layer (it contains one scale parameter per input).
- $\otimes$  represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- $\beta$  is the output shift (offset) parameter vector for the layer (it contains one shift parameter per input). Each input is offset by its corresponding shift parameter.
- $\mathbf{z}^{(i)}$  is the output of the BN operation. It is a rescaled and shifted version of the inputs.

So during training, BN standardizes its inputs, then rescales and offsets them. Good! What about at test time? Well, it's not that simple. Indeed, we may need to make predictions for individual instances rather than for batches of instances: in this case, we will have no way to compute each input's standard deviation. Moreover, even if we do have a batch of instances, it may be too small, or the instances may not be independent and identically distributed, so computing statistics over the batch instances would be unreliable. One solution is to wait until the end of training, then run the whole training set through the neural network and compute the mean and

standard deviation of each input of the BN layer. These “final” input means and standard deviations can then be used instead of the batch input means and standard deviations when making predictions.

However, most implementations of batch norm estimate these final statistics during training by using a moving average of the layer’s batch input means and variances. This is what PyTorch does automatically when you use its batch-norm layers, such as `nn.BatchNorm1d` (which we will discuss in the next section). To sum up, four parameter vectors are learned in each batch-norm layer:  $\gamma$  (the output scale vector) and  $\beta$  (the output offset vector) are learned through regular backpropagation, and  $\mu$  (the final input mean vector) and  $\sigma^2$  (the final input variance vector) are estimated using an exponential moving average. Note that  $\mu$  and  $\sigma^2$  are estimated during training, but they are used only after training, once you switch the model to evaluation mode using `model.eval()`:  $\mu$  and  $\sigma^2$  then replace  $\mu_B$  and  $\sigma_B^2$  in [Equation 11-4](#).

Ioffe and Szegedy demonstrated that batch norm considerably improved all the deep neural networks they experimented with, leading to a huge improvement in the ImageNet classification task (ImageNet is a large database of images classified into many classes, commonly used to evaluate computer vision systems). The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as tanh and even sigmoid. The networks were also much less sensitive to the weight initialization. The authors were able to use much larger learning rates, significantly speeding up the learning process. Specifically, they note that:

Applied to a state-of-the-art image classification model, batch norm achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. [...] Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.

Finally, like a gift that keeps on giving, batch norm acts like a regularizer, reducing the need for other regularization techniques (such as dropout, described later in this chapter).

Batch normalization does, however, add some complexity to the model (although it can remove the need for normalizing the input data, as discussed earlier). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. Fortunately, it’s often possible to fuse the BN layer with the previous layer after training, thereby avoiding the runtime penalty. This is done by updating the previous layer’s weights and biases so that it directly produces outputs of the appropriate scale and offset. For example, if the previous layer computes  $\mathbf{XW} + \mathbf{b}$ , then the BN layer will compute  $\gamma \otimes (\mathbf{XW} + \mathbf{b} - \mu) / \sigma + \beta$  (ignoring the smoothing term  $\epsilon$  in the denominator). If we define  $\mathbf{W}' = \gamma \otimes \mathbf{W} / \sigma$  and  $\mathbf{b}' = \gamma \otimes (\mathbf{b} - \mu) / \sigma + \beta$ , the equation simplifies to  $\mathbf{XW}' + \mathbf{b}'$ . So, if we replace

the previous layer's weights and biases (**W** and **b**) with the updated weights and biases ( $\mathbf{W}'$  and  $\mathbf{b}'$ ), we can get rid of the BN layer. This is one of the optimizations performed by `optimize_for_inference()` (see [Chapter 10](#)).



You may find that training is rather slow, because each epoch takes much more time when you use batch norm. This is usually counterbalanced by the fact that convergence is much faster with BN, so it will take fewer epochs to reach the same performance. All in all, *wall time* will usually be shorter (this is the time measured by the clock on your wall).

## Implementing batch norm with PyTorch

As with most things with PyTorch, implementing batch norm is straightforward and intuitive. Just add an `nn.BatchNorm1d` layer before or after each hidden layer's activation function, and specify the number of inputs of each BN layer. You may also add a BN layer as the first layer in your model, which removes the need to standardize the inputs manually. For example, let's create a Fashion MNIST image classifier (similar to the one we built in [Chapter 10](#)) using BN as the first layer in the model (after flattening the input images), then again after each hidden layer:

```
model = nn.Sequential(  
    nn.Flatten(),  
    nn.BatchNorm1d(1 * 28 * 28),  
    nn.Linear(1 * 28 * 28, 300),  
    nn.ReLU(),  
    nn.BatchNorm1d(300),  
    nn.Linear(300, 100),  
    nn.ReLU(),  
    nn.BatchNorm1d(100),  
    nn.Linear(100, 10)  
)
```

You can now train the model normally (as you learned in [Chapter 10](#)), and that's it! In this tiny example with just two hidden layers, batch norm is unlikely to have a large impact, but for deeper networks it can make a tremendous difference.



Since batch norm behaves differently during training and during evaluation, it's critical to switch to training mode during training (using `model.train()`), and switch to evaluation mode during evaluation (using `model.eval()`). Forgetting to do so is one of the most common mistakes.

If you look at the parameters of the first BN layer, you will find two: `weight` and `bias`, which correspond to  $\gamma$  and  $\beta$  in [Equation 11-4](#):

```
>>> dict(model[1].named_parameters()).keys()
dict_keys(['weight', 'bias'])
```

And if you look at the buffers of this same BN layer, you will find three: `running_mean`, `running_var`, and `num_batches_tracked`. The first two correspond to the running means  $\mu$  and  $\sigma^2$  discussed earlier, and `num_batches_tracked` simply counts the number of batches seen during training:

```
>>> dict(model[1].named_buffers()).keys()
dict_keys(['running_mean', 'running_var', 'num_batches_tracked'])
```

The authors of the BN paper argued in favor of adding the BN layers before the activation functions, rather than after (as we just did). There is some debate about this, and it seems to depend on the task, so you can experiment with this to see which option works best on your dataset. If you move the BN layers before the activation functions, you can also remove the bias term from the previous `nn.Linear` layers by setting their `bias` hyperparameter to `False`. Indeed, a batch-norm layer already includes one bias term per input. You can also drop the first BN layer to avoid sandwiching the first hidden layer between two BN layers, but this means you should normalize the training set before training. The updated code looks like this:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(1 * 28 * 28, 300, bias=False),
    nn.BatchNorm1d(300),
    nn.ReLU(),
    nn.Linear(300, 100, bias=False),
    nn.BatchNorm1d(100),
    nn.ReLU(),
    nn.Linear(100, 10)
)
```

The `nn.BatchNorm1d` class has a few hyperparameters you can tweak. The defaults will usually be fine, but you may occasionally need to tweak the `momentum`. This hyperparameter is used by the `BatchNorm1d` layer when it updates the exponential moving averages; given a new value  $v$  (i.e., a new vector of input means or variances computed over the current batch), the layer updates the running average  $\hat{v}$  using the following equation:

$$\hat{v} \leftarrow v \times \text{momentum} + \hat{v} \times (1 - \text{momentum})$$

A good momentum value is typically close to 0; for example, 0.01 or 0.001. You want more 0s for smaller mini-batches, and fewer for larger mini-batches. The default is 0.1, which is good for large batch sizes, but not great for small batch sizes such as 32 or 64.



When people talk about “momentum” in the context of a running mean, they usually refer to the weight of the current running mean in the update equation. Sadly, for historical reasons, PyTorch uses the opposite meaning in the BN layers. However, other parts of PyTorch use the conventional meaning (e.g., in optimizers), so don’t get confused.

## Batch norm 1D, 2D, and 3D

In the previous examples, we flattened the input images before sending them through the first `nn.BatchNorm1d` layer. This is because an `nn.BatchNorm1d` layer works on batches of shape `[batch_size, num_features]` (just like the `nn.Linear` layer does), so you would get an error if you moved it before the `nn.Flatten` layer.

However, you could use an `nn.BatchNorm2d` layer before the `nn.Flatten` layer: indeed, it expects its inputs to be image batches of shape `[batch_size, channels, height, width]`, and it computes the batch mean and variance across both the batch dimension (dimension 0) and the spatial dimensions (dimensions 2 and 3). This means that all pixels in the same batch and channel get normalized using the same mean and variance: the `nn.BatchNorm2d` layer only has one weight per channel and one bias per channel (e.g., three weights and three bias terms for color images with three channels for red, green, and blue). This generally works better when dealing with image datasets.

There’s also an `nn.BatchNorm3d` layer which expects batches of shape `[batch_size, channels, depth, height, width]`: this is useful for datasets of 3D images, such as CT scans.

The `nn.BatchNorm1d` layer can also work on batches of sequences. The convention in PyTorch is to represent batches of sequences as 3D tensors of shape `[batch_size, sequence_length, num_features]`. For example, suppose you work on particle physics and you have a dataset of particle trajectories, where each trajectory is composed of a sequence of 100 points in 3D space, then a batch of 32 trajectories will have a shape of `[32, 100, 3]`. However, the `nn.BatchNorm1d` layer expects the shape to be `[batch_size, num_features, sequence_length]`, and it computes the batch mean and variance across the first and last dimensions to get one mean and variance per feature. So you must permute the last two dimensions of the data using `X.permute(0, 2, 1)` before letting it go through the `nn.BatchNorm1d` layer. We will discuss sequences further in [Chapter 13](#).

Batch normalization has become one of the most-used layers in deep neural networks, especially deep convolutional neural networks discussed in [Chapter 12](#), to the point that it is often omitted in the architecture diagrams: it is assumed that BN is added after every layer. That said, it is not perfect. In particular, the computed statistics for an instance are biased by the other samples in a batch, which may reduce

performance (especially for small batch sizes). Moreover, BN struggles with some architectures, such as recurrent nets, as we will see in [Chapter 13](#). For these reasons, batch-norm is more and more often replaced by layer-norm.

## Layer Normalization

Layer normalization (LN) is very similar to batch norm, but instead of normalizing across the batch dimension, LN normalizes across the feature dimensions. This simple idea was introduced by Jimmy Lei Ba et al. in a [2016 paper](#),<sup>18</sup> and initially applied mostly to recurrent nets. However, in recent years it has been successfully applied to many other architectures, such as convolutional nets, transformers, diffusion nets, and more.

One advantage is that LN can compute the required statistics on the fly, at each time step, independently for each instance. This also means that it behaves the same way during training and testing (as opposed to BN), and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set, like BN does. Lastly, LN learns a scale and an offset parameter for each input feature, just like BN does.

PyTorch includes an `nn.LayerNorm` module. To create an instance, you must simply indicate the size of the dimensions that you want to normalize over. These must be the last dimension(s) of the inputs. For example, if the inputs are batches of  $100 \times 200$  RGB images of shape `[3, 100, 200]`, and you want to normalize each image over each of the three color channels separately, you would use the following `nn.LayerNorm` module:

```
inputs = torch.randn(32, 3, 100, 200) # a batch of random RGB images
layer_norm = nn.LayerNorm([100, 200])
result = layer_norm(inputs) # normalizes over the last two dimensions
```

The following code produces the same result:

```
means = inputs.mean(dim=[2, 3], keepdim=True) # shape: [32, 3, 1, 1]
vars_ = inputs.var(dim=[2, 3], keepdim=True, unbiased=False) # shape: same
stds = torch.sqrt(vars_ + layer_norm.eps) # eps is a smoothing term (1e-5)
result = layer_norm.weight * (inputs - means) / stds + layer_norm.bias
# result shape: [32, 3, 100, 200]
```

However, most computer vision architectures that use LN normalize over all channels at once. For this, you must include the size of the channel dimension when creating the `nn.LayerNorm` module:

```
layer_norm = nn.LayerNorm([3, 100, 200])
result = layer_norm(inputs) # normalizes over the last three dimensions
```

---

<sup>18</sup> Jimmy Lei Ba et al., “Layer Normalization”, arXiv preprint arXiv:1607.06450 (2016).

And that's all there is to it! Now let's look at one last technique to stabilize gradients during training: gradient clipping.

## Gradient Clipping

Another technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called *gradient clipping*.<sup>19</sup> This technique is generally used in recurrent neural networks, where using batch norm is tricky (as you will see in [Chapter 13](#)).

In PyTorch, gradient clipping is generally implemented by calling either `torch.nn.utils.clip_grad_norm_()` or `torch.nn.utils.clip_grad_value_()` at each iteration during training, right after the gradients are computed (i.e., after `loss.backward()`). Both functions take as a first argument the list of model parameters whose gradients must be clipped—typically all of them (`model.parameters()`). The `clip_grad_norm_()` function clips each gradient vector's norm if it exceeds the given `max_norm` argument. This is a hyperparameter you can tune (a typical default value is 1.0). The `clip_grad_value_()` function independently clips the individual components of the gradient vector between `-clip_value` and `+clip_value`, where `clip_value` is a hyperparameter you can tune. For example, this training loop clips the norm of each gradient vector to 1.0:

```
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        optimizer.zero_grad()
```

Note that `clip_grad_value_()` will change the orientation of the gradient vector when its components are clipped. For instance, if the original gradient vector is `[0.9, 100.0]`, it points mostly in the direction of the second dimension; but once you clip it by value, you get `[0.9, 1.0]`, which points roughly at the diagonal between the two axes. Despite this reorientation, this approach actually works quite well in practice. If you clipped the same vector by norm, the result would be `[0.00899964, 0.9999595]`: this would preserve the vector's orientation, but almost eliminate the first component. The best clipping function to use depends on the dataset.

---

<sup>19</sup> Razvan Pascanu et al., “On the Difficulty of Training Recurrent Neural Networks”, *Proceedings of the 30th International Conference on Machine Learning* (2013): 1310–1318.

# Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch without first trying to find an existing neural network that accomplishes a similar task to the one you are trying to tackle (I will discuss how to find them in [Chapter 12](#)). If you find such a neural network, then you can generally reuse most of its layers, except for the top ones. This technique is called *transfer learning*. It will not only speed up training considerably, but also require significantly less training data.

Suppose you have access to a DNN that was trained to classify pictures into one hundred different categories, including animals, plants, vehicles, and everyday objects, and you now want to train a DNN to classify specific types of vehicles. These tasks are very similar, even partly overlapping, so you should try to reuse parts of the first network (see [Figure 11-5](#)).

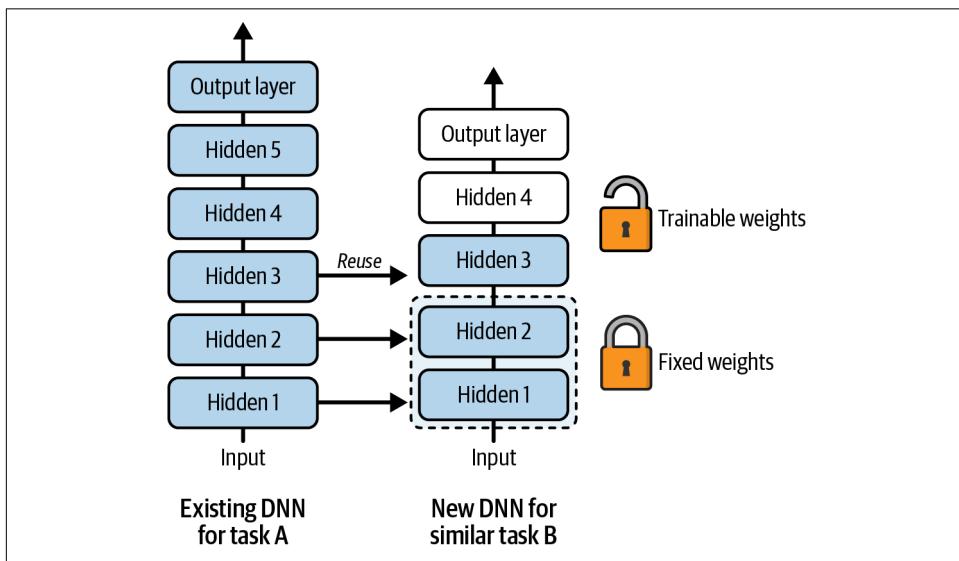


Figure 11-5. Reusing pretrained layers



If the input pictures for your new task don't have the same size as the ones used in the original task, you will usually have to add a preprocessing step to resize them to the size expected by the original model. More generally, transfer learning will work best when the inputs have similar low-level features. For example, a neural net trained on regular pictures taken from mobile phones will help with many other tasks on mobile phone pictures, but it will likely not help at all on satellite images or medical images.

The output layer of the original model should usually be replaced because it is most likely not useful at all for the new task, and it may not even have the right number of outputs.

Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. You want to find the right number of layers to reuse.



The more similar the tasks are, the more layers you will want to reuse (starting with the lower layers). For very similar tasks, try to keep all the hidden layers and just replace the output layer.

Try freezing all the reused layers first (i.e., make their parameters nontrainable by setting `requires_grad` to `False` so that gradient descent won't modify them and they will remain fixed), then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves. The more training data you have, the more layers you can unfreeze. It is also useful to reduce the learning rate when you unfreeze reused layers: this will avoid wrecking their fine-tuned weights.

If you still cannot get good performance, and you have little training data, try dropping the top hidden layer(s) and freezing all the remaining hidden layers again. You can iterate until you find the right number of layers to reuse. If you have plenty of training data, you may try replacing the top hidden layers instead of dropping them, and even adding more hidden layers.

## Transfer Learning with PyTorch

Let's look at an example. Suppose the Fashion MNIST dataset only contained eight classes—for example, all the classes except for Pullover and T-shirt/top. Someone built and trained a PyTorch model on that set and got reasonably good performance (~92% accuracy). Let's call this model A. You now want to tackle a different task: you have images of T-shirts and pullovers, and you want to train a binary classifier: positive for T-shirt/top, negative for Pullover. Your dataset is tiny; you only have 20 labeled images! When you train a new model for this task (let's call it model B) with the same architecture as model A, you get 71.6% test accuracy. While drinking your morning coffee, you realize that your task is quite similar to task A, so perhaps transfer learning can help? Let's find out!

First, let's look at model A:

```
torch.manual_seed(42)

model_A = nn.Sequential(
    nn.Flatten(),
    nn.Linear(1 * 28 * 28, 100),
    nn.ReLU(),
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.Linear(100, 8)
)
[...] # train this model or load pretrained weights
```

We can now reuse the layers we want, for example, all layers except for the output layer:

```
import copy

torch.manual_seed(42)
reused_layers = copy.deepcopy(model_A[:-1])
model_B_on_A = nn.Sequential(
    *reused_layers,
    nn.Linear(100, 1) # new output layer for task B
).to(device)
```

In this code, we use Python's `copy.deepcopy()` function to copy all the modules in the `nn.Sequential` module (along with all their data and submodules), except for the last layer. Since we're making a deep copy, all the submodules are copied as well. Then we create `model_B_on_A`, which is an `nn.Sequential` model based on the reused layers of model A, plus a new output layer for task B: it has a single output since task B is binary classification.

You could start training `model_B_on_A` for task B now, but since the new output layer was initialized randomly, it will make large errors (at least during the first few epochs), so there will be large error gradients that may wreck the reused weights. To avoid this, one approach is to freeze the reused layers during the first few epochs, giving the new layer some time to learn reasonable weights:

```
for layer in model_B_on_A[:-1]:
    for param in layer.parameters():
        param.requires_grad = False
```

Now you can train `model_B_on_A`. But don't forget that task B is binary classification, so you must switch the loss to `nn.BCEWithLogitsLoss` (or to `nn.BCELoss` if you prefer to add an `nn.Sigmoid` activation function on the output layer), as we discussed in [Chapter 10](#). Also, if you are using `torchmetrics`, make sure to set `task="binary"` when creating the `Accuracy` metric:

```
xentropy = nn.BCEWithLogitsLoss()
accuracy = torchmetrics.Accuracy(task="binary").to(device)
[...] # train model_B_on_A
```

After you have trained the model for a few epochs, you can unfreeze the reused layers (setting `param.requires_grad = True` for all parameters), reduce the learning rate, and continue training to fine-tune the reused layers for task B.

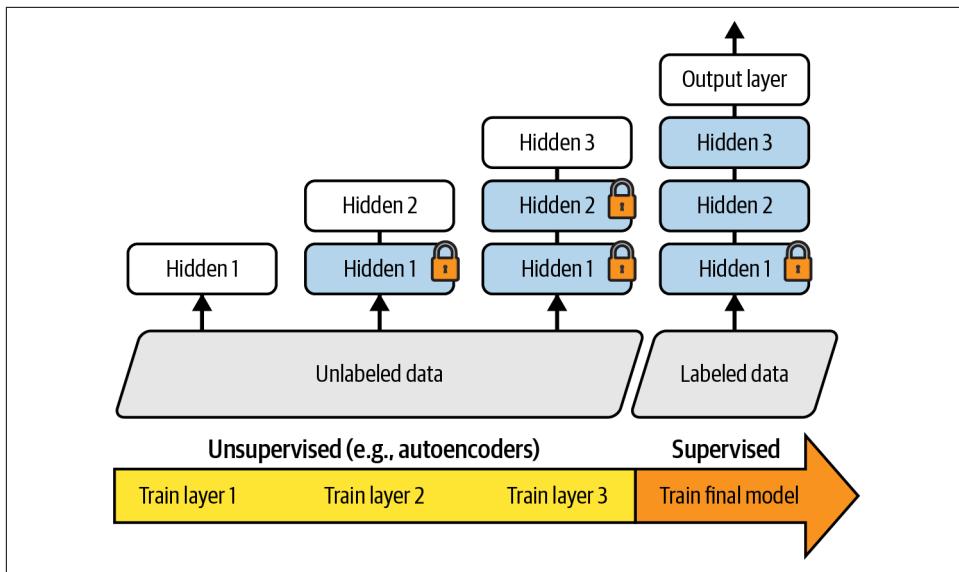
So, what's the final verdict? Well, this model's test accuracy is 92.5%, which is much better than the 71.6% accuracy we reached without pretraining!

Are you convinced? Well, you shouldn't be; I cheated! I tried many configurations until I found one that demonstrated a strong improvement. If you try to change the classes or the random seed, you will see that the improvement generally drops, or even vanishes or reverses. What I did is called "torturing the data until it confesses". When a paper looks too positive, you should be suspicious. Perhaps the flashy new technique does not actually help much (in fact, it may even degrade performance), but the authors tried many variants and reported only the best results—which may be due to sheer luck—without mentioning how many failures they encountered along the way. That's called *p-hacking*. Most of the time, this is not malicious, but it is part of the reason why so many results in science can never be reproduced.

But why did I cheat? It turns out that transfer learning does not work very well with small dense networks, presumably because small networks learn few patterns, and dense networks learn very specific patterns, which are unlikely to be useful for other tasks. Transfer learning works best with deep convolutional neural networks and with Transformer architectures. We will revisit transfer learning in Chapters 12 and 15, using the techniques we just discussed (and this time it will work fine without cheating, I promise!).

## Unsupervised Pretraining

Suppose you want to tackle a complex task for which you don't have much labeled training data, but unfortunately you cannot find a model trained on a similar task. Don't lose hope! First, you should try to gather more labeled training data, but if you can't, you may still be able to perform *unsupervised pretraining* (see [Figure 11-6](#)). Indeed, it is often cheap to gather unlabeled training examples, but expensive to label them. If you can gather plenty of unlabeled training data, you can try to use it to train an unsupervised model, such as an autoencoder (see [Chapter 18](#)). Then you can reuse the lower layers of the autoencoder, add the output layer for your task on top, and fine-tune the final network using supervised learning (i.e., with the labeled training examples).



*Figure 11-6. Greedy layer-wise pretraining used in the early days of deep learning; nowadays the unsupervised part is typically done in one shot on all the data rather than one layer at a time*

It is this technique that Geoffrey Hinton and his team used in 2006, and which led to the revival of neural networks and the success of deep learning. Until 2010, unsupervised pretraining—typically with restricted Boltzmann machines (RBMs; see the notebook at <https://homl.info/extr-a-anns>)—was the norm for deep nets, and only after the vanishing gradients problem was alleviated did it become much more common to train DNNs purely using supervised learning. Unsupervised pretraining (today typically using autoencoders or diffusion models rather than RBMs) is still a good option when you have a complex task to solve, no similar model you can reuse, and little labeled training data, but plenty of unlabeled training data.

Note that in the early days of deep learning it was difficult to train deep models, so people would use a technique called *greedy layer-wise pretraining* (depicted in [Figure 11-6](#)). They would first train an unsupervised model with a single layer, typically an RBM, then they would freeze that layer and add another one on top of it, then train the model again (effectively just training the new layer), then freeze the new layer and add another layer on top of it, train the model again, and so on. Nowadays, things are much simpler: people generally train the full unsupervised model in one shot and use models such as autoencoders or diffusion models rather than RBMs.

## Pretraining on an Auxiliary Task

If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

For example, if you want to build a system to recognize faces, you may only have a few pictures of each individual—clearly not enough to train a good classifier. Gathering hundreds of pictures of each person would not be practical. You could, however, use a public dataset containing millions of pictures of people (such as VGGFace2) and train a first neural network to detect whether two different pictures feature the same person. Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier that uses little training data.



You could also just scrape pictures of random people from the web, but this would probably be illegal. Firstly, photos are usually copyrighted by their creators, and websites like Instagram or Facebook enforce these copyright protections through their terms of service, which prohibit scraping and unauthorized use. Secondly, over 40 countries require explicit consent for collecting and processing personal data, including facial images.

For natural language processing (NLP) applications, you can download a corpus of millions of text documents and automatically generate labeled data from it. For example, you could randomly mask out some words and train a model to predict what the missing words are (e.g., it should predict that the missing word in the sentence “What \_\_\_ you saying?” is probably “are” or “were”). If you can train a model to reach good performance on this task, then it will already know quite a lot about language, and you can certainly reuse it for your actual task and fine-tune it on your labeled data (this is basically how large language models are trained and fine-tuned, as we will see in [Chapter 15](#)).



*Self-supervised learning* is when you automatically generate the labels from the data itself, as in the text-masking example, then you train a model on the resulting “labeled” dataset using supervised learning techniques.

# Faster Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using batch-norm or layer-norm, and reusing parts of a pretrained network (possibly built for an auxiliary task or using unsupervised learning). Another huge speed boost comes from using a faster optimizer than the regular gradient descent optimizer. In this section we will present the most popular optimization algorithms: momentum, Nesterov accelerated gradient, AdaGrad, RMSProp, and finally, Adam and its variants.

## Momentum

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the core idea behind *momentum optimization*, proposed by Boris Polyak in 1964.<sup>20</sup> In contrast, regular gradient descent will take small steps when the slope is gentle and big steps when the slope is steep, but it will never pick up speed. As a result, regular gradient descent is generally much slower to reach the minimum than momentum optimization.

As we saw in [Chapter 4](#), gradient descent updates the weights  $\theta$  by directly subtracting the gradient of the cost function  $J(\theta)$  with regard to the weights ( $\nabla_{\theta}J(\theta)$ ) multiplied by the learning rate  $\eta$ . The equation is  $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta)$ . It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the *momentum vector*  $m$  (multiplied by the learning rate  $\eta$ ), and it updates the weights by adding this momentum vector (see [Equation 11-5](#)). In other words, the gradient is used as a force learning to an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter  $\beta$ , called the *momentum coefficient*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

---

<sup>20</sup> Boris T. Polyak, “Some Methods of Speeding Up the Convergence of Iteration Methods”, *USSR Computational Mathematics and Mathematical Physics* 4, no. 5 (1964): 1–17.

### Equation 11-5. Momentum algorithm

1.  $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta)$
2.  $\theta \leftarrow \theta + \mathbf{m}$

You can verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate  $\eta$  multiplied by  $1 / (1 - \beta)$  (ignoring the sign). For example, if  $\beta = 0.9$ , then the terminal velocity is equal to 10 times the gradient times the learning rate, so momentum optimization ends up going 10 times faster than gradient descent! In practice, the gradients are not constant, so the speedup is not always as dramatic, but momentum optimization can escape from plateaus much faster than regular gradient descent. We saw in [Chapter 4](#) that when the inputs have very different scales, the cost function will look like an elongated bowl (see [Figure 4-7](#)). Gradient descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use batch-norm or layer-norm, the upper layers will often end up having inputs with very different scales, so using momentum optimization helps a lot. It can also help roll past local optima.



Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it's good to have a bit of friction in the system: it reduces these oscillations and thus speeds up convergence.

Implementing momentum optimization in PyTorch is a no-brainer: just use the SGD optimizer and set its `momentum` hyperparameter, then sit back and profit!

```
optimizer = torch.optim.SGD(model.parameters(), momentum=0.9, lr=0.05)
```

The one drawback of momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than regular gradient descent.

## Nesterov Accelerated Gradient

One small variant to momentum optimization, proposed by [Yuriii Nesterov in 1983](#),<sup>21</sup> is almost always faster than regular momentum optimization. The *Nesterov accelerated gradient* (NAG) method, also known as *Nesterov momentum optimization*,

---

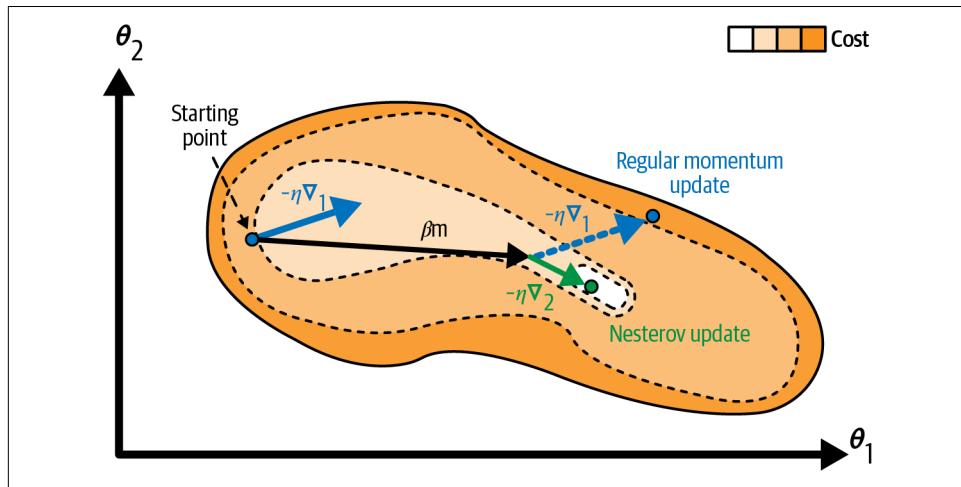
<sup>21</sup> Yuriii Nesterov, "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence  $O(1/k^2)$ ", *Doklady AN USSR* 269 (1983): 543–547.

measures the gradient of the cost function not at the local position  $\theta$  but slightly ahead in the direction of the momentum, at  $\theta + \beta\mathbf{m}$  (see [Equation 11-6](#)).

*Equation 11-6. Nesterov accelerated gradient algorithm*

1.  $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta + \beta\mathbf{m})$
2.  $\theta \leftarrow \theta + \mathbf{m}$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original position, as you can see in [Figure 11-7](#) (where  $\nabla_1$  represents the gradient of the cost function measured at the starting point  $\theta$ , and  $\nabla_2$  represents the gradient at the point located at  $\theta + \beta\mathbf{m}$ ).



*Figure 11-7. Regular versus Nesterov momentum optimization: the former applies the gradients computed before the momentum step, while the latter applies the gradients computed after*

As you can see, the Nesterov update ends up closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular momentum optimization. Moreover, note that when the momentum pushes the weights across a valley,  $\nabla_1$  continues to push farther across the valley, while  $\nabla_2$  pushes back toward the bottom of the valley. This helps reduce oscillations and thus NAG converges faster.

To use NAG, simply set `nesterov=True` when creating the SGD optimizer:

```
optimizer = torch.optim.SGD(model.parameters(),
                           momentum=0.9, nesterov=True, lr=0.05)
```

## AdaGrad

Consider the elongated bowl problem again: gradient descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley. It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum. The *AdaGrad* algorithm<sup>22</sup> achieves this correction by scaling down the gradient vector along the steepest dimensions (see [Equation 11-7](#)).

*Equation 11-7. AdaGrad algorithm*

1.  $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2.  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

The first step accumulates the square of the gradients into the vector  $\mathbf{s}$  (recall that the  $\otimes$  symbol represents the element-wise multiplication). This vectorized form is equivalent to computing  $s_i \leftarrow s_i + (\partial J(\boldsymbol{\theta}) / \partial \theta_i)^2$  for each element  $s_i$  of the vector  $\mathbf{s}$ ; in other words, each  $s_i$  accumulates the squares of the partial derivative of the cost function with regard to parameter  $\theta_i$ . If the cost function is steep along the  $i^{\text{th}}$  dimension, then  $s_i$  will get larger and larger at each iteration.

The second step is almost identical to gradient descent, but with one big difference: the gradient vector is scaled down by a factor of  $\sqrt{\mathbf{s} + \varepsilon}$  (the  $\oslash$  symbol represents the element-wise division, the square root is also computed element-wise, and  $\varepsilon$  is a smoothing term to avoid division by zero, typically set to  $10^{-10}$ ). This vectorized form is equivalent to simultaneously computing  $\theta_i \leftarrow \theta_i - \eta \partial J(\boldsymbol{\theta}) / \partial \theta_i / \sqrt{s_i + \varepsilon}$  for all parameters  $\theta_i$ .

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see [Figure 11-8](#)). One additional benefit is that it requires much less tuning of the learning rate hyperparameter  $\eta$ .

AdaGrad frequently performs well for simple quadratic problems, but it often stops too early when training neural networks: the learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum.

---

<sup>22</sup> John Duchi et al., “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, *Journal of Machine Learning Research* 12 (2011): 2121–2159.

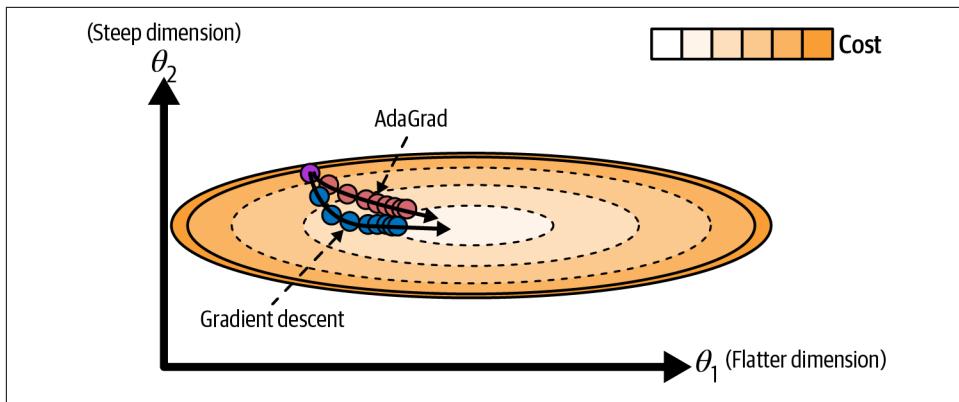


Figure 11-8. AdaGrad versus gradient descent: the former can correct its direction earlier to point to the optimum

So even though PyTorch has an `Adagrad` optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as linear regression, though). Still, understanding AdaGrad is helpful to comprehend the other adaptive learning rate optimizers.

## RMSProp

As we've seen, AdaGrad runs the risk of slowing down a bit too fast and never converging to the global optimum. The *RMSProp* algorithm<sup>23</sup> fixes this by accumulating only the gradients from the most recent iterations, as opposed to all the gradients since the beginning of training. It does so by using exponential decay in the first step (see [Equation 11-8](#)).

*Equation 11-8. RMSProp algorithm*

1.  $s \leftarrow \alpha s + (1 - \alpha) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2.  $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$

The decay rate  $\alpha$  is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

---

<sup>23</sup> This algorithm was created by Geoffrey Hinton and Tijmen Tieleman in 2012 and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: <https://hml.info/57>, video: <https://hml.info/58>). Amusingly, since the authors did not write a paper to describe the algorithm, researchers often cite “slide 29 in lecture 6e” in their papers.

As you might expect, PyTorch has an RMSprop optimizer:

```
optimizer = torch.optim.RMSprop(model.parameters(), alpha=0.9, lr=0.05)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

## Adam

*Adam*,<sup>24</sup> which stands for *adaptive moment estimation*, combines the ideas of momentum optimization and RMSProp: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients (see [Equation 11-9](#)). These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment*, while the variance is often called the *second moment*, hence the name of the algorithm.

*Equation 11-9. Adam algorithm*

1.  $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\Theta} J(\Theta)$
2.  $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\Theta} J(\Theta) \otimes \nabla_{\Theta} J(\Theta)$
3.  $\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4.  $\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5.  $\Theta \leftarrow \Theta + \eta \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \varepsilon}$

In this equation,  $t$  represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both momentum optimization and RMSProp:  $\beta_1$  corresponds to  $\beta$  in momentum optimization, and  $\beta_2$  corresponds to  $\alpha$  in RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just  $1 - \beta_1$  times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since  $\mathbf{m}$  and  $\mathbf{s}$  are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost  $\mathbf{m}$  and  $\mathbf{s}$  at the beginning of training.

---

<sup>24</sup> Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", arXiv preprint arXiv:1412.6980 (2014).

The momentum decay hyperparameter  $\beta_1$  is typically initialized to 0.9, while the scaling decay hyperparameter  $\beta_2$  is often initialized to 0.999. As earlier, the smoothing term  $\epsilon$  is usually initialized to a tiny number such as  $10^{-8}$ . These are the default values for the Adam class. Here is how to create an Adam optimizer using PyTorch:

```
optimizer = torch.optim.Adam(model.parameters(), betas=(0.9, 0.999), lr=0.05)
```

Since Adam is an adaptive learning rate algorithm, like AdaGrad and RMSProp, it requires less tuning of the learning rate hyperparameter  $\eta$ . You can often use the default value  $\eta = 0.001$ , making Adam even easier to use than gradient descent.



If you are starting to feel overwhelmed by all these different techniques and are wondering how to choose the right ones for your task, don't worry: some practical guidelines are provided at the end of this chapter.

Finally, three variants of Adam are worth mentioning: AdaMax, NAdam, and AdamW.

## AdaMax

The Adam paper also introduced AdaMax. Notice that in step 2 of [Equation 11-9](#), Adam accumulates the squares of the gradients in  $s$  (with a greater weight for more recent gradients). In step 5, if we ignore  $\epsilon$  and steps 3 and 4 (which are technical details anyway), Adam scales down the parameter updates by the square root of  $s$ . In short, Adam scales down the parameter updates by the  $\ell_2$  norm of the time-decayed gradients (recall that the  $\ell_2$  norm is the square root of the sum of squares).

AdaMax replaces the  $\ell_2$  norm with the  $\ell_\infty$  norm (a fancy way of saying the max). Specifically, it replaces step 2 in [Equation 11-9](#) with  $s \leftarrow \max(\beta_2 s, \text{abs}(\nabla_{\theta} J(\theta)))$ , it drops step 4, and in step 5 it scales down the gradient updates by a factor of  $s$ , which is the max of the absolute value of the time-decayed gradients.

In practice, this can make AdaMax more stable than Adam, but it really depends on the dataset, and in general Adam performs better. So, this is just one more optimizer you can try if you experience problems with Adam on some task.

## NAdam

NAdam optimization is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In his report introducing this technique,<sup>25</sup> the researcher Timothy Dozat compares many different optimizers on various tasks and

---

<sup>25</sup> Timothy Dozat, “Incorporating Nesterov Momentum into Adam”, (2016).

finds that NAdam generally outperforms Adam but is sometimes outperformed by RMSProp.

## AdamW

AdamW<sup>26</sup> is a variant of Adam that integrates a regularization technique called *weight decay*. Weight decay reduces the size of the model’s weights at each training iteration by multiplying them by a decay factor such as 0.99. This may remind you of  $\ell_2$  regularization (introduced in [Chapter 4](#)), which also aims to keep the weights small, and indeed it can be shown mathematically that  $\ell_2$  regularization is equivalent to weight decay when using SGD. However, when using Adam or its variants,  $\ell_2$  regularization and weight decay are *not* equivalent: in practice, combining Adam with  $\ell_2$  regularization results in models that often don’t generalize as well as those produced by SGD. AdamW fixes this issue by properly combining Adam with weight decay.



Adaptive optimization methods (including RMSProp, Adam, AdaMax, NAdam, and AdamW optimization) are often great, converging fast to a good solution. However, a [2017 paper](#)<sup>27</sup> by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model’s performance, try using NAG instead: your dataset may just be allergic to adaptive gradients.

To use NAdam, AdaMax, or AdamW in PyTorch, replace `torch.optim.Adam` with `torch.optim.NAdam`, `torch.optim.Adamax`, or `torch.optim.AdamW`. For AdamW, you probably want to tune the `weight_decay` hyperparameter.

All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*, which measure the slope of the loss function along each axis). The optimization literature also contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*, which are the partial derivatives of the Jacobians, measuring how each Jacobian changes along each axis; in other words, measuring the loss function’s curvature).

Unfortunately, these Hessian-based algorithms are hard to apply directly to deep neural networks because there are  $n^2$  second-order derivatives per output (where  $n$  is the number of parameters), as opposed to just  $n$  first-order derivatives per

---

<sup>26</sup> Ilya Loshchilov, and Frank Hutter, “Decoupled Weight Decay Regularization”, arXiv preprint arXiv:1711.05101 (2017).

<sup>27</sup> Ashia C. Wilson et al., “The Marginal Value of Adaptive Gradient Methods in Machine Learning”, *Advances in Neural Information Processing Systems* 30 (2017): 4148–4158.

output. Since DNNs typically have hundreds of thousands of parameters or more, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the *Hessian matrix* is just too slow.<sup>28</sup>

Luckily, it is possible to use stochastic methods that can efficiently approximate second-order information. One such algorithm is Shampoo,<sup>29</sup> which uses accumulated gradient information to approximate the second-order terms, similar to how Adam accumulates first-order statistics. It is not included in the PyTorch library, but you can get it in the PyTorch-Optimizer library (`pip install torch_optimizer`).

## Training Sparse Models

All the optimization algorithms we just discussed produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to zero) using `torch.nn.utils.prune.l1_unstructured()`. Or you can get rid of entire neurons, channels, or layers, not just individual weights, using `torch.nn.utils.prune.ln_structured()`, or other functions in the `torch.nn.utils.prune` package.

You should generally also apply fairly strong sparsity inducing regularization during training, such as  $\ell_1$  regularization (you'll see how later in this chapter), since it pushes the optimizer to zero out as many weights as it can (see “[Lasso Regression](#)” on [page 162](#)). You can also try scaling down random weights during initialization to encourage sparsity.

**Table 11-2** compares all the optimizers we've discussed so far (★ is bad, ★★ is average, and ★★★ is good).

<sup>28</sup> The *Jacobian matrix* contains all the first-order partial derivatives of a function with multiple parameters and multiple outputs: one column per parameter, and one row per output. When training a neural net with gradient descent, there's a single output—the loss—so the matrix contains a single row, and there's one column per model parameter, so it's a  $1 \times n$  matrix. The *Hessian matrix* contains all the second-order derivatives of a single-output function with multiple parameters: for each model parameter it contains one row and one column, so it's an  $n \times n$  matrix. The informal names *Jacobians* and *Hessians* refer to the elements of these matrices.

<sup>29</sup> V. Gupta et al., “[Shampoo: Preconditioned Stochastic Tensor Optimization](#)”, arXiv preprint arXiv:1802.09568 (2018).

Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
SGD	★	★★★
SGD(momentum=...)	★★	★★★
SGD(momentum=..., nesterov=True)	★★★	★★★
Adagrad	★★	★ (stops too early)
RMSprop	★★	★★ or ★★★
Adam	★★	★★ or ★★★
AdaMax	★★	★★ or ★★★
NAdam	★★	★★ or ★★★
AdamW	★★	★★ or ★★★

## Learning Rate Scheduling

Finding a good learning rate is very important. If you set it too high, training will diverge (as discussed in “[Gradient Descent](#)” on page 142). If you set it too low, then training will be painfully slow, and it may also get stuck in a local optimum and produce a suboptimal model. If you set the learning rate fairly high (but not high enough to diverge), then training will often make rapid progress at first, but it will end up dancing around the optimum toward the end of training and thereby produce a suboptimal model. If you find a really good learning rate, you can end up with an excellent model, but training will generally be a bit too slow. Luckily, you can do better than a constant learning rate. In particular, it’s a good idea to start with a fairly high learning rate and then reduce it toward the end of training (or whenever progress stops): this ensures that training starts fast, while also allowing backprop to settle down toward the end to really fine-tune the model parameters (see [Figure 11-9](#)).

There are various other strategies to tweak the learning rate during training. These are called *learning schedules* (I briefly introduced this concept in [Chapter 4](#)). The `torch.optim.lr_scheduler` module provides several implementations of common learning schedules. Let’s look at the most important ones, starting with exponential scheduling.

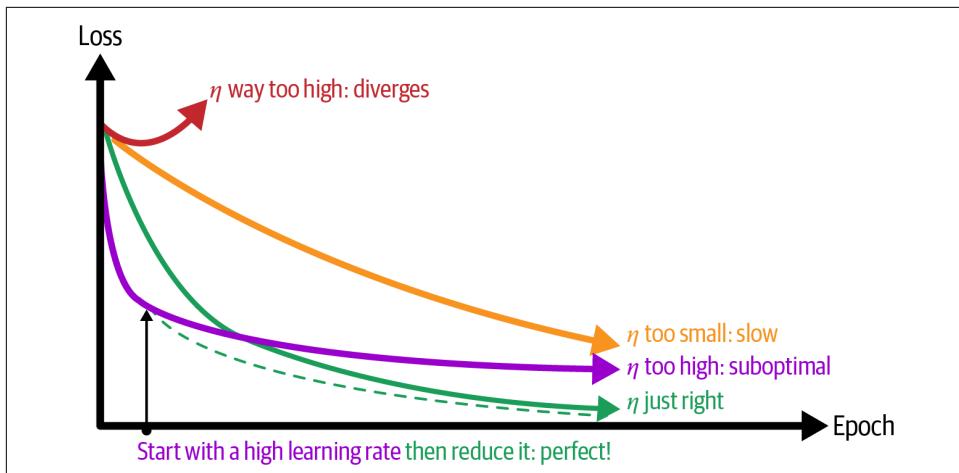


Figure 11-9. Learning curves for various learning rates  $\eta$

## Exponential Scheduling

The `ExponentialLR` class implements *exponential scheduling*, whereby the learning rate is multiplied by a constant factor `gamma` at some regular interval, typically at every epoch. As a result, after the  $n^{\text{th}}$  epoch, the learning rate will be equal to the initial learning rate times `gamma` to the power of  $n$ . This factor `gamma` is yet another hyperparameter you can tune. In general, you will want to set `gamma` to a value lower than 1, but fairly close to 1 to avoid decreasing the learning rate too fast. For example, if `gamma` is set to 0.9, then after 10 epochs the learning rate will be about 35% of the initial learning rate, and after 20 epochs it will be about 12%.

The `ExponentialLR` constructor expects at least two arguments—the optimizer whose learning rate will be tweaked during training, and the factor `gamma`:

```
model = [...] # build the model
optimizer = torch.optim.SGD(model.parameters(), lr=0.05) # or any other optim.
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.9)
```

Next, you must update the training loop to call `scheduler.step()` at the end of each epoch to tweak the optimizer’s learning rate:

```
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        [...]
        # the rest of the training loop remains unchanged

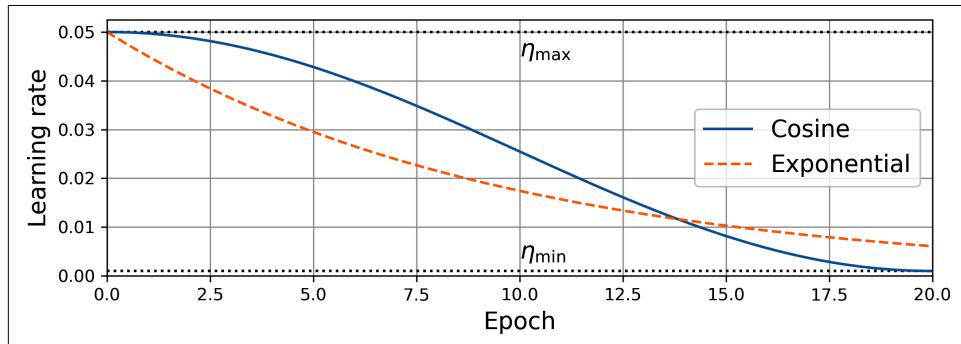
    scheduler.step()
```



If you interrupt training and you later want to resume it where you left off, you should set the `last_epoch` argument of the scheduler's constructor to the last epoch you ran (zero-indexed). The default is `-1`, which makes the scheduler start training from scratch.

## Cosine Annealing

Instead of decreasing the learning rate exponentially, you can use the cosine function to go from the maximum learning rate  $\eta_{\max}$  at the start of training, down to the minimum learning rate  $\eta_{\min}$  at the end. This is called *cosine annealing*. Compared to exponential scheduling, cosine annealing ensures that the learning rate remains fairly high during most of training, while getting closer to the minimum near the end (see [Figure 11-10](#)). All in all, cosine annealing generally performs better. The learning rate at epoch  $t$  (zero-indexed) is given by [Equation 11-10](#), where  $T_{\max}$  is the maximum number of epochs.



*Figure 11-10. Cosine annealing learning schedule*

*Equation 11-10. Cosine annealing equation*

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left( 1 + \cos \left( \frac{t}{T_{\max}} \pi \right) \right)$$

PyTorch includes the `CosineAnnealingLR` scheduler, which you can create as follows (`T_max` is  $T_{\max}$  and `eta_min` is  $\eta_{\min}$ ). You can then use it just like the `ExponentialLR` scheduler:

```
cosine_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(  
    optimizer, T_max=20, eta_min=0.001)
```

One problem with cosine annealing is that you have to set two new hyperparameters,  $T_{\max}$  and  $\eta_{\min}$ , and it's not easy to know in advance how many epochs to train and when to stop decreasing the learning rate. This is why I generally prefer to use the performance scheduling technique.

## Performance Scheduling

*Performance scheduling*, also called *adaptive scheduling*, is implemented by PyTorch’s ReduceLROnPlateau scheduler: it keeps track of a given metric during training—typically the validation loss—and if this metric stops improving for some time, it multiplies the learning rate by some factor. This scheduler has quite a few hyperparameters, but the default values work well for most of them. You may occasionally need to tweak the following (see the documentation for information on the other hyperparameters):

### mode

If the tracked metric must be maximized (such as the validation accuracy), then you must set the mode to ‘max’. The default is ‘min’, which is fine if the tracked metric must be minimized (such as the validation loss).

### patience

The number of consecutive steps (typically epochs) to wait for improvement in the monitored metric before reducing the learning rate. It defaults to 10, which is generally fine. If each epoch is very long, then you may want to reduce this value.

### factor

The factor by which the learning rate will be multiplied whenever the monitored metric fails to improve for too long. It defaults to 0.1, again a reasonable default, but perhaps a bit small in some cases.

For example, let’s implement performance scheduling based on the validation accuracy (i.e., which we want to maximize):

```
[...] # build the model and optimizer
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode="max", patience=2, factor=0.1)
```

The training loop needs to be tweaked again because we must evaluate the desired metric at each epoch (in this example, we are using the `evaluate_tm()` function that we defined in [Chapter 10](#)), and we must then pass the result to the scheduler’s `step()` method:

```
metric = torchmetrics.Accuracy(task="multiclass", num_classes=10).to(device)
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        [...] # the rest of the training loop remains unchanged
        val_metric = evaluate_tm(model, valid_loader, metric).item()
        scheduler.step(val_metric)
```

## Warming Up the Learning Rate

So far, we have always started training with the maximum learning rate. However, this can sometimes cause gradient descent to bounce around randomly at the beginning

of training, neither exploding nor making any significant progress. This typically happens with sensitive models, such as recurrent neural networks (Chapter 13), or when using a very large batch size. In such cases, one solution is to “warm up” the learning rate, starting close to zero and gradually increasing the learning rate over a few epochs, up to the maximum learning rate. During this warm-up phase, gradient descent has time to stabilize into a better region of the loss landscape, where it can then make quick progress using a high learning rate.

Why does this work? Well, the loss landscape sometimes resembles the Himalayas: it's very high up and full of gigantic spikes. If you start with a high learning rate, you might jump from one mountain peak to the next for a very long time. If instead you start with a small learning rate, you will just walk down the mountain and valleys and escape the spiky mountain range altogether until you reach flatter lands. From then on, you can use a large learning rate for the rest of your journey, slowing down only toward the end.

A common way to implement learning rate warm up using PyTorch is to use a `LinearLR` scheduler to increase the learning rate linearly over a few epochs. For example, the following scheduler will increase the learning rate from 10% to 100% of the optimizer's original learning rate over 3 epochs (i.e., 10% during the first epoch, 40% during the second epoch, 70% during the third epoch, and 100% after that):

```
warmup_scheduler = torch.optim.lr_scheduler.LinearLR(  
    optimizer, start_factor=0.1, end_factor=1.0, total_iters=3)
```

If you would like more flexibility, you can write your own custom function and wrap it in a `LambdaLR` scheduler. For example, the following scheduler is equivalent to the `LinearLR` scheduler we just defined:

```
warmup_scheduler = torch.optim.lr_scheduler.LambdaLR(  
    optimizer,  
    lambda epoch: (min(epoch, 3) / 3) * (1.0 - 0.1) + 0.1)
```

You must then insert `warmup_scheduler.step()` at the beginning of each epoch, and make sure you deactivate the scheduler(s) you are using for the rest of training during the warm-up phase. And that's all!

```
for epoch in range(n_epochs):  
    warmup_scheduler.step()  
    for X_batch, y_batch in train_loader:  
        [...] # the rest of the training loop is unchanged  
        if epoch >= 3: # deactivate other scheduler(s) during warmup  
            scheduler.step(val_metric)
```

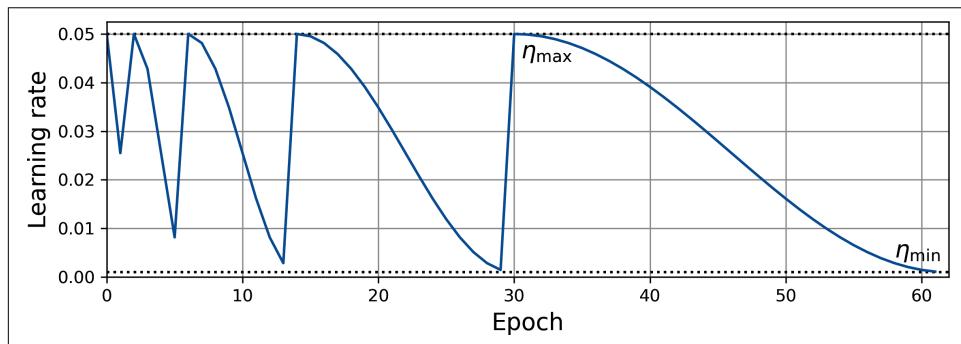
In short, you pretty much always want to cool down the learning rate at the end of training, and you may also want to warm it up at the beginning if gradient descent needs a bit of help getting started. But are there any cases where you may want to tweak the learning rate in the middle of training? Well yes, there are; for example, if

gradient descent gets stuck in a local optimum or a high plateau. Gradient descent could remain stuck here for a long time, or even forever. Luckily, there's a way to escape this trap: just increase the learning rate for a little while.

You could spend your time staring at the learning curves during training, and manually interrupting it to tweak the learning rate when needed, but you probably have better things to do. Alternatively, you could implement a custom scheduler that monitors the validation metric—much like the `ReduceLROnPlateau` scheduler—and increases the learning rate for a while if the validation metric is stuck in a bad plateau. For this, you could subclass the `LRScheduler` base class. This is beyond the scope of this book, but you can take inspiration from the `ReduceLROnPlateau` scheduler's source code (and get a little bit of help from your favorite AI assistant). But a much simpler option is to use the cosine annealing with warm restarts learning schedule. Let's look at it now.

## Cosine Annealing with Warm Restarts

*Cosine annealing with warm restarts* was introduced in a [2016 paper](#) by Ilya Loshchilov and Frank Hutter.<sup>30</sup> This schedule just repeats the cosine annealing schedule over and over again. Since the learning rate regularly shoots back up, this schedule allows gradient descent to escape local optima and plateaus automatically. The authors recommend starting with a fairly short round of cosine annealing, but then doubling  $T_{\max}$  after each round (see [Figure 11-11](#)). This allows gradient descent to do a lot of quick explorations at the start of training, while also taking the time to properly optimize the model later during training, possibly escaping a plateau or two along the way.



*Figure 11-11. Cosine annealing with warm restarts*

<sup>30</sup> Ilya Loshchilov and Frank Hutter, “SGDR: Stochastic Gradient Descent With Warm Restarts”, arXiv preprint arXiv:1608.03983 (2016).

Conveniently, PyTorch includes a `CosineAnnealingWarmRestarts` scheduler. You must set `T_0`, which is the value of  $T_{\max}$  for the first round of cosine annealing. You may also set `T_mult` to 2 if you want to double  $T_{\max}$  at each round (the default is 1, meaning  $T_{\max}$  stays constant and all rounds have the same length). Finally, you can set `eta_min` (it defaults to 0):

```
cosine_repeat_scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(  
    optimizer, T_0=2, T_mult=2, eta_min=0.001)
```

## 1cycle Scheduling

Yet another popular learning schedule is *1cycle*, introduced in a [2018 paper](#) by Leslie Smith.<sup>31</sup> It starts by warming up the learning rate, starting at  $\eta_0$  and growing linearly up to  $\eta_1$  halfway through training. Then it decreases the learning rate linearly down to  $\eta_0$  again during the second half of training, finishing the last few epochs by dropping the rate down by several orders of magnitude (still linearly). The maximum learning rate  $\eta_1$  is chosen using the same approach we used to find the optimal learning rate, and the initial learning rate  $\eta_0$  is usually 10 times lower. When using a momentum, we start with a high momentum first (e.g., 0.95), then drop it down to a lower momentum during the first half of training (e.g., down to 0.85, linearly), and then bring it back up to the maximum value (e.g., 0.95) during the second half of training, finishing the last few epochs with that maximum value. Smith did many experiments showing that this approach was often able to speed up training considerably and reach better performance. For example, on the popular CIFAR10 image dataset, this approach reached 91.9% validation accuracy in just 100 epochs, compared to 90.3% accuracy in 800 epochs through a standard approach (using the same neural network architecture). This feat was dubbed *super-convergence*. PyTorch implements this schedule in the `OneCycleLR` scheduler.



If you are not sure which learning schedule to use, 1cycle can be a good default, but I tend to have more luck with performance scheduling. If you run into instabilities at the start of training, try adding learning rate warm-up. And if training gets stuck on plateaus, try cosine annealing with warm restarts.

We have now covered the most popular learning schedules, but PyTorch offers a few extra schedulers (e.g., a polynomial scheduler, a cyclic scheduler, a scheduler that makes it easy to chain other schedulers, and a few more), so make sure to check out the documentation.

---

<sup>31</sup> Leslie N. Smith, “A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay”, arXiv preprint arXiv:1803.09820 (2018).

Now let's move on to one final topic before we complete this chapter on deep learning training techniques: regularization. Deep learning is highly prone to overfitting, so regularization is key!

## Avoiding Overfitting Through Regularization

With four parameters I can fit an elephant and with five I can make him wiggle his trunk.

—John von Neumann, cited by Enrico Fermi in *Nature* 427

With thousands of parameters, you can fit the whole zoo. Deep neural networks typically have tens of thousands of parameters, sometimes even millions or billions. This gives them an incredible amount of freedom and means they can fit a huge variety of complex datasets. But this great flexibility also makes the network prone to overfitting the training set. Regularization is often needed to prevent this.

We already implemented a common regularization technique in [Chapter 4](#): early stopping. Moreover, even though batch-norm and layer-norm were designed to solve the unstable gradients problems, they also act like pretty good regularizers. In this section we will examine other popular regularization techniques for neural networks:  $\ell_1$  and  $\ell_2$  regularization, dropout, MC dropout, and max-norm regularization.

### $\ell_1$ and $\ell_2$ Regularization

Just like you did in [Chapter 4](#) for simple linear models, you can use  $\ell_2$  regularization to constrain a neural network's connection weights, and/or  $\ell_1$  regularization if you want a sparse model (with many weights equal to 0). As we saw earlier (when discussing the AdamW optimizer),  $\ell_2$  regularization is mathematically equivalent to weight decay when using an SGD optimizer (with or without momentum), so if that's the case you can implement  $\ell_2$  regularization by simply setting the optimizer's `weight_decay` argument. For example, here is how to apply  $\ell_2$  regularization to the connection weights of a PyTorch model trained using SGD, with a regularization factor of  $10^{-4}$ :

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.05, weight_decay=1e-4)
[...] # use the optimizer normally during training
```

If instead you are using an Adam optimizer, you should switch to AdamW and set the `weight_decay` argument. This is not exactly equivalent to  $\ell_2$  regularization, but as we saw earlier it's pretty close and it works better.

Note that weight decay is applied to every model parameter, including bias terms, and even parameters of batch-norm and layer-norm layers. Generally that's not a big deal, but penalizing these parameters does not contribute much to regularization and it may sometimes negatively impact training performance. So how can we apply weight decay to some model parameters and not others? One approach is to implement  $\ell_2$

regularization manually, without relying on the optimizer's weight decay feature. For this, you must tweak the training loop to manually compute the  $\ell_2$  loss based only on the parameters you want, and add this  $\ell_2$  loss to the main loss:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
params_to_regularize = [
    param for name, param in model.named_parameters()
    if not "bias" in name and not "bn" in name]
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        [...] # the rest of the training loop is unchanged
        main_loss = loss_fn(y_pred, y_batch)
        l2_loss = sum(param.pow(2.0).sum() for param in params_to_regularize)
        loss = main_loss + 1e-4 * l2_loss
        [...]
```

Another approach is to use PyTorch's *parameter groups* feature, which lets the optimizer apply different hyperparameters to different groups of model parameters. So far, we have always created optimizers by passing them the full list of model parameters: PyTorch automatically put them all in a single parameter group, sharing the same hyperparameters. Instead, we can pass a list of dictionaries to the optimizer, each with a "params" entry containing a list of parameters, and (optionally) some hyperparameter key/value pairs specific to this group of parameters. The group-specific hyperparameters take precedence over the optimizer's global hyperparameters. For example, let's create an optimizer with two parameter groups: the first group will contain all the parameters we want to regularize and it will use weight decay, while the second group will contain all the bias terms and BN parameters, and it will not use weight decay at all.

```
params_bias_and_bn = [
    param for name, param in model.named_parameters()
    if "bias" in name or "bn" in name]
optimizer = torch.optim.SGD([
    {"params": params_to_regularize, "weight_decay": 1e-4},
    {"params": params_bias_and_bn},
], lr=0.05)
[...] # use the optimizer normally during training
```



Parameter groups also allow you to apply different learning rates to different parts of your model. This is most common for transfer learning, when you want new layers to be updated faster than reused ones.

Now how about  $\ell_1$  regularization? Well unfortunately PyTorch does not provide any helper for this, so you need to implement it manually, much like we did for  $\ell_2$  regularization. This means tweaking the training loop to compute the  $\ell_1$  loss and adding it to the main loss:

```
l1_loss = sum(param.abs().sum() for param in params_to_regularize)
loss = main_loss + 1e-4 * l1_loss
```

That's all there is to it! Now let's move on to Dropout, which is one of the most popular regularization techniques for deep neural networks.

## Dropout

Dropout was proposed in a paper<sup>32</sup> by Geoffrey Hinton et al. in 2012 and further detailed in a 2014 paper<sup>33</sup> by Nitish Srivastava et al., and it has proven to be highly successful: many state-of-the-art neural networks use dropout, as it gives them a 1%–2% accuracy boost. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

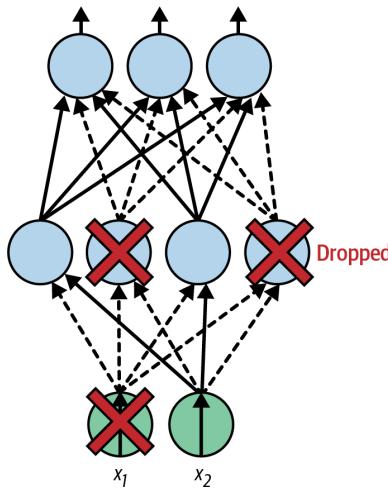
It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability  $p$  of being temporarily “dropped out”, meaning it will be entirely ignored during this training step, but it may be active during the next step (see [Figure 11-12](#)). The hyperparameter  $p$  is called the *dropout rate*, and it is typically set between 10% and 50%: closer to 20%–30% in recurrent neural nets (see [Chapter 13](#)), and closer to 40%–50% in convolutional neural networks (see [Chapter 12](#)). After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss shortly).

It's surprising at first that this destructive technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether to go to work? Well, who knows; perhaps it would! The company would be forced to adapt its organization; it could not rely on any single person to work the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn't make much of a difference. It's unclear whether this idea would actually work for companies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to all of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end, you get a more robust network that generalizes better.

---

<sup>32</sup> Geoffrey E. Hinton et al., “Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors”, arXiv preprint arXiv:1207.0580 (2012).

<sup>33</sup> Nitish Srivastava et al., “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, *Journal of Machine Learning Research* 15 (2014): 1929–1958.



*Figure 11-12. With dropout regularization, at each training iteration a random subset of all neurons in one or more layers—except the output layer—are “dropped out”; these neurons output 0 at this iteration (represented by the dashed arrows)*

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or absent, there are a total of  $2^N$  possible networks (where  $N$  is the total number of droppable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run 10,000 training steps, you have essentially trained 10,000 different neural networks, each with just one training instance. These neural networks are obviously not independent because they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.



Higher layers, which learn more complex feature combinations, benefit more from dropout because they are more prone to overfitting. So you can usually apply dropout only to the neurons of the top hidden layers (e.g., one to three hidden layers). However, you should avoid dropping the output neurons, as this would be like changing the task during training; it wouldn't help.

There is one small but important technical detail. Suppose  $p = 75\%$ : on average only 25% of all neurons are active at each step during training. This means that after training, each neuron receives four times more inputs than during training, on average. This discrepancy is so large that the model is unlikely to work well. To avoid this issue, a simple solution is to multiply the inputs by 4 during training, which is the

same as dividing them by 25%. More generally, we need to divide the inputs by the *keep probability* ( $1 - p$ ) during training.

To implement dropout using PyTorch, you can use the `nn.Dropout` layer. It's important to switch to training mode during training, and to evaluation mode during evaluation (just like for batch norm). In training mode, the layer randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. In evaluation mode, it does nothing at all; it just passes the inputs to the next layer. The following code applies dropout regularization before every `nn.Linear` layer, using a dropout rate of 0.2:

```
model = nn.Sequential(  
    nn.Flatten(),  
    nn.Dropout(p=0.2), nn.Linear(1 * 28 * 28, 100), nn.ReLU(),  
    nn.Dropout(p=0.2), nn.Linear(100, 100), nn.ReLU(),  
    nn.Dropout(p=0.2), nn.Linear(100, 100), nn.ReLU(),  
    nn.Dropout(p=0.2), nn.Linear(100, 10)  
).to(device)
```



Since dropout is only active during training, comparing the training loss and the validation loss can be misleading. In particular, a model may be overfitting the training set and yet have similar training and validation losses. So make sure to evaluate the training loss without dropout (e.g., after training).

If you observe that the model is overfitting, you can increase the dropout rate. Conversely, you should try decreasing the dropout rate if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones. Moreover, many state-of-the-art architectures only apply dropout to the last few hidden layers, so you may want to try this if full dropout is too strong.

Dropout does tend to significantly slow down convergence, but it often results in a better model when tuned properly. So it is generally well worth the extra time and effort, especially for large models.



If you want to regularize a self-normalizing network based on the SELU activation function (as discussed earlier), you should use *alpha dropout*: this is a variant of dropout that preserves the mean and standard deviation of its inputs. It was introduced in the same paper as SELU, as regular dropout would break self-normalization. PyTorch implements it in the `nn.AlphaDropout` layer.

## Monte Carlo Dropout

In 2016, a [paper<sup>34</sup>](#) by Yarin Gal and Zoubin Ghahramani added a few more good reasons to use dropout:

- First, the paper established a profound connection between dropout networks (i.e., neural networks containing `Dropout` layers) and approximate Bayesian inference,<sup>35</sup> giving dropout a solid mathematical justification.
- Second, the authors introduced a powerful technique called *Monte Carlo (MC) dropout*, which can boost the performance of any trained dropout model without having to retrain it or even modify it at all. It also provides a much better measure of the model's uncertainty, and it can be implemented in just a few lines of code.

This description of MC dropout sounds like some “one weird trick” clickbait, so let me explain: it is just like regular dropout, except it is active not only during training, but also during evaluation. This means that the predictions are always a bit random (hence the name Monte Carlo). But instead of making a single prediction, you make many predictions and average them out. It turns out that this produces better predictions than the original model.

Following is a full implementation of MC dropout, using the model we trained in the previous section to make predictions for a batch of images:

```
model.eval()
for module in model.modules():
    if isinstance(module, nn.Dropout):
        module.train()

X_new = [...] # some new images, e.g., the first 3 images of the test set
X_new = X_new.to(device)

torch.manual_seed(42)
with torch.no_grad():
    X_new_repeated = X_new.repeat_interleave(100, dim=0)
    y_logits_all = model(X_new_repeated).reshape(3, 100, 10)
    y_probas_all = torch.nn.functional.softmax(y_logits_all, dim=-1)
    y_probas = y_probas_all.mean(dim=1)
```

---

<sup>34</sup> Yarin Gal and Zoubin Ghahramani, “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”, *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1050–1059.

<sup>35</sup> Specifically, they show that training a dropout network is mathematically equivalent to approximate Bayesian inference in a specific type of probabilistic model called a *deep Gaussian process*.

Let's go through this code:

- First, we switch the model to evaluation mode as we always do before making predictions, but this time we immediately switch all the dropout layers back to training mode, so they will behave just like during training (i.e., randomly dropping out some of their inputs). In other words, we convert the dropout layers to MC dropout layers.
- Next we load a new batch of images `X_new`, and we move it to the GPU. In this example, let's assume `X_new` contains three images.
- We then use the `repeat_interleave()` method to create a batch containing 100 copies of each image in `X_new`. The images are repeated along the first dimension (`dim=0`) so `X_new_repeated` has a shape of [300, 1, 28, 28].
- Next, we pass this big batch to the model, which predicts 10 logits per image, as usual. This tensor's shape is [300, 10], but we reshape it to [3, 100, 10] to group the predictions for each image. Remember that the dropout layers are active, which means that there's some variability across the predictions, even for copies of the same image.
- Then we convert these logits to estimated probabilities using the softmax function.
- Lastly, we compute the mean over the second dimension (`dim=1`) to get the average estimated probability for each class and each image, across all 100 predictions. The result is a tensor of shape [3, 10]. These are our final predictions:

```
>>> y_probas.round(decimals=2)
tensor([[0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.010, 0.000, 0.990],
       [0.990, 0.000, 0.000, 0.000, 0.000, 0.000, 0.010, 0.000, 0.000],
       [0.410, 0.040, 0.040, 0.230, 0.040, 0.000, 0.230, 0.000, 0.010, 0.000]],
      device='cuda:0')
```



Rather than converting the logits to probabilities and then computing the mean probabilities, you may be tempted to do the reverse: first average over the logits and *then* convert the mean logits to probabilities. This is faster but it does not properly reflect the model's uncertainty, so it tends to produce overconfident models.

MC dropout tends to improve the reliability of the model's probability estimates. This means that it's less likely to be confidently wrong, making it safer (you don't want a self-driving car confidently ignoring a stop sign). It's also useful when you're interested in the top  $k$  classes, not just the most likely. Additionally, you can take a look at the **standard deviation of each class probability**:

```
>>> y_std = y_probas_all.std(dim=1)
>>> y_std.round(decimals=2)
```

```
tensor([[0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.020, 0.000, 0.020],
       [0.020, 0.000, 0.000, 0.000, 0.000, 0.000, 0.020, 0.000, 0.000, 0.000],
       [0.170, 0.030, 0.030, 0.130, 0.050, 0.000, 0.090, 0.000, 0.010, 0.000]],
      device='cuda:0')
```

There's a standard deviation of 0.02 for the probability estimate of class 9 (ankle boot) for the first image. This adds a grain of salt to the estimated probability of 99% for this class: in fact, the model is really saying "mmh, I'm guessing over 95%". If you were building a risk-sensitive system (e.g., a medical or financial system), you may want to consider only the predictions with both a high estimated probability *and* a low standard deviation.



The number of Monte Carlo samples you use (100 in this example) is a hyperparameter you can tweak. The higher it is, the more accurate the predictions and their uncertainty estimates are, but also the slower the predictions are. Moreover, above a certain number of samples, you will notice little improvement. Your job is to find the right trade-off among latency, throughput, and accuracy, depending on your application.

If you want to train an MC dropout model from scratch rather than reuse an existing dropout model, you should probably use a custom `McDropout` module rather than using `nn.Dropout` and hacking around with `train()` and `eval()`, as this is a bit brittle (e.g., it won't play nicely with the evaluation function). Here is a three-line implementation:

```
class McDropout(nn.Dropout):
    def forward(self, input):
        return F.dropout(input, self.p, training=True)
```

In short, MC dropout is a great technique that boosts dropout models and provides better uncertainty estimates. And of course, since it is just regular dropout during training, it also acts like a regularizer.

## Max-Norm Regularization

Another fairly popular regularization technique for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights  $\mathbf{w}$  of the incoming connections such that  $\|\mathbf{w}\|_2 \leq r$ , where  $r$  is the max-norm hyperparameter and  $\|\cdot\|_2$  is the  $\ell_2$  norm.

Reducing  $r$  increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the unstable gradients problems (if you are not using batch-norm or layer-norm).

Rather than adding a regularization loss term to the overall loss function, max-norm regularization is typically implemented by computing  $\|\mathbf{w}\|_2$  after each training step

and rescaling  $\mathbf{w}$  if needed ( $\mathbf{w} \leftarrow \mathbf{w} r / \|\mathbf{w}\|_2$ ). Here's a common way to implement this in PyTorch:

```
def apply_max_norm(model, max_norm=2, epsilon=1e-8, dim=1):
    with torch.no_grad():
        for name, param in model.named_parameters():
            if 'bias' not in name:
                actual_norm = param.norm(p=2, dim=dim, keepdim=True)
                target_norm = torch.clamp(actual_norm, 0, max_norm)
                param *= target_norm / (epsilon + actual_norm)
```

This function iterates through all of the model's weight matrices (i.e., all parameters except for the bias terms), and for each one of them it uses the `norm()` method to compute the  $\ell_2$  norm of each row (`dim=1`). A `nn.Linear` layer has weights of shape [*number of neurons, number of inputs*], so using `dim=1` means that we will get one norm per neuron, as desired. Then the function uses `torch.clamp()` to compute the target norm for each neuron's weights: this creates a copy of the `actual_norm` tensor, except that all values greater than `max_norm` are replaced by `max_norm` (this corresponds to  $r$  in the previous equation). Lastly, we rescale the weight matrix so that each column ends up with the target norm. Note that the smoothing term `epsilon` is used to avoid division by zero in case some columns have a norm equal to zero.

Next, all you need to do is call `apply_max_norm(model)` in the training loop, right after calling the optimizer's `step()` method. And of course you probably want to fine-tune the `max_norm` hyperparameter.



When using max-norm with layers other than `nn.Linear`, you may need to tweak the `dim` argument. For example, when using convolutional layers (see [Chapter 12](#)), you generally want to set `dim=[1, 2, 3]` to limit the norm of each convolutional kernel.

## Practical Guidelines

In this chapter we have covered a wide range of techniques, and you may be wondering which ones you should use. This depends on the task, and there is no clear consensus yet, but I have found the configuration in [Table 11-3](#) to work fine in most cases, without requiring much hyperparameter tuning. That said, please do not consider these defaults as hard rules!

*Table 11-3. Default DNN configuration*

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ReLU if shallow; Swish if deep
Normalization	None if shallow; batch-norm or layer-norm if deep

Hyperparameter	Default value
Regularization	Early stopping; weight decay if needed
Optimizer	Nesterov accelerated gradients or AdamW
Learning rate schedule	Performance scheduling or 1cycle

You should also try to reuse parts of a pretrained neural network if you can find one that solves a similar problem, or use unsupervised pretraining if you have a lot of unlabeled data, or use pretraining on an auxiliary task if you have a lot of labeled data for a similar task.

While the previous guidelines should cover most cases, there are some exceptions:

- If you need a sparse model, use  $\ell_1$  regularization. You can also try zeroing out the smallest weights after training (for example, using the `torch.nn.utils.prune.l1_unstructured()` function). This will break self-normalization, so you should use the default configuration in this case.
- If you need a low-latency model (one that performs lightning-fast predictions), you may need to use fewer layers; use a fast activation function such as `nn.ReLU`, `nn.LeakyReLU`, or `nn.Hardswish`; and fold the batch-norm and layer-norm layers into the previous layers after training. Having a sparse model will also help. Finally, you may want to reduce the float precision from 32 bits to 16 or even 8 bits. [Appendix B](#) covers several techniques to make models smaller and faster, including reduced precision models, mixed precision models, and quantization.
- If you are building a risk-sensitive application, or inference latency is not very important in your application, you can use MC dropout to boost performance and get more reliable probability estimates, along with uncertainty estimates.

Over the last three chapters, we have learned what artificial neural nets are, how to build and train them using Scikit-Learn and PyTorch, and a variety of techniques that make it possible to train deep and complex nets. In the next chapter, all of this will come together as we dive into one of the most important applications of deep learning: computer vision.

## Exercises

1. What is the problem that Glorot initialization and He initialization aim to fix?
2. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
3. Is it OK to initialize the bias terms to 0?
4. In which cases would you want to use each of the activation functions we discussed in this chapter?

5. What may happen if you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC dropout?
8. Practice training a deep neural network on the CIFAR10 image dataset:
  - a. Load CIFAR10 just like you loaded the FashionMNIST dataset in [Chapter 10](#), but using `torchvision.datasets.CIFAR10` instead of `FashionMNIST`. The dataset is composed of 60,000  $32 \times 32$ -pixel color images (50,000 for training, 10,000 for testing) with 10 classes.
  - b. Build a DNN with 20 hidden layers of 100 neurons each (that's too many, but it's the point of this exercise). Use He initialization and the Swish activation function (using `nn.SiLU`). Since this is a classification task, you will need an output layer with one neuron per class.
  - c. Using NAdam optimization and early stopping, train the network on the CIFAR10 dataset. Remember to search for the right learning rate each time you change the model's architecture or hyperparameters.
  - d. Now try adding batch-norm and compare the learning curves: is it converging faster than before? Does it produce a better model? How does it affect training speed?
  - e. Try replacing batch-norm with SELU, and make the necessary adjustments to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).
  - f. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC dropout.
  - g. Retrain your model using 1cycle scheduling and see if it improves training speed and model accuracy.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.



# 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 our sensory modules work.

*Convolutional neural networks* (CNNs) emerged from the study of the brain's visual cortex, and they have been used in computer image recognition since the 1980s. Over the last 15 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 PyTorch. 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 1958<sup>1</sup> and 1959<sup>2</sup> (and a few years later on monkeys<sup>3</sup>), 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 12-1, 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.

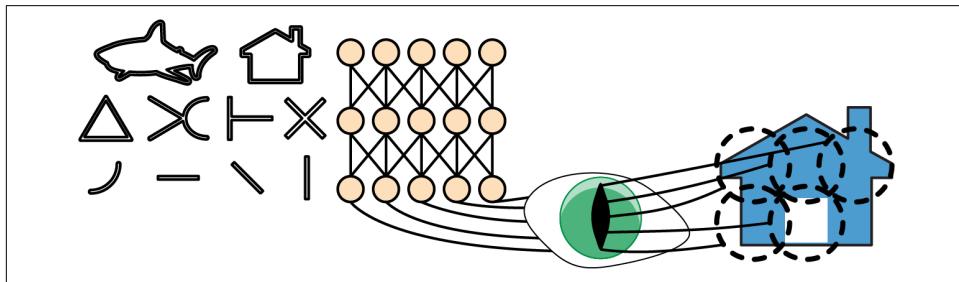


Figure 12-1. Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields

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 12-1, notice that each neuron is connected only to

<sup>1</sup> David H. Hubel, "Single Unit Activity in Striate Cortex of Unrestrained Cats", *The Journal of Physiology* 147 (1959): 226–238.

<sup>2</sup> 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.

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

nearby neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

These studies of the visual cortex inspired the **neocognitron**,<sup>4</sup> introduced in 1980, which gradually evolved into what we now call convolutional neural networks. An important milestone was a **1998 paper**<sup>5</sup> by Yann LeCun et al. that introduced the famous *LeNet-5* architecture, which became widely used by banks to recognize handwritten digits on checks. 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., Fashion MNIST), it breaks down for larger images because of the huge number of parameters it requires. For example, a  $100 \times 100$ -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.

## Convolutional Layers

The most important building block of a CNN is the *convolutional layer*.<sup>6</sup> 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 12-2](#)). 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 well-suited to deal with composite objects, which are common in real-world images: this is one of the reasons why CNNs work so well for image recognition.

---

<sup>4</sup> 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.

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

<sup>6</sup> 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).

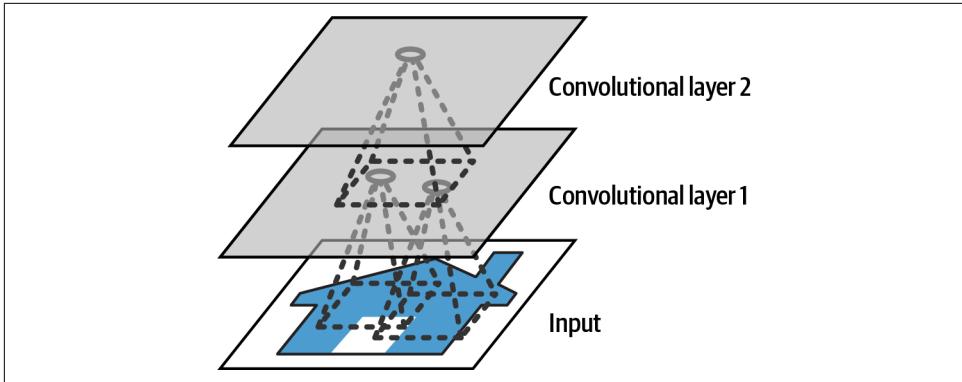


Figure 12-2. CNN layers with rectangular local receptive fields



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.

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 12-3](#)). 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*.

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 12-4](#). This dramatically reduces the model's computational complexity. The horizontal or vertical step size from one receptive field to the next is called the *stride*. In the diagram, a  $5 \times 7$  input layer (plus zero padding) is connected to a  $3 \times 4$  layer, using  $3 \times 3$  receptive fields and a stride of 2. In this example the stride is the same in both directions, which is generally the case (although there are exceptions). 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.

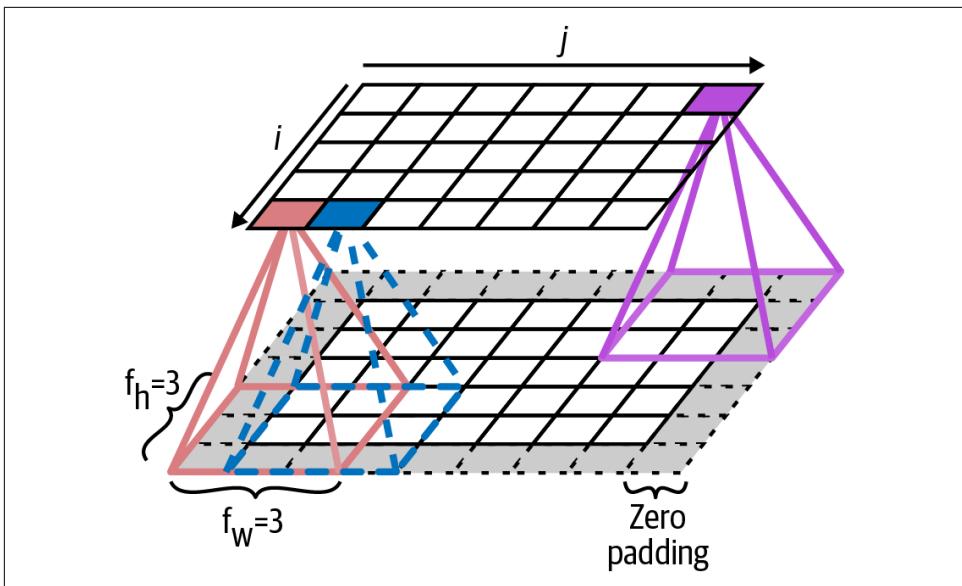


Figure 12-3. Connections between layers and zero padding

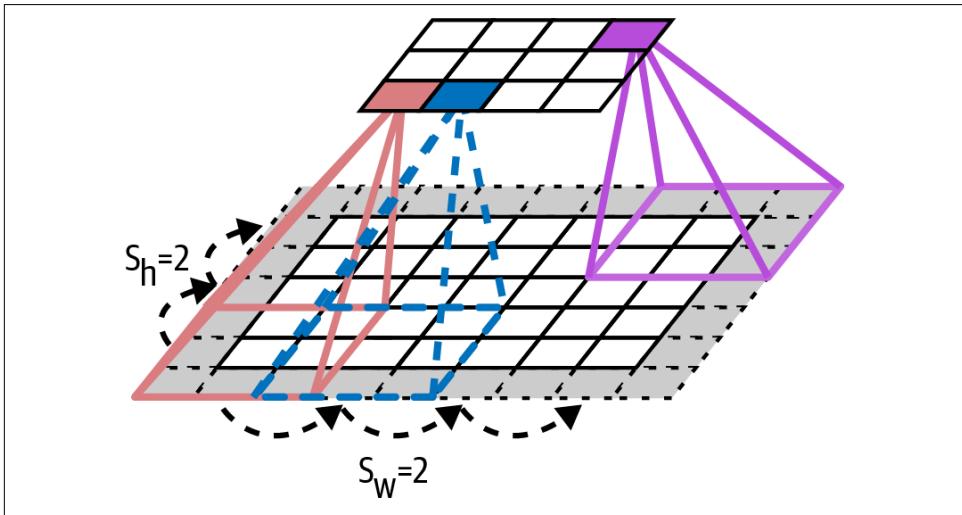
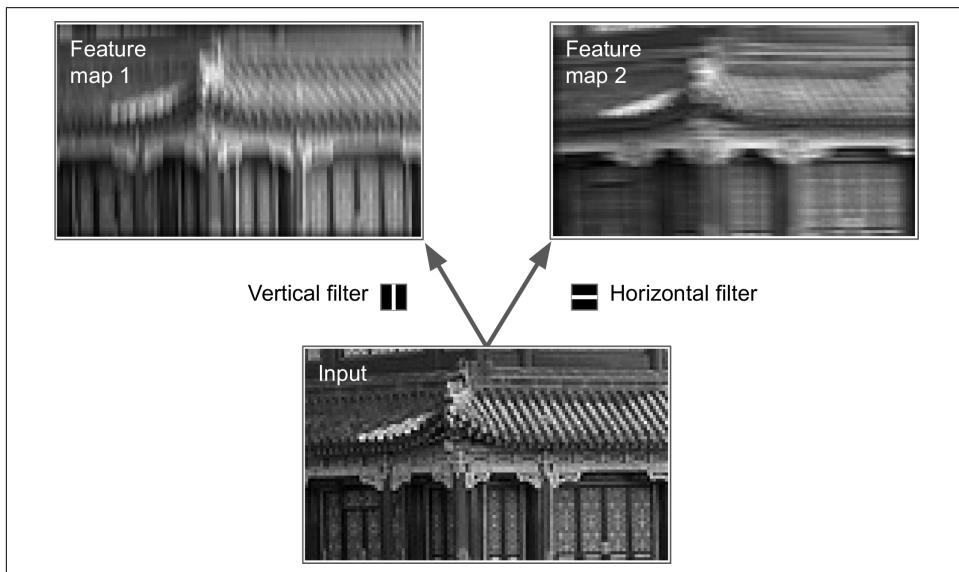


Figure 12-4. Reducing dimensionality using a stride of 2

## Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, [Figure 12-5](#) shows two possible sets of weights, called *filters* (or *convolution kernels*, or just *kernels*). The first one is represented as a black square with a vertical white line in the middle (it's a  $7 \times 7$  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 be multiplied by 0, except for the ones in the central vertical line). The second filter is a black square with a horizontal white line in the middle. Neurons using these weights will ignore everything in their receptive field except for the central horizontal line.



*Figure 12-5. Applying two different filters to get two feature maps*



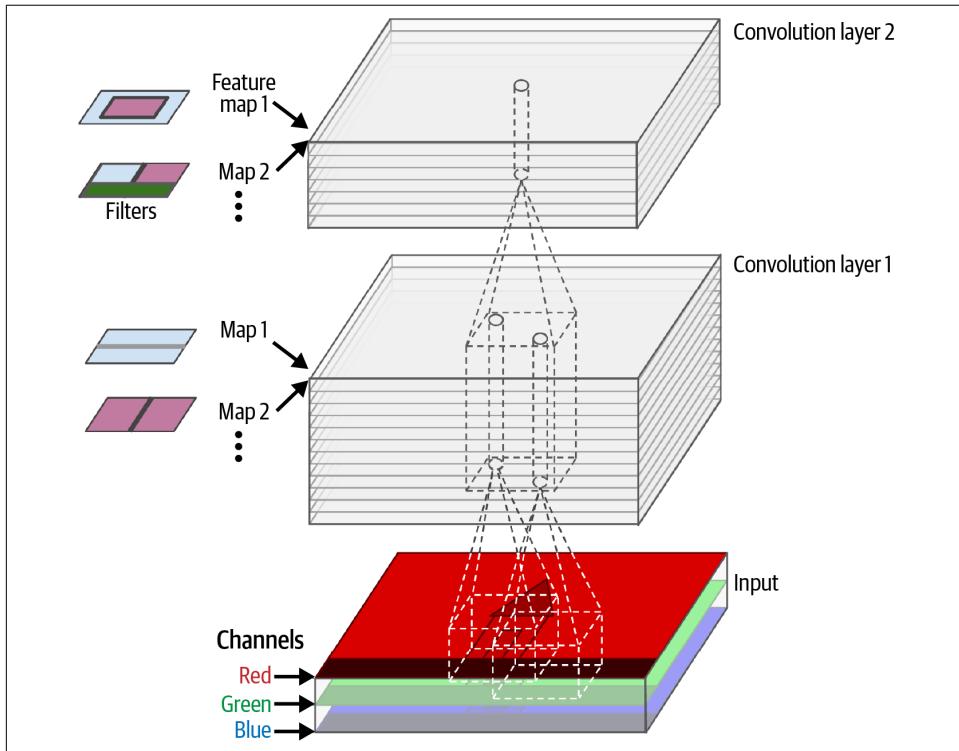
In deep learning, we often build a single model that takes the raw inputs and produces the final outputs. This is called *end-to-end learning*. In contrast, classical vision systems would usually divide the system into a sequence of specialized modules.

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 12-5](#) (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. But don't worry, you won't 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.

## Stacking Multiple Feature Maps

Up to now, for simplicity, I have represented each convolutional layer as a 2D layer, but in reality a convolutional layer has multiple filters (you decide how many) and it outputs one feature map per filter, so the output is more accurately represented in 3D (see [Figure 12-6](#)).



*Figure 12-6. Two convolutional layers with multiple filters each (kernels), processing a color image with three color channels; each convolutional layer outputs one feature map per filter*

There is one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same kernel 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 feature maps of the previous

layer. 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 fully connected neural network has learned to recognize a pattern in one location, it can only recognize it in that particular location.

Input images are also composed of multiple sublayers: one per *color channel*. As mentioned in [Chapter 8](#), there are typically three: red, green, and blue (RGB). Grayscale images have just one channel, but some images may have many more—for example, satellite images that capture extra light frequencies (such as infrared).

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, within a layer, 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 12-1](#) 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 12-1. Computing the output of a neuron in a convolutional layer*

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

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  $k'$ .

Let's see how to create and use a convolutional layer using PyTorch.

## Implementing Convolutional Layers with PyTorch

First, let's load a couple of sample images using Scikit-Learn's `load_sample_images()` function. The first image represents the tower of buddhist incense in China, while the second one represents a beautiful *Dahlia pinnata* flower. These images are represented as a Python list of NumPy unsigned byte arrays, so let's stack these images into a single NumPy array, then convert it to a 32-bit float tensor, and rescale the pixel values from 0–255 to 0–1:

```
import numpy as np
import torch
from sklearn.datasets import load_sample_images

sample_images = np.stack(load_sample_images()["images"])
sample_images = torch.tensor(sample_images, dtype=torch.float32) / 255
```

Let's look at this tensor's shape:

```
>>> sample_images.shape
torch.Size([2, 427, 640, 3])
```

We have two images, both are 427 pixels high and 640 pixels wide, and they have three color channels: red, green, and blue. As we saw in [Chapter 10](#), PyTorch expects the channel dimension to be just *before* the height and width dimensions, not after, so we need to permute the dimensions using the `permute()` method:

```
>>> sample_images_permuted = sample_images.permute(0, 3, 1, 2)
>>> sample_images_permuted.shape
torch.Size([2, 3, 427, 640])
```

Let's also use TorchVision's `CenterCrop` class to center-crop the images:

```
>>> import torchvision
>>> import torchvision.transforms.v2 as T
>>> cropped_images = T.CenterCrop((70, 120))(sample_images_permuted)
>>> cropped_images.shape
torch.Size([2, 3, 70, 120])
```

Now let's create a 2D convolutional layer and feed it these cropped images to see what comes out. For this, PyTorch provides the `nn.Conv2d` layer. Under the hood, this layer relies on the `torch.nn.((("torch", "F.conv2d()"))functional.conv2d())` function. Let's create a convolutional layer with 32 filters, each of size  $7 \times 7$  (using

`kernel_size=7`, which is equivalent to using `kernel_size=(7, 7)`, and apply this layer to our small batch of two images:

```
import torch.nn as nn

torch.manual_seed(42)
conv_layer = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7)
fmaps = conv_layer(cropped_images)
```



When we talk about a 2D convolutional layer, “2D” refers to the number of *spatial* dimensions (height and width), but as you can see, the layer takes 4D inputs: as we saw, the two additional dimensions are the batch size (first dimension) and the channels (second dimension).

Now let’s look at the output’s shape:

```
>>> fmaps.shape
torch.Size([2, 32, 64, 114])
```

The output shape is similar to the input shape, with two main differences. First, there are 32 channels instead of 3. This is because we set `out_channels=32`, so we get 32 output feature maps: instead of the intensity of red, green, and blue at each location, we now have the intensity of each feature at each location. Second, the height and width have both shrunk by 6 pixels. This is due to the fact that the `nn.Conv2d` layer does not use any zero-padding by default, which means that we lose a few pixels on the sides of the output feature maps, depending on the size of the filters. In this case, since the kernel size is 7, we lose 6 pixels horizontally and 6 pixels vertically (i.e., 3 pixels on each side).



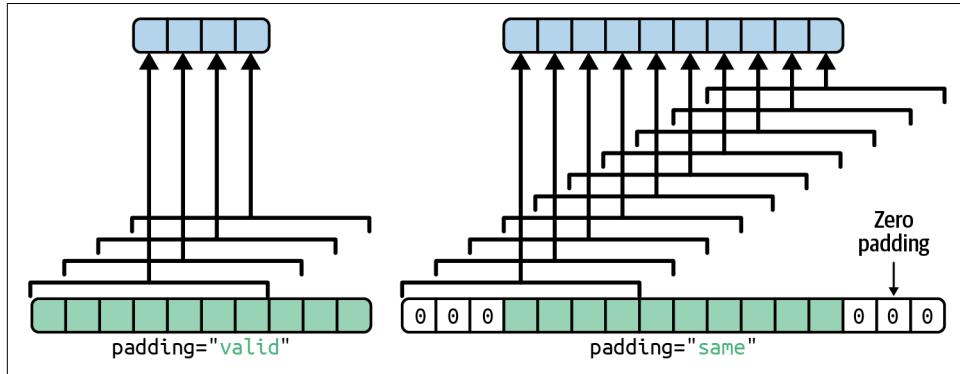
By default, the `padding` hyperparameter is set to 0, which means that padding is turned off. Oddly, this is also called *valid padding* since every neuron’s receptive field lies strictly within *valid* positions inside the input (it does not go out of bounds). You can actually set `padding="valid"`, which is equivalent to `padding=0`. It’s not a PyTorch naming quirk: everyone uses this confusing nomenclature.

If instead we set `padding="same"`, then the inputs are padded with enough zeros on all sides to ensure that the output feature maps end up with the *same* size as the inputs (hence the name of this option):

```
>>> conv_layer = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7,
...                         padding="same")
...
>>> fmaps = conv_layer(cropped_images)
```

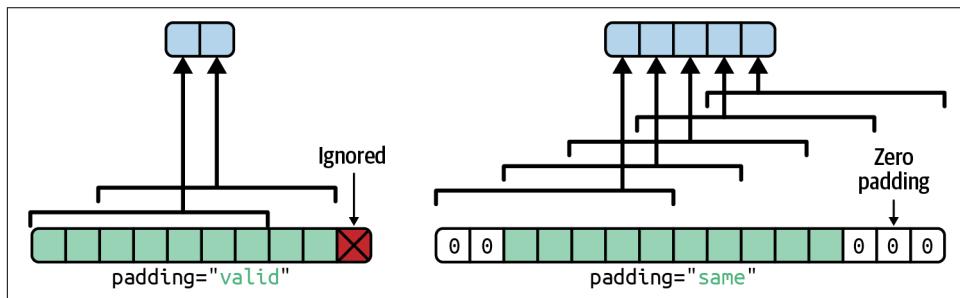
```
>>> fmaps.shape  
torch.Size([2, 32, 70, 120])
```

These two padding options are illustrated in [Figure 12-7](#). For simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension as well.



*Figure 12-7. Two different padding options, with `stride=1` and `kernel_size=7`*

If the stride is greater than 1 (in any direction), then the output size will be much smaller than the input size. For example, assuming the input size is  $70 \times 120$ , then if you set `stride=2` (or equivalently `stride=(2, 2)`), `padding=3`, and `kernel_size=7`, then the output feature maps will be  $35 \times 60$ : halved both vertically and horizontally. You could set a very large padding value to make the output size identical to the input size, but that's almost certainly a bad idea since it would drown your image in a sea of zeros (for this reason, PyTorch raises an exception if you set `padding="same"` along with a `stride` greater than 1). [Figure 12-8](#) illustrates `stride=2`, with `kernel_size=7` and `padding` set to 0 or 3.



*Figure 12-8. Two different padding options, with `stride=2` and `kernel_size=7`: the output size is much smaller*

Now let's look at the layer's parameters (which were denoted as  $w_{u, v, k, k}$  and  $b_k$  in [Equation 12-1](#)). Just like an `nn.Linear` layer, an `nn.Conv2d` layer holds all the layer's parameters, including the kernels and biases, which are accessible via the `weight` and `bias` attributes:

```
>>> conv_layer.weight.shape  
torch.Size([32, 3, 7, 7])  
>>> conv_layer.bias.shape  
torch.Size([32])
```

The `weight` tensor is 4D, and its shape is [*output\_channels*, *input\_channels*, *kernel\_height*, *kernel\_width*]. The `bias` tensor is 1D, with shape [*output\_channels*]. The number of output channels is equal to the number of output feature maps, which is also equal to the number of filters. Most importantly, note that the height and width of the input images do not appear in the kernel's shape: this is because all the neurons in the output feature maps share the same weights, as explained earlier. This means that you can feed images of any size to this layer, as long as they are at least as large as the kernels, and if they have the right number of channels (three in this case).

It's important to add an activation function after each convolutional layer. This is for the same reason as for `nn.Linear` layers: a convolutional layer performs a linear operation, so if you stacked multiple convolutional layers without any activation functions, they would all be equivalent to a single convolutional layer, and they wouldn't be able to learn anything really complex.

Both the `weight` and `bias` parameters are initialized randomly, using a uniform distribution similar to the one used by the `nn.Linear` layer, between  $-\frac{1}{\sqrt{k}}$  and  $+\frac{1}{\sqrt{k}}$ , where  $k$  is the fan<sub>in</sub>. In `nn.Conv2d`,  $k = f_h \times f_w \times f_n$ , where  $f_h$  and  $f_w$  are the height and width of the kernel, and  $f_n$  is the number of input channels. As we saw in [Chapter 11](#), you will generally want to reinitialize the weights depending on the activation function you use. For example, you should apply He initialization whenever you use the ReLU activation function. As for the biases, they can just be reinitialized to zero.

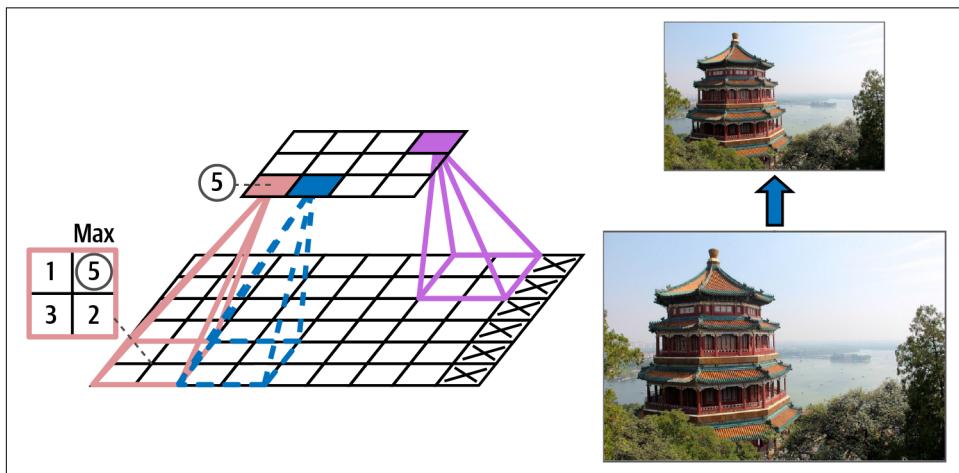
As you can see, convolutional layers have quite a few hyperparameters: the number of filters (`out_channels`), the kernel size, the type of padding, the strides, and the activation function. As always, you can use cross-validation to find the right hyperparameter values, but this is very time-consuming. We will discuss common CNN architectures later in this chapter to give you some idea of which hyperparameter values work best in practice.

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 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 or biases; all it does is aggregate the inputs using an aggregation function such as the max or mean. [Figure 12-9](#) shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a  $2 \times 2$  *pooling kernel*,<sup>7</sup> 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 12-9](#), the input values are 1, 5, 3, and 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).



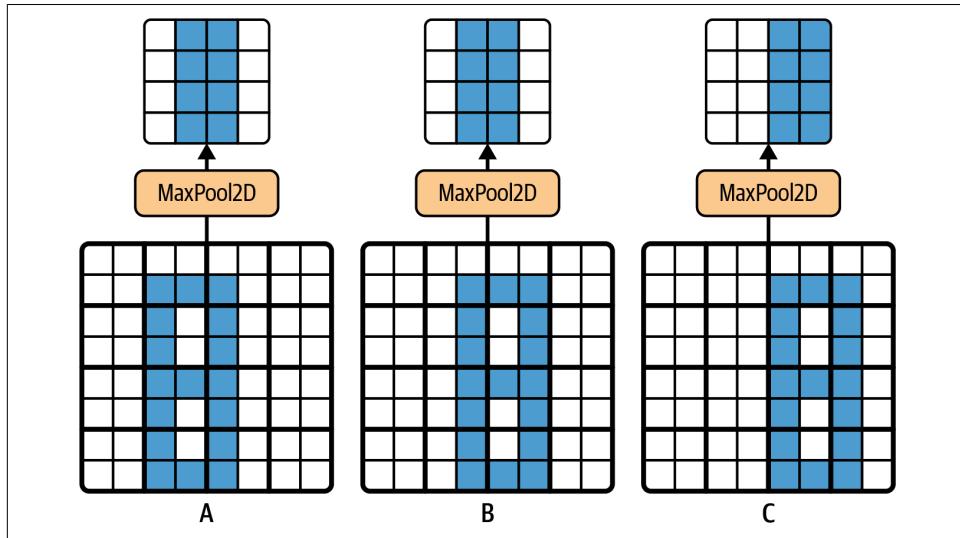
*Figure 12-9. Max pooling layer ( $2 \times 2$  pooling kernel, stride 2, no padding)*

<sup>7</sup> Other kernels we've discussed so far had weights, but pooling kernels do not: they are just stateless sliding windows.



A pooling layer typically works on every input channel independently, so the output depth (i.e., the number of channels) 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 12-10](#). 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  $2 \times 2$  kernel and stride 2. Images B and C are the same as image A, but 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 50% 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 12-10. Invariance to small translations*

However, max pooling has some downsides too. It's obviously very destructive: even with a tiny  $2 \times 2$  kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), thereby 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.

## Implementing Pooling Layers with PyTorch

The following code creates an `nn.MaxPool2d` layer, using a  $2 \times 2$  kernel. The strides default to the kernel size, so this layer uses a stride of 2 (horizontally and vertically). By default, it uses `padding=0` (i.e., "valid" padding):

```
max_pool = nn.MaxPool2d(kernel_size=2)
```

To create an *average pooling layer*, just use `nn.AvgPool2d`, instead of `nn.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 also be performed along the depth dimension instead of the spatial dimensions, although it's 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 hand-written digits; see [Figure 12-11](#)), 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: thickness, brightness, skew, color, and so on.

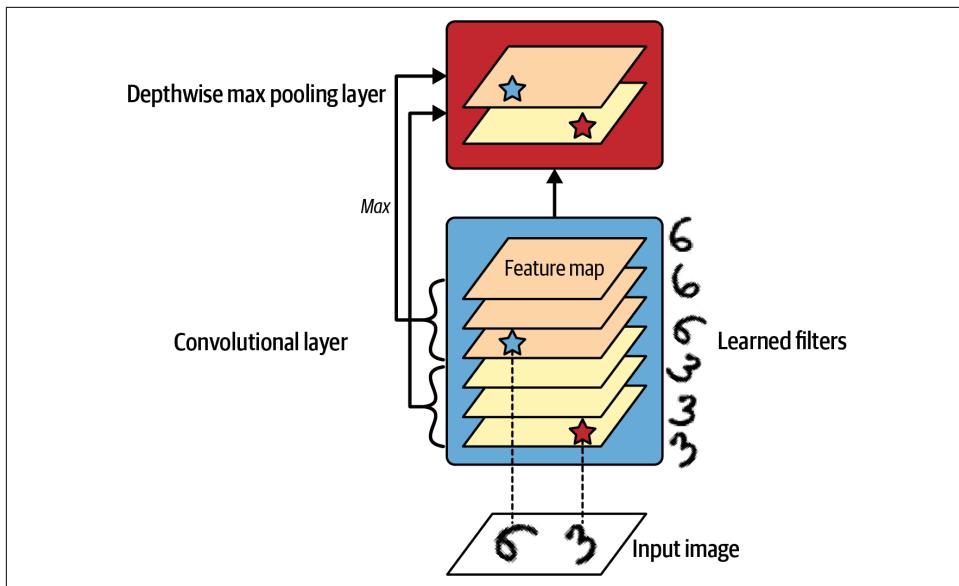


Figure 12-11. Depthwise max pooling can help the CNN learn to be invariant (to rotation in this case)

PyTorch does not include a depthwise max pooling layer, but we can implement a custom module based on the `torch.F.max_pool1d()` function:

```
import torch.nn.functional as F

class DepthPool(torch.nn.Module):
    def __init__(self, kernel_size, stride=None, padding=0):
        super().__init__()
        self.kernel_size = kernel_size
        self.stride = stride if stride is not None else kernel_size
        self.padding = padding

    def forward(self, inputs):
        batch, channels, height, width = inputs.shape
        Z = inputs.view(batch, channels, height * width) # merge spatial dims
        Z = Z.permute(0, 2, 1) # switch spatial and channels dims
        Z = F.max_pool1d(Z, kernel_size=self.kernel_size, stride=self.stride,
                         padding=self.padding) # compute max pool
        Z = Z.permute(0, 2, 1) # switch back spatial and channels dims
        return Z.view(batch, -1, height, width) # unmerge spatial dims
```

For example, suppose the input batch contains two  $70 \times 120$  images, each with 32 channels (i.e., the inputs have a shape of [2, 32, 70, 120]), and we use `kernel_size=4`, and the default `stride` (equal to `kernel_size`) and `padding=0`:

- The `forward()` method starts by merging the spatial dimensions, which gives us a tensor of shape [2, 32, 8400] (since  $70 \times 120 = 8,400$ ).
- It then permutes the last two dimensions, so we get a shape of [2, 8400, 32].
- Next, it uses the `max_pool1d()` function to compute the max pool along the last dimension, which corresponds to our original 32 channels. Since `kernel_size` and `stride` are both equal to 4, and we don't use any padding, the size of the last dimension gets divided by 4, so the resulting shape is [2, 8400, 8].
- The function then permutes the last two dimensions again, giving us a shape of [2, 8, 8400].
- Lastly, it separates the spatial dimensions to get the final shape of [2, 8, 50, 100]. You can verify that the output is exactly what we were after.

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. Therefore it 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 just before the output layer, as you will see later in this chapter.

To create such a layer, one option is to use a regular `nn.AvgPool2d` layer and set its kernel size to the same size as the inputs. However, this is not very convenient since it requires knowing the exact dimensions of the inputs ahead of time. A simpler solution is to use the `nn.AdaptiveAvgPool2d` layer, which lets you specify the desired spatial dimensions of the output: it automatically adapts the kernel size (with an equal stride) to get the desired result, adding a bit of padding if needed. If we set the output size to 1, we get a global average pooling layer:

```
global_avg_pool = nn.AdaptiveAvgPool2d(output_size=1)
output = global_avg_pool(cropped_images)
```

Alternatively, you could just use the `torch.mean()` function to get the same output:

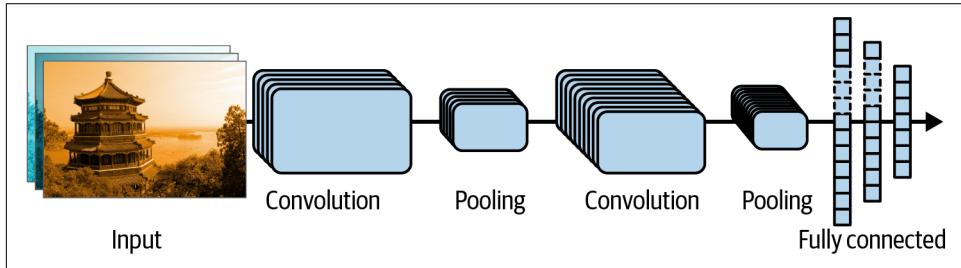
```
output = cropped_images.mean(dim=(2, 3), keepdim=True)
```

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 12-12](#)). 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 12-12. Typical CNN architecture*



Instead of using a convolutional layer with a  $5 \times 5$  kernel, it is generally preferable to stack two layers with  $3 \times 3$  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.,  $5 \times 5$  or  $7 \times 7$ ), usually with a stride of 2 or more. This reduces 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 basic CNN to tackle the Fashion MNIST dataset (introduced in [Chapter 9](#)):

```
from functools import partial

DefaultConv2d = partial(nn.Conv2d, kernel_size=3, padding="same")
model = nn.Sequential(
    DefaultConv2d(in_channels=1, out_channels=64, kernel_size=7), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    DefaultConv2d(in_channels=64, out_channels=128), nn.ReLU(),
    DefaultConv2d(in_channels=128, out_channels=128), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    DefaultConv2d(in_channels=128, out_channels=256), nn.ReLU(),
    DefaultConv2d(in_channels=256, out_channels=256), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Flatten(),
    nn.Linear(in_features=2304, out_features=128), nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(in_features=128, out_features=64), nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(in_features=64, out_features=10),
).to(device)
```

Let's go through this code:

- We use the `functools.partial()` function (introduced in [Chapter 11](#)) to define `DefaultConv2d`, which acts just like `nn.Conv2d` but with different default arguments: a small kernel size of 3, and "same" padding. This avoids having to repeat these arguments throughout the model.
- Next, we create the `nn.Sequential` model. Its first layer is a `DefaultConv2d` with 64 fairly large filters ( $7 \times 7$ ). It uses the default stride of 1 because the input images are not very large. It also uses `in_channels=1` because the Fashion MNIST images have a single color channel (i.e., grayscale). Each convolutional layer is followed by the ReLU activation function.
- We then add a max pooling layer with a kernel size of 2, so it divides each spatial dimension by a factor of 2 (rounded down if needed).
- 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 doubles 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 (`nn.Linear`) with the ReLU activation function, plus a dense output layer. Since it's a classification task with 10 classes, the output layer has 10 units. As we did in [Chapter 10](#), we leave out the softmax activation function, so the model will output logits rather than probabilities, and we must use the `nn.CrossEntropyLoss` to train the model. Note that we must flatten the inputs just before the first dense layer, since it 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.



The first `nn.Linear` layer has 2,304 input features: where did this number come from? Well the Fashion MNIST images are  $28 \times 28$  pixels, but the pooling layers shrink them to  $14 \times 14$ , then  $7 \times 7$ , and finally  $3 \times 3$ . Just before the first `nn.Linear` layer, there are 256 feature maps, so we end up with  $256 \times 3 \times 3 = 2,304$  input features. Figuring out the number of features can sometimes be a bit difficult, but one trick is to set `in_features` to some arbitrary value (say, 999), and let training crash. The correct number of features appears in the error message: “`RuntimeError: mat1 and mat2 shapes cannot be multiplied (32x2304 and 999x128)`”. Another option is to use `nn.LazyLinear` instead of `nn.Linear`: it’s just like the `nn.Linear` layer, except it only creates the weights matrix the first time it gets called: it can then automatically set the number of input features to the correct value. Other layers—such as convolutional layers and batch-norm layers—also have lazy variants.

If you train this model on the Fashion MNIST training set, it should reach close to 92% accuracy on the test set (you can use the `train()` and `evaluate_tm()` functions we defined in [Chapter 10](#)). It’s not state of the art, but it is pretty good, and better than what we achieved with dense networks in [Chapter 9](#).

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 error rate for image classification fell from over 26% to less than 2.3% in just 6 years. More precisely, this was the *top-five error rate*, which is the ratio of test images for which the system’s five most confident predictions did *not* include the correct answer. The images are fairly large (e.g., 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, and how research in deep learning progresses.

We will first look at the classical LeNet-5 architecture (1998), then several winners of the ILSVRC challenge: AlexNet (2012), GoogLeNet (2014), ResNet (2015), and SENet (2017). We will also discuss a few more architectures, including VGGNet, Xception, ResNeXt, DenseNet, MobileNet, CSPNet, EfficientNet, and ConvNeXt (and we will discuss vision transformers in [Chapter 16](#)).

## LeNet-5

The [LeNet-5 architecture](#)<sup>8</sup> 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 12-1](#).

*Table 12-1. 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	–	–	–

As you can see, this looks pretty similar to our Fashion MNIST model: a stack of convolutional layers and pooling layers, followed by a dense network. Perhaps the main difference with more modern classification CNNs is the activation functions: today, we would use ReLU instead of tanh, and softmax instead of RBF (introduced in [Chapter 2](#)). There were several other minor differences that don't really matter much, but in case you are interested, they are listed in this chapter's notebook at <https://homl.info/colab-p>. Yann LeCun's [website](#) also features great demos of LeNet-5 classifying digits.

## AlexNet

The [AlexNet CNN architecture](#)<sup>9</sup> won the 2012 ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best competitor achieved only 26%! AlexNet 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 12-2](#) presents this architecture.

---

<sup>8</sup> Yann LeCun et al., “Gradient-Based Learning Applied to Document Recognition”, *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

<sup>9</sup> 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.

Table 12-2. 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	$6 \times 6$	$3 \times 3$	2	valid	–
C7	Convolution	256	$13 \times 13$	$3 \times 3$	1	same	ReLU
C6	Convolution	384	$13 \times 13$	$3 \times 3$	1	same	ReLU
C5	Convolution	384	$13 \times 13$	$3 \times 3$	1	same	ReLU
S4	Max pooling	256	$13 \times 13$	$3 \times 3$	2	valid	–
C3	Convolution	256	$27 \times 27$	$5 \times 5$	1	same	ReLU
S2	Max pooling	96	$27 \times 27$	$3 \times 3$	2	valid	–
C1	Convolution	96	$55 \times 55$	$11 \times 11$	4	valid	ReLU
In	Input	3 (RGB)	$227 \times 227$	–	–	–	–

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.

## 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 12-13](#)). To do this, you can use tools available in `torchvision.transforms.v2` (e.g., `RandomCrop`, `RandomRotation`, etc.). This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. You can similarly use transforms to tweak the colors, and contrasts to simulate many different lighting conditions. In general, you can also flip the pictures horizontally (except for text and other asymmetrical objects). By combining these transformations (using `Compose`), you can greatly increase your training set size.

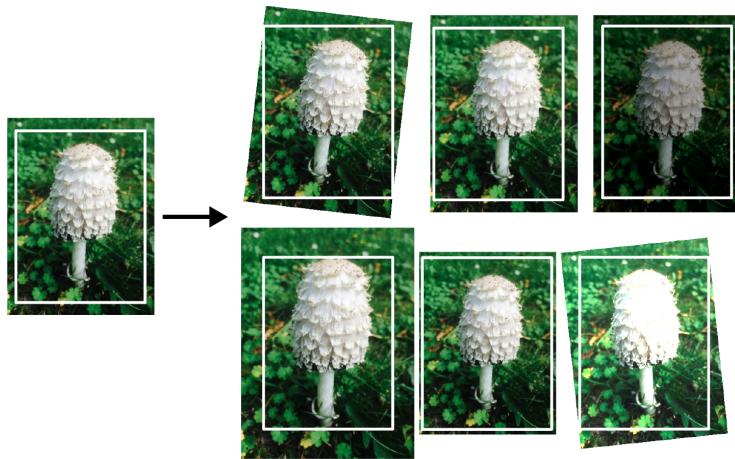


Figure 12-13. Generating new training instances from existing ones

Data augmentation is also useful when you have an unbalanced dataset: you can use it to generate more samples of the less frequent classes. This is called the *synthetic minority oversampling technique*, or SMOTE for short.

Lastly, although data augmentation is typically used only during training, one exception is *test-time augmentation* (TTA): this technique involves augmenting the test data and combining the predictions to boost accuracy. For example, if three augmented versions of an image are classified as a bus, while seven are classified as a truck, then it's probably a truck.

AlexNet also used a regularization technique 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. However, this technique was mostly superseded by simpler and more efficient regularization techniques, especially batch normalization.

A variant of AlexNet called *ZFNet*<sup>10</sup> 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.).

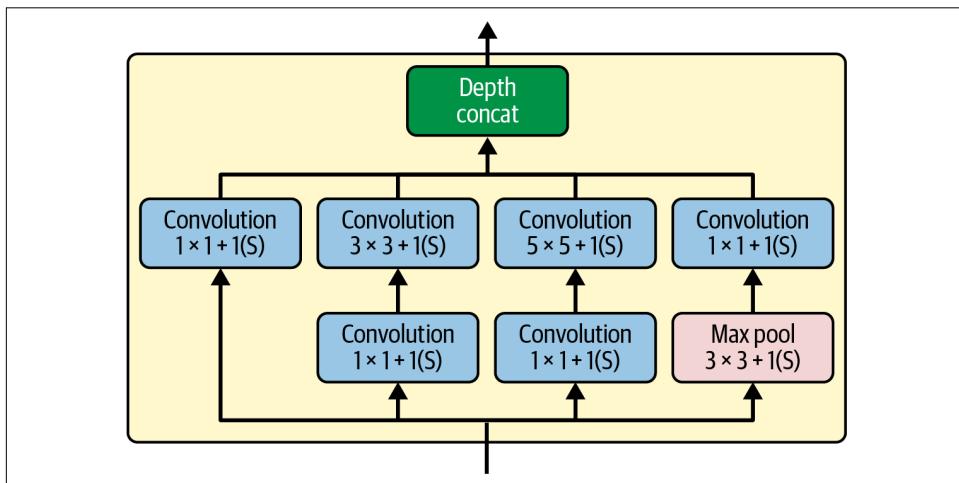
---

<sup>10</sup> Matthew D. Zeiler and Rob Fergus, “Visualizing and Understanding Convolutional Networks”, *Proceedings of the European Conference on Computer Vision* (2014): 818–833.

## GoogLeNet

The **GoogLeNet architecture** was developed by Christian Szegedy et al. from Google Research,<sup>11</sup> 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 12-15](#)). This was made possible by subnetworks called *inception modules*,<sup>12</sup> 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 12-14](#) shows the architecture of an inception module. The notation “ $3 \times 3 + 1(S)$ ” means that the layer uses a  $3 \times 3$  kernel, stride 1, and "same" padding. The input signal is first fed to four different layers in parallel. All convolutional layers use the ReLU activation function. Note that the top convolutional layers use different kernel sizes ( $1 \times 1$ ,  $3 \times 3$ , and  $5 \times 5$ ), 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., it concatenates the multiple feature maps output by each of the upper four convolutional layers). It can be implemented using the `torch.cat()` function, with `dim=1`.



*Figure 12-14. Inception module*

<sup>11</sup> Christian Szegedy et al., “Going Deeper with Convolutions”, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 1–9.

<sup>12</sup> In the 2010 movie *Inception*, the characters keep going deeper and deeper into multiple layers of dreams; hence the name of these modules.

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

- Although they cannot capture spatial patterns, they can capture patterns along the depth dimension (i.e., across channels).
- They are configured to output fewer feature maps than their inputs, so they serve as *bottleneck layers*, meaning they reduce dimensionality. This cuts the computational 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. A convolutional layer is equivalent to sweeping a dense layer across the image (at each location, it only looks at a small receptive field), and these pairs of convolutional layers are equivalent to sweeping two-layer neural networks 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.

Now let's look at the architecture of the GoogLeNet CNN (see [Figure 12-15](#)). 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 12-14](#)). Note that all the convolutional layers use the ReLU activation function.

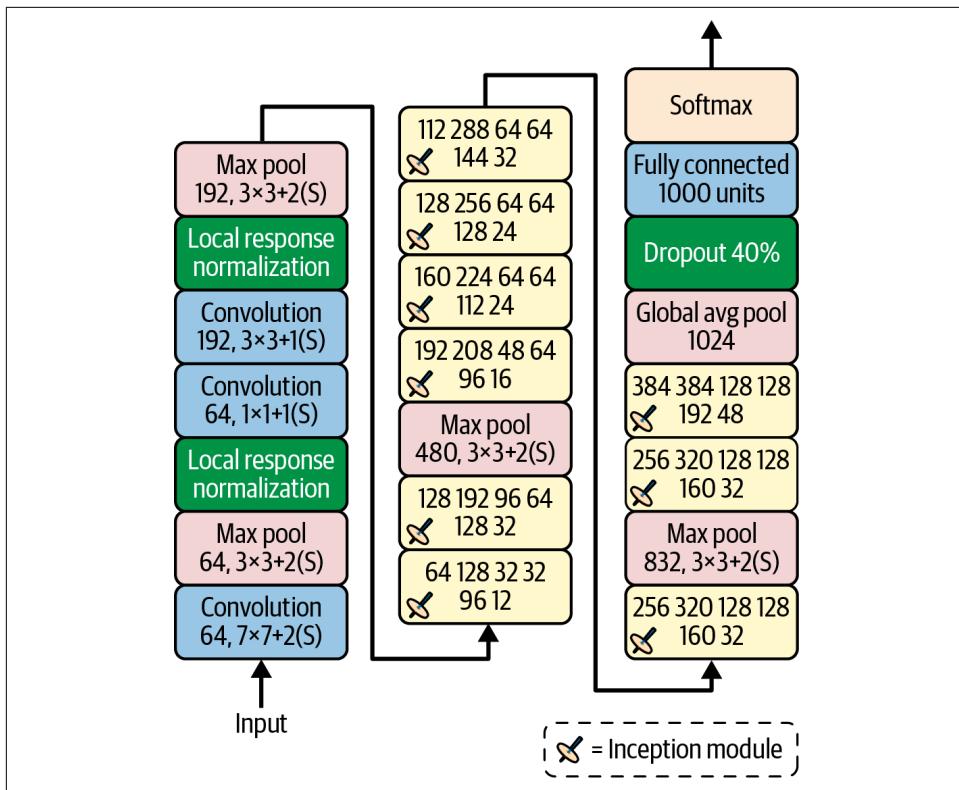


Figure 12-15. GoogLeNet architecture

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,  $7 \times 7$ , 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 mentioned, 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 CNN's *backbone*: a tall stack of nine inception modules, interleaved with a couple of 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 is not much spatial information left at that point. Indeed, GoogLeNet input images are typically expected to be  $224 \times 224$  pixels, so after 5 max pooling layers, each dividing the height and width by 2, the feature maps are down to  $7 \times 7$ . Moreover, this is a classification task, not localization, so it doesn't 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.

The original GoogLeNet architecture 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, but 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 to reach even better performance.

## ResNet

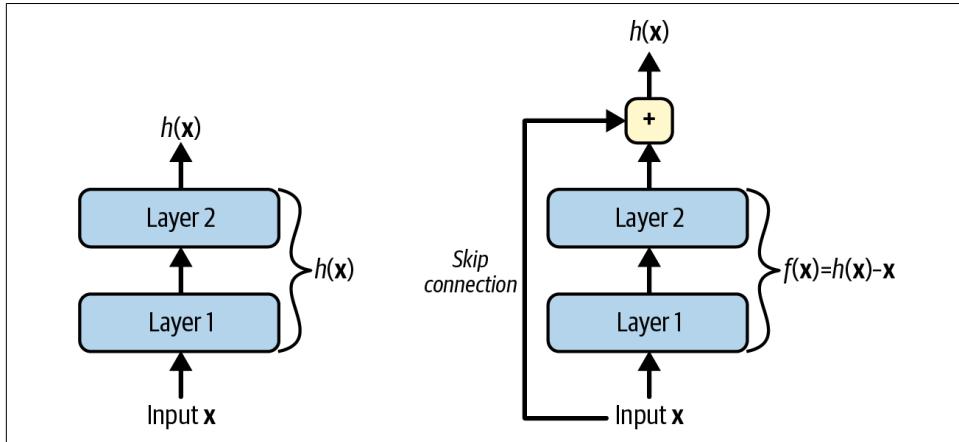
Kaiming He et al. won the ILSVRC 2015 challenge using a **Residual Network (ResNet)**<sup>13</sup> that delivered an astounding top-five error rate under 3.6%. The winning variant used an extremely deep CNN composed of 152 layers (other variants had 34, 50, and 101 layers). It confirmed the general trend: computer vision models were 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 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),

---

<sup>13</sup> Kaiming He et al., "Deep Residual Learning for Image Recognition", arXiv preprint arXiv:1512:03385 (2015).

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 12-16](#)).



*Figure 12-16. Residual learning*

When you initialize a neural network, its weights are close to zero, so a regular network just outputs values close to zero when training starts. But if you add a skip connection, the resulting network outputs a copy of its inputs; in other words, it acts as the identity function at the start of training. 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 12-17](#)). 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.

Now let's look at ResNet's architecture (see [Figure 12-18](#)). 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 residual units. Each residual unit is composed of two convolutional layers (and no pooling layer!), with batch normalization (BN) and ReLU activation, using  $3 \times 3$  kernels and preserving spatial dimensions (stride 1, "same" padding).

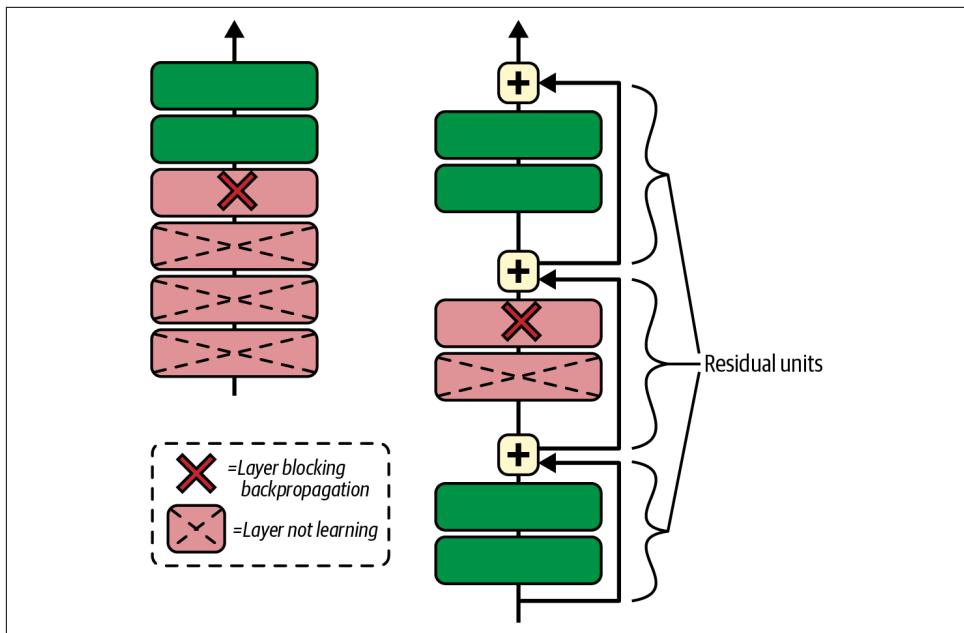


Figure 12-17. Regular deep neural network (left) and deep residual network (right)

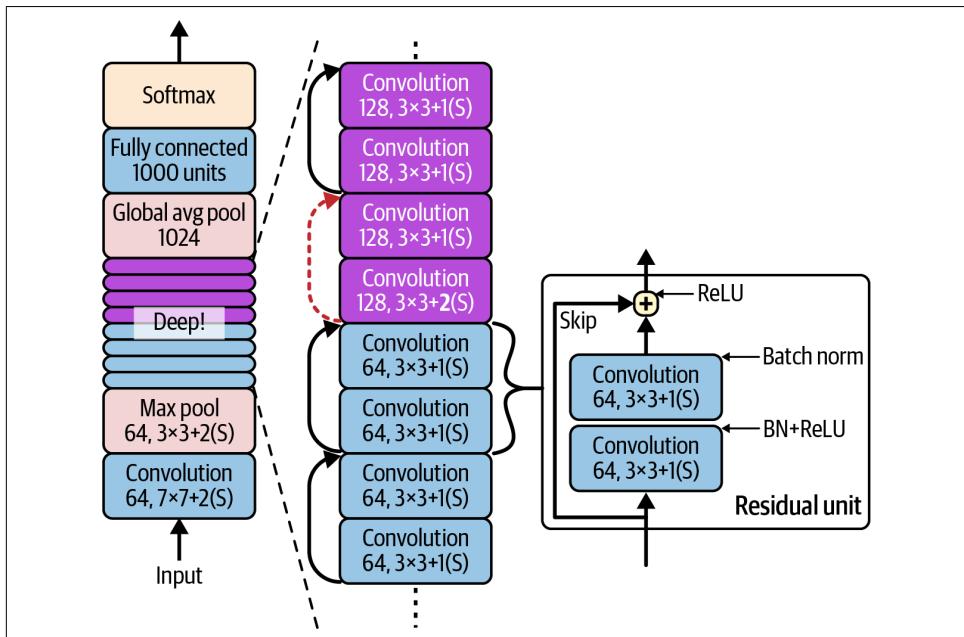
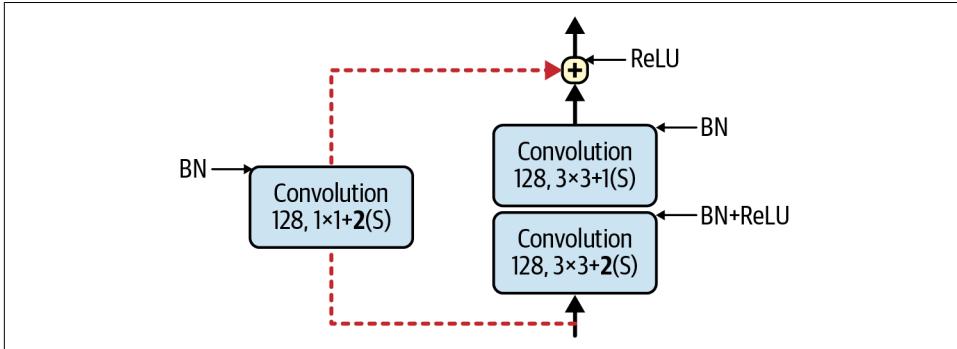


Figure 12-18. ResNet architecture

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 12-18](#)). To solve this problem, the inputs are passed through a  $1 \times 1$  convolutional layer with stride 2 and the right number of output feature maps (see [Figure 12-19](#)).



*Figure 12-19. Skip connection when changing feature map size and depth*



During training, for each mini-batch, you can skip a random set of residual units. This *stochastic depth* technique<sup>14</sup> speeds up training considerably without compromising accuracy. You can implement it using the `torchvision.ops.stochastic_depth()` function.

Different variations of the architecture exist, with different numbers of layers. ResNet-34 is a ResNet with 34 layers (only counting the convolutional layers and the fully connected layer)<sup>15</sup> containing 3 RUs 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.

<sup>14</sup> Gao Huang, Yu Sun, et al., “Deep Networks with Stochastic Depth”, arXiv preprint arXiv:1603.09382 (2016).

<sup>15</sup> It is a common practice when describing a neural network to count only layers with parameters.

ResNets deeper than that, such as ResNet-152, use slightly different residual units. Instead of two  $3 \times 3$  convolutional layers with, say, 256 feature maps, they use three convolutional layers: first a  $1 \times 1$  convolutional layer with just 64 feature maps (4 times less), which acts as a bottleneck layer (as discussed already), then a  $3 \times 3$  layer with 64 feature maps, and finally another  $1 \times 1$  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 architecture](#)<sup>16</sup> merged the ideas of GoogLeNet and ResNet and achieved a top-five error rate of close to 3% on ImageNet classification.

## Xception

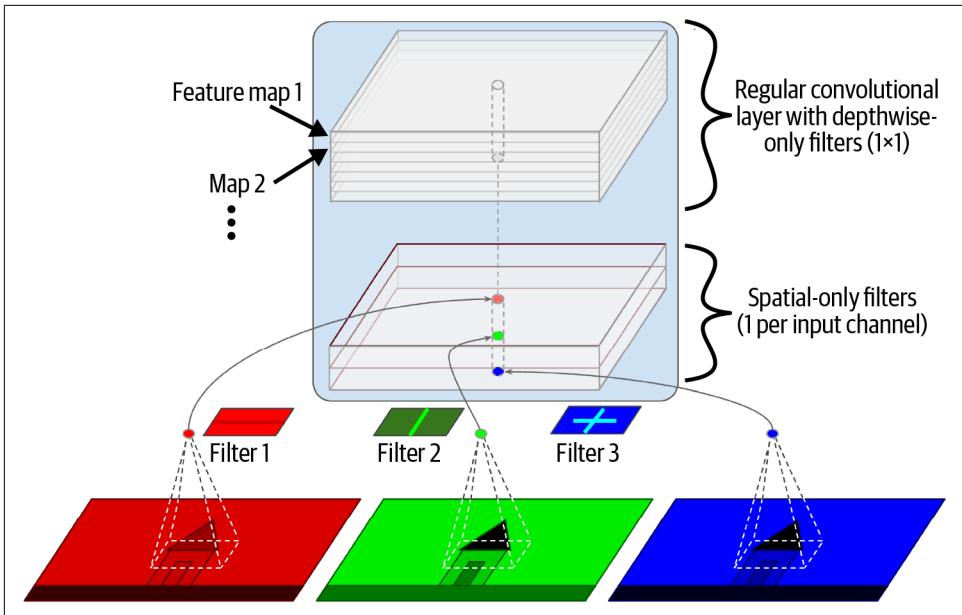
Another variant of the GoogLeNet architecture is worth noting: [Xception](#)<sup>17</sup> (which stands for *Extreme Inception*) was proposed in 2016 by François Chollet (the author of the deep learning framework Keras), and it significantly outperformed Inception-v3 on a huge vision task (350 million 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 short<sup>18</sup>). 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 12-20](#)). Thus, it is composed of two parts: the first part applies a single spatial filter to each input feature map, then the second part looks exclusively for cross-channel patterns—it is just a regular convolutional layer with  $1 \times 1$  filters.

---

<sup>16</sup> Christian Szegedy et al., “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”, arXiv preprint arXiv:1602.07261 (2016).

<sup>17</sup> François Chollet, “Xception: Deep Learning with Depthwise Separable Convolutions”, arXiv preprint arXiv:1610.02357 (2016).

<sup>18</sup> This name can sometimes be ambiguous, since spatially separable convolutions are often called “separable convolutions” as well.



*Figure 12-20. 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 12-20](#) 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 modules at all. Well, as discussed earlier, an inception module contains convolutional layers with  $1 \times 1$  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 often perform better.

PyTorch does not include a `SeparableConv2d` module, but it's fairly straightforward to implement your own:

```
class SeparableConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0):
        super().__init__()
        self.depthwise_conv = nn.Conv2d(
            in_channels, in_channels, kernel_size, stride=stride,
            padding=padding, groups=in_channels)
        self.pointwise_conv = nn.Conv2d(
            in_channels, out_channels, kernel_size=1, stride=1, padding=0)

    def forward(self, inputs):
        return self.pointwise_conv(self.depthwise_conv(inputs))
```

Notice the `groups` argument on the seventh line: it lets you split the input channels into the given number of independent groups, each with its own filters (note that `in_channels` and `out_channels` need to be divisible by `groups`). By default `groups=1`, giving you a normal convolutional layer, but if you set both `groups=in_channels` and `out_channels=in_channels`, you get a depthwise convolutional layer, with one filter per input channel. That's the first layer in the separable convolutional layer. The second is a regular convolutional layer, except we set its kernel size and stride to 1. And that's it!



Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and they often perform better. Consider using them by default, except after layers with few channels (such as the input channel).

## SENet

The winning architecture in the ILSVRC 2017 challenge was the [Squeeze-and-Excitation Network \(SENet\)](#).<sup>19</sup> 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 inception module or residual unit in the original architecture, as shown in [Figure 12-21](#).

---

<sup>19</sup> Jie Hu et al., “Squeeze-and-Excitation Networks”, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 7132–7141.

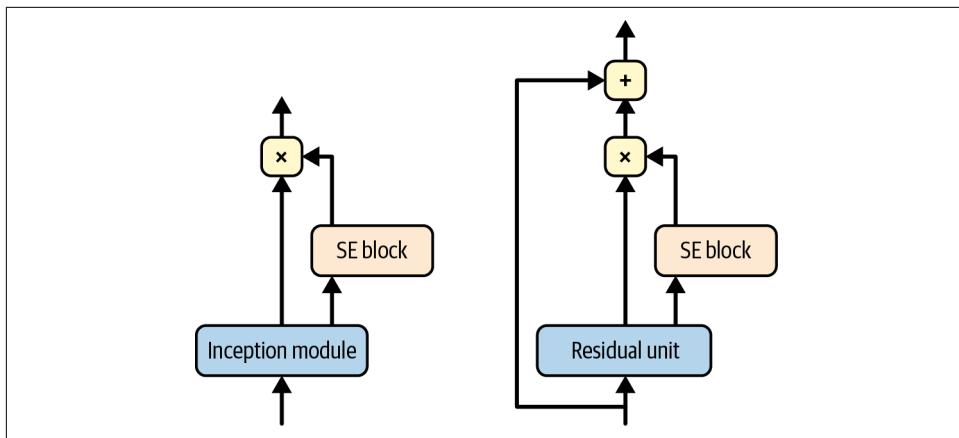


Figure 12-21. SE-Inception module (left) and SE-ResNet unit (right)

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 12-22. 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.

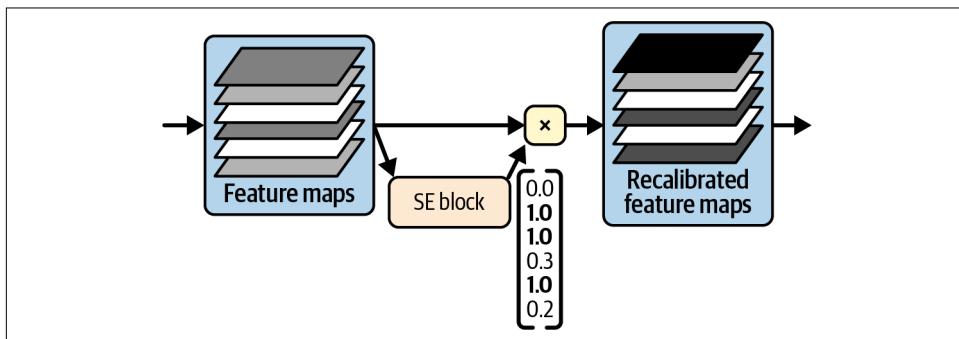


Figure 12-22. An SE block performs feature map recalibration

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 12-23).

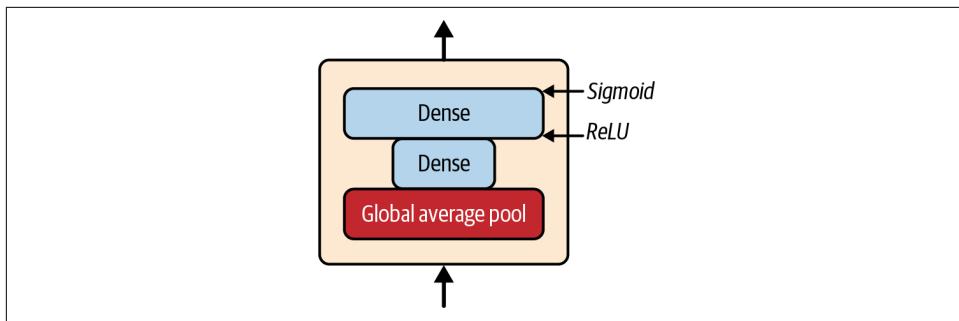


Figure 12-23. SE block architecture

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 numbers representing the overall level of response for each filter. The next layer is where the “squeeze” happens: this layer has significantly fewer than 256 neurons—typically 16 times fewer than the number of feature maps (e.g., 16 neurons)—so the 256 numbers get compressed into a small vector (e.g., 16 dimensions). This is a low-dimensional vector representation (i.e., an embedding) of the distribution of feature responses. This bottleneck step forces the SE block to learn a general representation of the feature combinations (we will see this principle in action again when we discuss autoencoders in [Chapter 18](#)). Finally, the output layer takes the embedding and outputs a recalibration vector containing one number per feature map (e.g., 256), each between 0 and 1. The feature maps are then multiplied by this recalibration vector, so irrelevant features (with a low recalibration score) get scaled down while relevant features (with a recalibration score close to 1) are left alone.

## Other Noteworthy Architectures

There are many other CNN architectures to explore. Here’s a brief overview of some of the most noteworthy:

### VGGNet<sup>20</sup>

VGGNet was the runner-up in the ILSVRC 2014 challenge. Karen Simonyan and Andrew Zisserman, from the Visual Geometry Group (VGG) research lab at Oxford University, developed a very simple and classical architecture; it had 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 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 small  $3 \times 3$  filters, but it had many of them.

---

<sup>20</sup> Karen Simonyan and Andrew Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, arXiv preprint arXiv:1409.1556 (2014).

### *ResNeXt<sup>21</sup>*

ResNeXt improves the residual units in ResNet. Whereas the residual units in the best ResNet models just contain 3 convolutional layers each, the ResNeXt residual units are composed of many parallel stacks (e.g., 32 stacks), with 3 convolutional layers each. However, the first two layers in each stack only use a few filters (e.g., just four), so the overall number of parameters remains the same as in ResNet. Then the outputs of all the stacks are added together, and the result is passed to the next residual unit (along with the skip connection).

### *DenseNet<sup>22</sup>*

A DenseNet is composed of several dense blocks, each made up of a few densely connected convolutional layers. This architecture achieved excellent accuracy while using comparatively few parameters. What does “densely connected” mean? The output of each layer is fed as input to every layer after it within the same block. For example, layer four in a block takes as input the depthwise concatenation of the outputs of layers one, two, and three in that block. Dense blocks are separated by a few transition layers.

### *MobileNet<sup>23</sup>*

MobileNets are streamlined models designed to be lightweight and fast, making them popular in mobile and web applications. They are based on depthwise separable convolutional layers, like Xception. The authors proposed several variants, trading a bit of accuracy for faster and smaller models. Several other CNN architectures are available for mobile devices, such as SqueezeNet, ShuffleNet, or MNasNet.

### *CSPNet<sup>24</sup>*

A Cross Stage Partial Network (CSPNet) is similar to a DenseNet, but part of each dense block’s input is concatenated directly to that block’s output, without going through the block.

### *EfficientNet<sup>25</sup>*

EfficientNet is arguably the most important model in this list. The authors proposed a method to scale any CNN efficiently by jointly increasing the depth

---

<sup>21</sup> Saining Xie et al., “Aggregated Residual Transformations for Deep Neural Networks”, arXiv preprint arXiv:1611.05431 (2016).

<sup>22</sup> Gao Huang et al., “Densely Connected Convolutional Networks”, arXiv preprint arXiv:1608.06993 (2016).

<sup>23</sup> Andrew G. Howard et al., “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”, arXiv preprint arXiv:1704.04861 (2017).

<sup>24</sup> Chien-Yao Wang et al., “CSPNet: A New Backbone That Can Enhance Learning Capability of CNN”, arXiv preprint arXiv:1911.11929 (2019).

<sup>25</sup> Mingxing Tan and Quoc V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”, arXiv preprint arXiv:1905.11946 (2019).

(number of layers), width (number of filters per layer), and resolution (size of the input image) in a principled way. This is called *compound scaling*. They used neural architecture search to find a good architecture for a scaled-down version of ImageNet (with smaller and fewer images), and then used compound scaling to create larger and larger versions of this architecture. When EfficientNet models came out, they vastly outperformed all existing models, across all compute budgets, and they remain among the best models out there today. The authors published a follow-up paper in 2021, introducing EfficientNetV2, which improved training time and parameter efficiency even further.

### *ConvNeXt*<sup>26</sup>

ConvNeXt is quite similar to ResNet, but with a number of tweaks inspired from the most successful vision transformer architectures (see [Chapter 16](#)), such as using large kernels (e.g.,  $7 \times 7$  instead of  $3 \times 3$ ), using fewer activation functions and normalization layers in each residual unit, and more.

Understanding EfficientNet's compound scaling method is helpful to gain a deeper understanding of CNNs, especially if you ever need to scale a CNN architecture. It is based on a logarithmic measure of the compute budget, denoted  $\phi$ : if your compute budget doubles, then  $\phi$  increases by 1. In other words, the number of floating-point operations available for training is proportional to  $2^\phi$ . Your CNN architecture's depth, width, and resolution should scale as  $\alpha^\phi$ ,  $\beta^\phi$ , and  $\gamma^\phi$ , respectively. The factors  $\alpha$ ,  $\beta$ , and  $\gamma$  must be greater than 1, and  $\alpha\beta^2\gamma^2$  should be close to 2. The optimal values for these factors depend on the CNN's architecture. To find the optimal values for the EfficientNet architecture, the authors started with a small baseline model (EfficientNetB0), fixed  $\phi = 1$ , and simply ran a grid search: they found  $\alpha = 1.2$ ,  $\beta = 1.1$ , and  $\gamma = 1.1$ . They then used these factors to create several larger architectures, named EfficientNetB1 to EfficientNetB7, for increasing values of  $\phi$ .

I hope you enjoyed this deep dive into the main CNN architectures! But how do you choose the right one?

## Choosing the Right CNN Architecture

As you might expect, the best architecture depends on what matters most for your project: Accuracy? Model size (e.g., for deployment to a mobile device)? Inference speed? Energy consumption? [Table 12-3](#) lists some of the pretrained classification models currently available in TorchVision (you'll see how to use them later in this chapter). You can find the full list at <https://pytorch.org/vision/stable/models> (including models for other computer vision tasks). The table shows each model's top-1 and top-5 accuracy on the ImageNet dataset, its number of parameters (in millions), and

---

<sup>26</sup> Zhuang Liu et al, "A ConvNet for the 2020s", arXiv preprint arXiv:2201.03545 (2022).

how much compute it requires for each image (measured in GFLOPs: a Giga-FLOP is one billion floating-point operations). As you can see, larger models are generally more accurate, but not always; for example, the small variant of EfficientNet v2 outperforms Inception v3 both in size and accuracy (but not in compute).

*Table 12-3. Some of the pretrained models available in TorchVision, sorted by size*

Class name	Top-1 acc	Top-5 acc	Params	GFLOPs
MobileNet v3 small	67.7%	87.4%	2.5M	0.1
EfficientNet B0	77.7%	93.5%	5.3M	0.4
GoogLeNet	69.8%	89.5%	6.6M	1.5
DenseNet 121	74.4%	92.0%	8.0M	2.8
EfficientNet v2 small	84.2%	96.9%	21.5M	8.4
ResNet 34	73.3%	91.4%	21.8M	3.7
Inception V3	77.3%	93.5%	27.2M	5.7
ConvNeXt Tiny	82.6%	96.1%	28.6M	4.5
DenseNet 161	77.1%	93.6%	28.7M	7.7
ResNet 152	82.3%	96.0%	60.2M	11.5
AlexNet	56.5%	79.1%	61.1M	0.7
EfficientNet B7	84.1%	96.9%	66.3M	37.8
ResNeXt 101 32x8D	82.8%	96.2%	88.8M	16.4
EfficientNet v2 large	85.8%	97.8%	118.5M	56.1
VGG 11 with BN	70.4%	89.8%	132.9M	7.6
ConvNeXt Large	84.4%	97.0%	197.8M	34.4

The smaller models will run on any GPU, but what about a large model, such as ConvNeXt Large? Since each parameter is represented as a 32-bit float (4 bytes), you might think you just need 800 MB of RAM to run a 200M parameter model, but you actually need *much* more, typically 5 GB per image at inference time (depending on the image size), and even more at training time. Let's see why.

## GPU RAM Requirements: Inference Versus Training

CNNs need a *lot* of RAM. For example, consider a single convolutional layer with 200  $5 \times 5$  filters, stride 1 and "same" padding, processing a  $150 \times 100$  RGB image (3 channels):

- The number of parameters is  $(5 \times 5 \times 3 + 1) \times 200 = 15,200$  (the + 1 corresponds to the bias terms). That's not much: to produce the same size outputs, a fully connected layer would need  $200 \times 150 \times 100$  neurons, each connected to all  $150 \times 100 \times 3$  inputs. It would have  $200 \times 150 \times 100 \times (150 \times 100 \times 3 + 1) \approx 135$  billion parameters!

- However, each of the 200 feature maps contains  $150 \times 100$  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.
- Importantly, the convolutional layer's output will occupy  $200 \times 150 \times 100 \times 32 = 96$  million bits (12 MB) of RAM, assuming we're using 32-bit floats.<sup>27</sup> And that's just for one instance—if a training batch contains 100 instances, then this single convolutional 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 backward pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers. You can easily run out of GPU RAM.

If training crashes because of an out-of-memory error, you can try reducing the batch size. To still get some of the benefits of large batches, you can accumulate the gradients after each batch, and only update the model weights every few batches. Alternatively, you can try reducing dimensionality using a stride, removing a few layers, using 16-bit floats instead of 32-bit floats, distributing the CNN across multiple devices, or offloading the most memory-hungry modules to the CPU (using `module.to("cpu")`).

Yet another option is to trade more compute in exchange for a lower memory usage. For example, instead of saving all of the activations during the forward pass, you can save some of them, called *activation checkpoints*, then during the backward pass, you can recompute the missing activations as needed by running a partial forward pass starting from the previous checkpoint.

To implement activation checkpointing (also called *gradient checkpointing*) in PyTorch, you can use the `torch.utils.checkpoint.checkpoint()` function: instead of calling a module `z = foo(x)`, you can call it using `z = checkpoint(foo, x)`. During inference, it will make no difference, but during training this module's activations will no longer be saved during the forward pass, and `foo(x)` will be recomputed during the backward pass when needed. This approach is fairly simple to implement, and it doesn't require any changes to your model architecture.

---

<sup>27</sup> 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. And 1 MiB = 1,024 kB =  $1,024 \times 1,024$  bytes. So 12 MB  $\approx$  11.44 MiB.



The forward pass needs to produce the same result if you call it twice with the same inputs, or else the gradients will be incorrect. This means that custom modules must respect a few constraints, such as avoiding in-place ops or using controlled states for random number generation: please see the `checkpoint()` function's documentation for more details.

That said, if you're OK with tweaking your model architecture, then there's a much more efficient solution you can use to exchange compute for memory: reversible residual networks.

## Reversible Residual Networks (RevNets)

RevNets were proposed by Aidan Gomez et al. in 2017:<sup>28</sup> they typically only increase compute by about 33% and actually don't require you to save any activations at all during the forward pass! Here's how they work:

- Each layer, called a *reversible layer*, takes two inputs of equal sizes,  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , and computes two outputs:  $\mathbf{y}_1 = \mathbf{x}_1 + f(\mathbf{x}_2)$  and  $\mathbf{y}_2 = g(\mathbf{y}_1) + \mathbf{x}_2$ , where  $f$  and  $g$  can be any functions, as long as the output size equals the input size, and as long as they always produce the same output for a given input. For example,  $f$  and  $g$  can be identical modules composed of a few convolutional layers with stride 1 and "same" padding (each convolutional layer comes with its own batch-norm and ReLU activation).
- During backpropagation, the inputs of each reversible layer can be recomputed from the outputs whenever needed, using:  $\mathbf{x}_2 = \mathbf{y}_2 - g(\mathbf{y}_1)$  and  $\mathbf{x}_1 = \mathbf{y}_1 - f(\mathbf{x}_2)$  (you can easily verify that these two equalities follow directly from the first two). No need to store any activations during the forward pass: brilliant!

Since  $f$  and  $g$  must output the same shape as the input, reversible layers cannot contain convolutional layers with a stride greater than 1, or with "valid" padding. You can still use such layers in your CNN, but the RevNet trick won't be applicable to them, so you will have to save their activations during the forward pass; luckily, a CNN usually requires only a handful of such layers. This includes the very first layer, which reduces the spatial dimensions and increases the number of channels: the result can be split in two equal parts along the channel dimension and fed to the first reversible layer.

RevNets aren't limited to CNNs. In fact, they are at the heart of an influential Transformer architecture named Reformer (see [Chapter 17](#)).

---

<sup>28</sup> Aidan Gomez et al., "The Reversible Residual Network: Backpropagation Without Storing Activations", arXiv preprint arXiv:1707.04585 (2017).

OK, it's now time to get our hands dirty! Let's implement one of the most popular CNN architectures from scratch using PyTorch.

## Implementing a ResNet-34 CNN Using PyTorch

Most CNN architectures described so far can be implemented pretty naturally using PyTorch (although generally you would load a pretrained network instead, as you will see). To illustrate the process, let's implement a ResNet-34 from scratch with PyTorch. First, we'll create a `ResidualUnit` layer:

```
class ResidualUnit(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        DefaultConv2d = partial(
            nn.Conv2d, kernel_size=3, stride=1, padding=1, bias=False)
        self.main_layers = nn.Sequential(
            DefaultConv2d(in_channels, out_channels, stride=stride),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            DefaultConv2d(out_channels, out_channels),
            nn.BatchNorm2d(out_channels),
        )
        if stride > 1:
            self.skip_connection = nn.Sequential(
                DefaultConv2d(in_channels, out_channels, kernel_size=1,
                             stride=stride, padding=0),
                nn.BatchNorm2d(out_channels),
            )
        else:
            self.skip_connection = nn.Identity()

    def forward(self, inputs):
        return F.relu(self.main_layers(inputs) + self.skip_connection(inputs))
```

As you can see, this code matches [Figure 12-19](#) pretty closely. In the constructor, we create all the layers we need: the main layers are the ones on the righthand side of the figure, and the skip connection corresponds to the layers on the left when the stride is greater than 1, or an `nn.Identity` module when the stride is 1—the `nn.Identity` module does nothing at all, it just returns its inputs. Then in the `forward()` method, we make the inputs go through both the main layers and the skip connection, then we add both outputs and apply the activation function.

Next, let's build our `ResNet34` module! Now that we have our `ResidualUnit` module, the whole ResNet-34 architecture becomes one big stack of modules, so we can base our `ResNet34` class on a single `nn.Sequential` module. The code closely matches [Figure 12-18](#):

```

class ResNet34(nn.Module):
    def __init__(self):
        super().__init__()
        layers = [
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=7, stride=2,
                      padding=3, bias=False),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        ]
        prev_filters = 64
        for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
            stride = 1 if filters == prev_filters else 2
            layers.append(ResidualUnit(prev_filters, filters, stride=stride))
            prev_filters = filters
        layers += [
            nn.AdaptiveAvgPool2d(output_size=1),
            nn.Flatten(),
            nn.Linear(10),
        ]
        self.resnet = nn.Sequential(*layers)

    def forward(self, inputs):
        return self.resnet(inputs)

```

The only tricky part in this code is the loop that adds the `ResidualUnit` layers to the list of layers: as explained earlier, the first 3 RUs have 64 filters, then the next 4 RUs have 128 filters, and so on. At each iteration, we must set the stride to 1 when the number of filters is the same as in the previous RU, or else we set it to 2; then we append the `ResidualUnit` to the list, and finally we update `prev_filters`.

And that's it, you could now train this model on ImageNet or any other dataset of  $224 \times 224$  images. It is amazing that in just 45 lines of code, we can build the model that won the ILSVRC 2015 challenge! This demonstrates both the elegance of the ResNet model and the expressiveness of PyTorch (and Python). Implementing the other CNN architectures we discussed would take more time, but it wouldn't be much harder. However, TorchVision comes with several of these architectures built in, so why not use them instead?

## Using TorchVision's Pretrained Models

In general, you won't have to implement standard models like GoogLeNet, ResNet, or ConvNeXt manually, since pretrained networks are readily available with a couple lines of code using TorchVision.



TIMM is another very popular library built on PyTorch: it provides a collection of pretrained image classification models, as well as many related tools such as data loaders, data augmentation utilities, optimizers, schedulers, and more. Hugging Face’s Hub is also a great place to get all sorts of pretrained models (see [Chapter 14](#)).

For example, you can load a ConvNeXt model pretrained on ImageNet with the following code. There are several variants of the ConvNeXt model—tiny, small, base, and large—and this code loads the base variant:

```
weights = torchvision.models.ConvNeXt_Base_Weights.IMAGENET1K_V1
model = torchvision.models.convnext_base(weights=weights).to(device)
```

That’s all! This code automatically downloads the weights (338 MB) from the *Torch Hub*, an online repository of pretrained models. The weights are saved and cached for future use (e.g., in `~/.cache/torch/hub`; run `torch.hub.get_dir()` to find the exact path on your system). Some models have newer weights versions (e.g., `IMAGENET1K_V2`) or other weight variants. For the full list of available models, run `torchvision.models.list_models()`. To find the list of pretrained weights available for a given model, such as `convnext_base`, run `list(torchvision.models.get_model_weights("convnext_base"))`. Alternatively, visit <https://pytorch.org/vision/main/models>.

Let’s use this model to classify the two sample images we loaded earlier. Before we can do this, we must first ensure that the images are preprocessed exactly as the model expects. In particular, they must have the right size. A ConvNeXt model expects  $224 \times 224$  pixel images (other models may expect other sizes, such as  $299 \times 299$ ). Since our sample images are  $427 \times 640$  pixels, we need to resize them. We could do this using TorchVision’s `CenterCrop` and/or `Resize` transform, but it’s much easier and safer to use the transforms returned by `weights.transforms()`, as they are specifically designed for this particular pretrained model:

```
transforms = weights.transforms()
preprocessed_images = transforms(sample_images_permuted)
```

Importantly, these transforms also normalize the pixel intensities just like during training. In this case, the transforms standardize the pixel intensities separately for each color channel, using ImageNet’s means and standard deviations for each channel (we will see how to do this manually later in this chapter).

Next we can move the images to the GPU and pass them to the model. As always, remember to switch the model to evaluation mode before making predictions—the model is in training mode by default—and also turn off autograd:

```
model.eval()
with torch.no_grad():
    y_logits = model(preprocessed_images.to(device))
```

The result is a  $2 \times 1,000$  tensor containing the class logits for each image (recall that ImageNet has 1,000 classes). As we did in [Chapter 10](#), we can use `torch.argmax()` to get the predicted class for each image (i.e., the class with the maximum logit):

```
>>> y_pred = torch.argmax(y_logits, dim=1)
>>> y_pred
tensor([698, 985], device='cuda:0')
```

So far, so good, but what exactly do these classes represent? Well you could find the ImageNet class names online, but once again it's simpler and safer to get the class names directly from the `weights` object. Indeed, its `meta` attribute is a dictionary containing metadata about the pretrained model, including the class names:

```
>>> class_names = weights.meta["categories"]
>>> [class_names[class_id] for class_id in y_pred]
['palace', 'daisy']
```

There you have it: the first image is classified as a palace, and the second as a daisy. Since the ImageNet dataset does not have classes for Chinese towers or dahlia flowers, a palace and a daisy are reasonable substitutes (the tower is part of the Summer Palace in Beijing). Let's look at the top-three predictions using `topk()`:

```
>>> y_top3_logits, y_top3_class_ids = y_logits.topk(k=3, dim=1)
>>> [[class_names[class_id] for class_id in top3] for top3 in y_top3_class_ids]
[['palace', 'monastery', 'lakeside'], ['daisy', 'pot', 'ant']]
```

Let's look at the estimated probabilities for each of these classes:

```
>>> y_top3_logits.softmax(dim=1)
tensor([[0.8618, 0.1185, 0.0197],
       [0.8106, 0.0964, 0.0930]], device='cuda:0')
```

As you can see, TorchVision makes it easy to download and use pretrained models, and it works quite well out of the box for ImageNet classes. But what if you need to classify images into classes that don't belong to the ImageNet dataset, such as various flower species? In that case, you may still benefit from the pretrained models by using them to perform transfer learning.

## Pretrained Models for Transfer Learning

If you want to build an image classifier but you do not have enough data to train it from scratch, then it is often a good idea to reuse the lower layers of a pretrained model, as we discussed in [Chapter 11](#). In this section we will reuse the ConvNeXt model we loaded earlier—which was pretrained on ImageNet—and after replacing its classification head, we will fine-tune it on the [102 Category Flower Dataset](#)<sup>29</sup>

---

<sup>29</sup> M. Nilsback and A. Zisserman, “Automated Flower Classification over a Large Number of Classes”, *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing* (2008).

(Flowers102 for short). This dataset only contains 10 images per class, and there are 102 classes in total (as the name indicates), so if you try to train a model from scratch, you will really struggle to get high accuracy. However, it's quite easy to get over 90% accuracy using a good pretrained model. Let's see how. First, let's download the dataset using Torchvision:

```
DefaultFlowers102 = partial(torchvision.datasets.Flowers102, root="datasets",
                           transform=weights.transforms(), download=True)
train_set = DefaultFlowers102(split="train")
valid_set = DefaultFlowers102(split="val")
test_set = DefaultFlowers102(split="test")
```

This code uses `partial()` to avoid repeating the same arguments three times. We also set `transform=weights.transforms()` to preprocess the images immediately when they are loaded. The Flowers102 dataset comes with three predefined splits, for training, validation, and testing. The first two have 10 images per class, but surprisingly the test set has many more (it has a variable number of images per class, between 20 and 238). In a real project, you would normally use most of your data for training rather than for testing, but this dataset was designed for computer vision research, and the authors purposely restricted the training set and the validation set.

We then create the data loaders, as usual:

```
from torch.utils.data import DataLoader

train_loader = DataLoader(train_set, batch_size=32, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=32)
test_loader = DataLoader(test_set, batch_size=32)
```

Many TorchVision datasets conveniently contain the class names in the `classes` attribute, but sadly not this dataset.<sup>30</sup> If you prefer to see lovely names like “tiger lily”, “monkshood”, or “snapdragon” rather than boring class IDs, then you need to manually define the list of class names:

```
class_names = ['pink primrose', ..., 'trumpet creeper', 'blackberry lily']
```

Now let's adapt our pretrained ConvNeXt-base model to this dataset. Since it was pretrained on ImageNet, which has 1,000 classes, the model's head (i.e., its upper layers) was designed to output 1,000 logits. But we only have 102 classes, so we must chop the model's head off and replace it with a smaller one. But how can we find it? Well let's use the model's `named_children()` method to find the name of its submodules:

```
>>> [name for name, child in model.named_children()]
['features', 'avgpool', 'classifier']
```

---

<sup>30</sup> TorchVision PR #8838 might have fixed this by the time you read these lines.

The `features` module is the main part of the model, which includes all layers except for the global average pooling layer (`avgpool`) and the model's head (`classifier`). Let's look more closely at the head:

```
>>> model.classifier
Sequential(
    (0): LayerNorm2d((1024,), eps=1e-06, elementwise_affine=True)
    (1): Flatten(start_dim=1, end_dim=-1)
    (2): Linear(in_features=1024, out_features=1000, bias=True)
)
```

As you can see, it's an `nn.Sequential` module composed of a layer normalization layer, an `nn.Flatten` layer, and an `nn.Linear` layer with 1,024 inputs and 1,000 outputs. This `nn.Linear` layer is the output layer, and it's the one we need to replace. We must only change the number of outputs:

```
n_classes = 102 # len(class_names) == 102
model.classifier[2] = nn.Linear(1024, n_classes).to(device)
```

As explained in [Chapter 11](#), it's usually a good idea to freeze the weights of the pretrained layers, at least at the beginning of training. We can do this by freezing every single parameter in the model, and then unfreezing only the parameters of the head:

```
for param in model.parameters():
    param.requires_grad = False

for param in model.classifier.parameters():
    param.requires_grad = True
```

Next, you can train this model for a few epochs, and you will already reach about 90% accuracy just by training the new head, without even fine-tuning the pretrained layers. After that, you can unfreeze the whole model, lower the learning rate—typically by a factor of 10—and continue training the model. Give this a try, and see what accuracy you can reach!

To reach an even higher accuracy, it's usually a good idea to perform some data augmentation on the training images. For this, you can try randomly flipping the training images horizontally, randomly rotating them by a small angle, randomly resizing and cropping them, and randomly tweaking their colors. This must all be done before running the ImageNet normalization step, which you can implement using a `Normalize` transform:

```
import torchvision.transforms.v2 as T

transforms = T.Compose([
    T.RandomHorizontalFlip(p=0.5),
    T.RandomRotation(degrees=30),
    T.RandomResizedCrop(size=(224, 224), scale=(0.8, 1.0)),
    T.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
```

```
T.ToImage(),
T.ToDtype(torch.float32, scale=True),
T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```



TorchVision comes with an `AutoAugment` transform which applies multiple augmentation operations optimized for ImageNet. It generalizes well to many other image datasets, and it also offers predefined settings for two other datasets: CIFAR10 and the street view house numbers (SVHN) dataset.

Here are some more ideas to continue to improve your model's accuracy:

- Try other pretrained models.
- Extend the training set: find more flower images and label them.
- Create an ensemble of models, and combine their predictions.
- Analyze failure cases, and see whether they share specific characteristics, such as similar texture or color. You can then try to tweak image preprocessing to address these issues.
- Use a learning schedule such as performance scheduling.
- Unfreeze the layers gradually, starting from the top. Alternatively, you can use *differential learning rates*: apply a smaller learning rate to lower layers, and a larger learning rate to upper layers. You can do this by using parameter groups (see [Chapter 11](#)).
- Explore different optimizers and fine-tune their hyperparameters.
- Try different regularization techniques.



It's worth spending time looking for models that were pretrained on similar images. For example, if you're dealing with satellite images, aerial images, or even raster data such as digital elevation models (DEM), then models pretrained on ImageNet won't help much. Instead, check out Microsoft's *TorchGeo* library, which is similar to TorchVision but for geospatial data. For medical images, check out Project MONAI. For agricultural images, check out AgML. And so on.

With that, you can start training amazing image classifiers on your own images and classes! But there's more to computer vision than just classification. For example, what if you also want to know *where* the flower is in a picture? Let's look at this now.

# Classification and Localization

Localizing an object in a picture can be expressed as a regression task, as discussed in [Chapter 9](#): to predict a bounding box around the object, a common approach is to predict the location of the bounding box's center, as well as its width and height (alternatively, you could predict the horizontal and vertical coordinates of the object's upper-left and lower-right corners). This means we have four numbers to predict. It does not require much change to the ConvNeXt model; we just need to add a second dense output layer with four units (e.g., on top of the global average pooling layer). Here's a `FlowerLocator` model that adds a localization head to a given base model, such as our ConvNeXt model:

```
class FlowerLocator(nn.Module):
    def __init__(self, base_model):
        super().__init__()
        self.base_model = base_model
        self.localization_head = nn.Sequential(
            nn.Flatten(),
            nn.Linear(base_model.classifier[2].in_features, 4)
        )

    def forward(self, X):
        features = self.base_model.features(X)
        pool = self.base_model.avgpool(features)
        logits = self.base_model.classifier(pool)
        bbox = self.localization_head(pool)
        return logits, bbox

torch.manual_seed(42)
locator_model = FlowerLocator(model).to(device)
```

This locator model has two heads: the first outputs class logits, while the second outputs the bounding box. The localization head has the same number of inputs as the `nn.Linear` layer of the classification head, but it outputs just four numbers. The `forward()` method takes a batch of preprocessed images as input and outputs both the predicted class logits (102 per image) and the predicted bounding boxes (1 per image). After training this model, you can use it as follows:

```
preproc_images = [...] # a batch of preprocessed images
y_pred_logits, y_pred_bbox = locator_model(preprocessed_images.to(device))
```

But how can we train this model? Well, we saw how to train a model with two or more outputs in [Chapter 10](#), and this one is no different: in this case, we can use the `nn.CrossEntropyLoss` for the classification head, and the `nn.MSELoss` for the localization head. The final loss can just be a weighted sum of the two. Voilà, that's all there is to it.

Hey, not so fast! We have a problem: the Flowers102 dataset does not include any bounding boxes, so we need to add them ourselves. This is often one of the hardest

and most costly parts of a machine learning project: labeling and annotating the data. It's a good idea to spend time looking for the right tools. To annotate images with bounding boxes, you may want to use an open source labeling tool like Label Studio, OpenLabeler, ImgLab, Labelme, VoTT, or VGG Image Annotator, or perhaps a commercial tool like LabelBox, Supervisely, Roboflow, or RectLabel. Many of these are now AI assisted, greatly speeding up the annotation task. You may also want to consider crowdsourcing platforms such as Amazon Mechanical Turk if you have a very large number of images to annotate. However, it is quite a lot of work to set up a crowdsourcing platform, prepare the form to be sent to the workers, supervise them, and ensure that the quality of the bounding boxes they produce is good, so make sure it is worth the effort. If there are just a few hundred or even a couple of thousand images to label, and you don't plan to do this frequently, it may be preferable to do it yourself: with the right tools, it will only take a few days, and you'll also gain a better understanding of your dataset and task.

You can then create a custom dataset (see [Chapter 10](#)) where each entry contains an image, a label, and a bounding box. TorchVision conveniently includes a `BoundingBoxes` class that represents a list of bounding boxes. For example, the following code creates a bounding box for the largest flower in the first image of the Flowers102 training set (for now we only consider one bounding box per image, but we'll discuss multiple bounding boxes per image later in this chapter):

```
import torchvision.tv_tensors

bbox = torchvision.tv_tensors.BoundingBoxes(
    [[377, 199, 248, 262]], # center x=377, center y=199, width=248, height=262
    format="CXYWH", # other possible formats: "XYXY" and "XYWH"
    canvas_size=(500, 754) # raw image size before preprocessing
)
```



To visualize bounding boxes, use the `torchvision.utils.draw_bounding_boxes()` function. You will first need to convert the bounding boxes to the XYXY format using `torchvision.ops.box_convert()`.

The `BoundingBoxes` class is a subclass of `TVTensor`, which is a subclass of `torch.Tensor`, so you can treat bounding boxes exactly like regular tensors, with extra features. Most importantly, you can transform bounding boxes using TorchVision's transforms API v2. For example, let's use the transform we defined earlier to preprocess this bounding box:

```
>>> transform(bbox)
BoundingBoxes([[ 90,  91, 120, 154]], format=BoundingBoxFormat.CXYWH,
              canvas_size=(224, 224), clamping_mode=soft)
```



Resizing and cropping a bounding box works as expected, but rotation is special: the bounding box can't be rotated since it doesn't have any rotation parameter, so instead it is resized to fit the rotated box (*not* the rotated object). As a result, it may end up being a bit too large for the object.

You can pass a nested data structure to a transform and the output will have the same structure, except with all the images and bounding boxes transformed. For example, the following code transforms the first flower image in the training set and its bounding box, leaving the label unchanged. In this example, the input and output are both 2-tuples containing an image and a dictionary composed of a label and a bounding box, but you could use any other data structure:

```
first_image = [...] # load the first training image without any preprocessing
preproc_image, preproc_target = transform(
    (first_image, {"label": 0, "bbox": bbox}))
)
preproc_bbox = preproc_target["bbox"]
```



When using the MSE, a 10-pixel error for a large bounding box will be penalized just as much as a 10-pixel error for a small bounding box. To avoid this, you can use a custom loss function that computes the square root of the width and height—for both the target and the prediction—before computing the MSE.

The MSE is simple and often works fairly well to train the model, but it is not a great metric to evaluate how well the model can predict bounding boxes. The most common metric for this is the *intersection over union* (IoU, also known as the *Jaccard index*): it is the area of overlap between the target bounding box T and the predicted bounding box P, divided by the area of their union  $P \cup T$  (see [Figure 12-24](#)). In short,  $\text{IoU} = |P \cap T| / |P \cup T|$ , where  $|x|$  is the area of  $x$ . The IoU ranges from 0 (no overlap) to 1 (perfect overlap). It is implemented by the `torchvision.ops.box_iou()` function.

The IoU is not great for training because it is equal to zero whenever P and T have no overlap, regardless of the distance between them or their shapes: in this case the gradient is also equal to zero and therefore gradient descent cannot make any progress. Luckily, it's possible to fix this flaw by incorporating extra information. For example, the *Generalized IoU* (GIoU), introduced in a [2019 paper](#) by H. Rezatofighi et al.,<sup>31</sup> considers the smallest box S that contains both P and T, and it subtracts from the IoU the ratio of S that is not covered by P or T. In short,  $\text{GIoU} = \text{IoU} -$

---

<sup>31</sup> H. Rezatofighi et al., “Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression”, arXiv preprint arXiv:1902.09630 (2019).

$|S - (P \cup T)| / |S|$ . This means that the GIoU gets smaller as P and T get further apart, which gives gradient descent something to play with so it can pull P closer to T. Since we want to maximize the GIoU, the GIoU loss is equal to  $1 - \text{GIoU}$ . This loss quickly became popular, and it is implemented by the `torchvision.ops.generalized_box_iou_loss()` function.

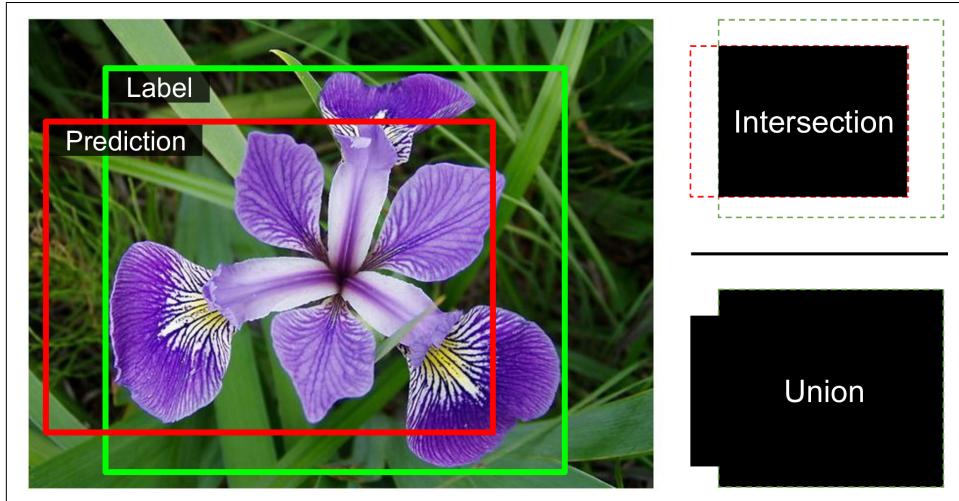


Figure 12-24. IoU metric for bounding boxes

Another important variant of the IoU is the *Complete IoU* (CIoU), introduced in a [2020 paper](#) by Z. Zheng et al.<sup>32</sup> It considers three geometric factors: the IoU (the more overlap, the better), the distance between the centers of P and T (the closer, the better), normalized by the length of the diagonal of S, and the similarity between the aspect ratios of P and T (the closer, the better). The loss is  $1 - \text{CIoU}$ , and it is implemented by the `torchvision.ops.complete_box_iou_loss()` function. It generally performs better than the MSE or the GIoU, converging faster and leading to more accurate bounding boxes, so it is becoming the default loss for localization.

Classifying and localizing a single object is nice, but what if the images contain multiple objects (as is often the case in the flowers dataset)?

<sup>32</sup> Z. Zheng et al., “Enhancing Geometric Factors in Model Learning and Inference for Object Detection and Instance Segmentation”, arXiv preprint arXiv:2005.03572 (2020).

# Object Detection

The task of classifying and localizing multiple objects in an image is called *object detection*. Until a few years ago, a common approach was to take a CNN that was trained to classify and locate a single object roughly centered in the image, then slide this CNN across the image and make predictions at each step. The CNN was generally trained to predict not only class probabilities and a bounding box, but also an *objectness score*: this is the estimated probability that the image does indeed contain an object centered near the middle. This is a binary classification output; it can be produced by a dense output layer with a single unit, using the sigmoid activation function and trained using the binary cross-entropy loss.



Instead of an objectness score, a “no-object” class was sometimes added, but in general this did not work as well. The questions “Is an object present?” and “What type of object is it?” are best answered separately.

This sliding-CNN approach is illustrated in [Figure 12-25](#). In this example, the image was chopped into a  $5 \times 7$  grid, and we see a CNN—the thick black rectangle—sliding across all  $3 \times 3$  regions and making predictions at each step.

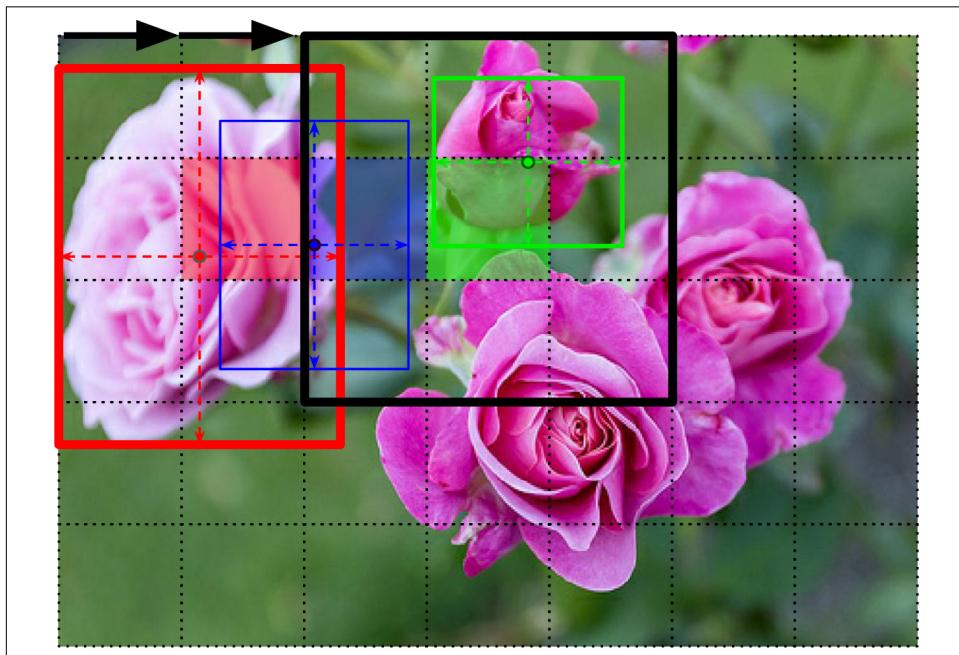


Figure 12-25. Detecting multiple objects by sliding a CNN across the image

In this figure, the CNN has already made predictions for three of these  $3 \times 3$  regions:

- When looking at the top-left  $3 \times 3$  region (centered on the red-shaded grid cell located in the second row and second column), it detected the leftmost rose. Notice that the predicted bounding box exceeds the boundary of this  $3 \times 3$  region. That's absolutely fine: even though the CNN could not see the bottom part of the rose, it was able to make a reasonable guess as to where it might be. It also predicted class probabilities, giving a high probability to the "rose" class. Lastly, it predicted a fairly high objectness score, since the center of the bounding box lies within the central grid cell (in this figure, the objectness score is represented by the thickness of the bounding box).
- When looking at the next  $3 \times 3$  region, one grid cell to the right (centered on the shaded blue square), it did not detect any flower centered in that region, so it predicted a very low objectness score; therefore, the predicted bounding box and class probabilities can safely be ignored. You can see that the predicted bounding box was no good anyway.
- Finally, when looking at the next  $3 \times 3$  region, again one grid cell to the right (centered on the shaded green cell), it detected the rose at the top, although not perfectly. This rose is not well centered within this region, so the predicted objectness score was not very high.

You can imagine how sliding the CNN across the whole image would give you a total of 15 predicted bounding boxes, organized in a  $3 \times 5$  grid, with each bounding box accompanied by its estimated class probabilities and objectness score. Since objects can have varying sizes, you may then want to slide the CNN again across  $2 \times 2$  and  $4 \times 4$  regions as well, to capture smaller and larger objects.

This technique is fairly straightforward, but as you can see it will often detect the same object multiple times, at slightly different positions. Some post-processing is needed to get rid of all the unnecessary bounding boxes. A common approach for this is called *non-max suppression* (NMS). Here's how it works:

1. First, get rid of all the bounding boxes for which the objectness score is below some threshold; since the CNN believes there's no object at that location, the bounding box is useless.
2. Find the remaining bounding box with the highest objectness score, and get rid of all the other remaining bounding boxes that overlap a lot with it (e.g., with an IoU greater than 60%). For example, in [Figure 12-25](#), the bounding box with the max objectness score is the thick bounding box over the leftmost rose. The other bounding box that touches this same rose overlaps a lot with the max bounding box, so we will get rid of it (although in this example it would already have been removed in the previous step).

3. Repeat step 2 until there are no more bounding boxes to get rid of.

This simple approach to object detection works pretty well, but it requires running the CNN many times (15 times in this example), so it is quite slow. Fortunately, there is a much faster way to slide a CNN across an image: using a *fully convolutional network* (FCN).

## Fully Convolutional Networks

The idea of FCNs was first introduced in a [2015 paper<sup>33</sup>](#) by Jonathan Long et al., for semantic segmentation (the task of classifying every pixel in an image according to the class of the object it belongs to). The authors pointed out that you could replace the dense layers at the top of a CNN with convolutional layers. To understand this, let's look at an example: suppose a dense layer with 200 neurons sits on top of a convolutional layer that outputs 100 feature maps, each of size  $7 \times 7$  (this is the feature map size, not the kernel size). Each neuron will compute a weighted sum of all  $100 \times 7 \times 7$  activations from the convolutional layer (plus a bias term). Now let's see what happens if we replace the dense layer with a convolutional layer using 200 filters, each of size  $7 \times 7$ , and with "valid" padding. This layer will output 200 feature maps, each  $1 \times 1$  (since the kernel is exactly the size of the input feature maps and we are using "valid" padding). In other words, it will output 200 numbers, just like the dense layer did; and if you look closely at the computations performed by a convolutional layer, you will notice that these numbers will be precisely the same as those the dense layer produced. The only difference is that the dense layer's output was a tensor of shape  $[batch\ size, 200]$ , while the convolutional layer will output a tensor of shape  $[batch\ size, 200, 1, 1]$ .



To convert a dense layer to a convolutional layer, the number of filters in the convolutional layer must be equal to the number of units in the dense layer, the filter size must be equal to the size of the input feature maps, and you must use "valid" padding. The stride may be set to 1 or more, as we will see shortly.

Why is this important? Well, while a dense layer expects a specific input size (since it has one weight per input feature), a convolutional layer will happily process images of any size<sup>34</sup> (however, it does expect its inputs to have a specific number of channels, since each kernel contains a different set of weights for each input channel). Since

---

<sup>33</sup> Jonathan Long et al., "Fully Convolutional Networks for Semantic Segmentation", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 3431–3440.

<sup>34</sup> There is one small exception: a convolutional layer using "valid" padding will complain if the input size is smaller than the kernel size.

an FCN contains only convolutional layers (and pooling layers, which have the same property), it can be trained and executed on images of any size!

For example, suppose we'd already trained a CNN for flower classification and localization, with an extra head for objectness. It was trained on  $224 \times 224$  images, and it outputs 107 values per image:

- The classification head outputs 102 class logits (one per class), trained using the `nn.CrossEntropyLoss`.
- The objectness head outputs a single objectness logit, trained using the `nn.BCE Loss`.
- The localization head outputs four numbers describing the bounding box, trained using the CIoU loss.

We can now convert the CNN's dense layers (`nn.Linear`) to convolutional layers (`nn.Conv2d`). In fact, we don't even need to retrain the model; we can just copy the weights from the dense layers to the convolutional layers! Alternatively, we could have converted the CNN into an FCN before training.

Now suppose the last convolutional layer before the output layer (also called the bottleneck layer) outputs  $7 \times 7$  feature maps when the network is fed a  $224 \times 224$  image (see the left side of [Figure 12-26](#)). For example, this would be the case if the network contains 5 layers with stride 2 and "same" padding, so the spatial dimensions get divided by  $2^5 = 32$  overall. If we feed the FCN a  $448 \times 448$  image (see the righthand side of [Figure 12-26](#)), the bottleneck layer will now output  $14 \times 14$  feature maps. Since the dense output layer was replaced by a convolutional layer using 107 filters of size  $7 \times 7$ , with "valid" padding and stride 1, the output will be composed of 107 feature maps, each of size  $8 \times 8$  (since  $14 - 7 + 1 = 8$ ).

In other words, the FCN will process the whole image only once, and it will output an  $8 \times 8$  grid where each cell contains the predictions for one region of the image: 107 numbers representing 102 class probabilities, 1 objectness score, and 4 bounding box coordinates. It's exactly like taking the original CNN and sliding it across the image using 8 steps per row and 8 steps per column. To visualize this, imagine chopping the original image into a  $14 \times 14$  grid, then sliding a  $7 \times 7$  window across this grid; there will be  $8 \times 8 = 64$  possible locations for the window, hence  $8 \times 8$  predictions. However, the FCN approach is *much* more efficient, since the network only looks at the image once. In fact, *You Only Look Once* (YOLO) is the name of a very popular object detection architecture, which we'll look at next.

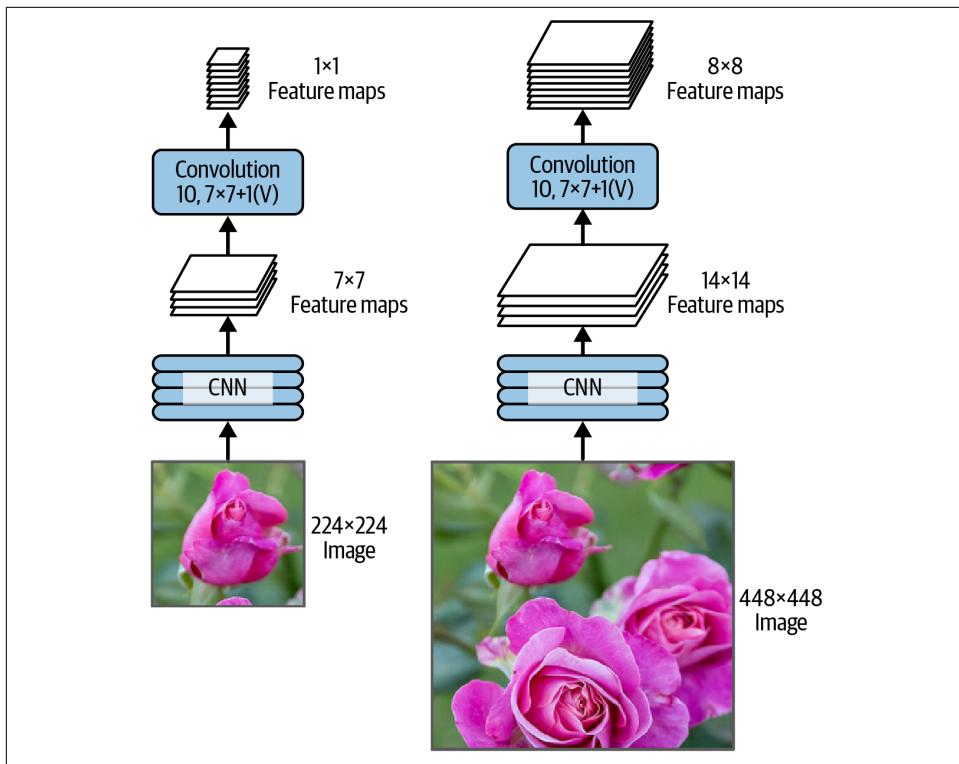


Figure 12-26. The same fully convolutional network processing a small image (left) and a large one (right)

## You Only Look Once

YOLO is a fast and accurate object detection architecture proposed by Joseph Redmon et al. in a [2015 paper](#).<sup>35</sup> It is so fast that it can run in real time on a video, as seen in Redmon's [demo](#). YOLO's architecture is quite similar to the one we just discussed, but with a few important differences:

- For each grid cell, YOLO only considers objects whose bounding box center lies within that cell. The bounding box coordinates are relative to that cell, where (0, 0) means the top-left corner of the cell and (1, 1) means the bottom right. However, the bounding box's height and width may extend well beyond the cell.

<sup>35</sup> Joseph Redmon et al., "You Only Look Once: Unified, Real-Time Object Detection", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016): 779–788.

- It outputs two bounding boxes for each grid cell (instead of just one), which allows the model to handle cases where two objects are so close to each other that their bounding box centers lie within the same cell. Each bounding box also comes with its own objectness score.
- YOLO also outputs a class probability distribution for each grid cell, predicting 20 class probabilities per grid cell since YOLO was trained on the PASCAL VOC dataset, which contains 20 classes. This produces a coarse *class probability map*. Note that the model predicts one class probability distribution per grid cell, not per bounding box. However, it's possible to estimate class probabilities for each bounding box during post-processing by measuring how well each bounding box matches each class in the class probability map. For example, imagine a picture of a person standing in front of a car. There will be two bounding boxes: one large horizontal one for the car, and a smaller vertical one for the person. These bounding boxes may have their centers within the same grid cell. So how can we tell which class should be assigned to each bounding box? Well, the class probability map will contain a large region where the “car” class is dominant, and inside it there will be a smaller region where the “person” class is dominant. Hopefully, the car’s bounding box will roughly match the “car” region, while the person’s bounding box will roughly match the “person” region: this will allow the correct class to be assigned to each bounding box.

YOLO was originally developed using Darknet, an open source deep learning framework initially developed in C by Joseph Redmon, but it was soon ported to PyTorch and other libraries. It has been continuously improved over the years, initially by Joseph Redmon et al. (YOLOv2, YOLOv3, and YOLO9000), then by various other teams since 2020. Each version brought some impressive improvements in speed and accuracy, using a variety of techniques; for example, YOLOv3 boosted accuracy in part thanks to *anchor priors*, exploiting the fact that some bounding box shapes are more likely than others, depending on the class (e.g., people tend to have vertical bounding boxes, while cars usually don’t). They also increased the number of bounding boxes per grid cell, they trained on different datasets with many more classes (up to 9,000 classes organized in a hierarchy in the case of YOLO9000), they added skip connections to recover some of the spatial resolution that is lost in the CNN (we will discuss this shortly when we look at semantic segmentation), and much more. There are many variants of these models too, such as scaled down “tiny” YOLOs, optimized to be trained on less powerful machines and which can run extremely fast (at over 1,000 frames per second!), but with a slightly lower *mean average precision* (mAP).

## Mean Average Precision

A very common metric used in object detection tasks is the mean average precision. “Mean average” sounds a bit redundant, doesn’t it? To understand this metric, let’s go back to two classification metrics we discussed in [Chapter 3](#): precision and recall. Remember the trade-off: in general, the higher the recall, the lower the precision. You can visualize this in a precision/recall curve (see [Figure 3-6](#)). To summarize this curve into a single number, we could compute its area under the curve (AUC). But note that the precision/recall curve may contain a few sections where precision actually goes up when recall increases, especially at low recall values (you can see this at the top right of [Figure 3-6](#)). This is one of the motivations for the mAP metric.

Suppose the classifier has 90% precision at 10% recall, but 96% precision at 20% recall. There’s really no trade-off here: it simply makes more sense to use the classifier at 20% recall rather than at 10% recall, as you will get both higher recall and higher precision. So instead of looking at the precision *at* 10% recall, we should really be looking at the *maximum* precision that the classifier can offer with *at least* 10% recall. It would be 96%, not 90%. Therefore, one way to get a fair idea of the model’s performance is to compute the maximum precision you can get with at least 0% recall, then 10% recall, 20%, and so on up to 100%, and then calculate the mean of these maximum precisions. This is called the *average precision* (AP) metric. Now when there are more than two classes, we can compute the AP for each class, and then compute the mean AP (mAP). Conveniently, the TorchMetrics library implements all of this in the `MeanAveragePrecision` metric.

In an object detection system, there is an additional level of complexity: what if the system detected the correct class, but at the wrong location (i.e., the bounding box is completely off)? Surely we should not count this as a positive prediction. One approach is to define an IoU threshold: for example, we may consider that a prediction is correct only if the IoU is greater than, say, 0.5, and the predicted class is correct. The corresponding mAP is generally denoted  $\text{mAP}@0.5$  (or  $\text{mAP}@50\%$ , or sometimes just  $\text{AP}_{50}$ ). In some competitions (such as the PASCAL VOC challenge), this is what is done. In others (such as the COCO competition), the mAP is computed for different IoU thresholds (0.50, 0.55, 0.60, ..., 0.95), and the final metric is the mean of all these mAPs (denoted  $\text{mAP}@[.50:.95]$  or  $\text{mAP}@[.50:0.05:.95]$ ). Yes, that’s a mean mean average.

TorchVision does not include any YOLO model, but you can use the Ultralytics library, which provides a simple API to download and use various pretrained YOLO models, based on PyTorch. These models were pretrained on the COCO dataset which contains over 330,000 images, including 200,000 images annotated for object detection with 80 different classes (person, car, truck, bicycle, ball, etc.). The Ultralytics library is not installed on Colab by default, so we must run `%pip install ultralytics`. Then we can download a YOLO model and use it. For example, here is

how to use this library to download the YOLOv9 model (medium variant) and detect objects in a batch of images (the model accepts PIL images, NumPy arrays, and even URLs):

```
from ultralytics import YOLO

model = YOLO('yolov9m.pt') # n=nano, s=small, m=medium, x=large
images = ["https://homl.info/soccer.jpg", "https://homl.info/traffic.jpg"]
results = model(images)
```

The output is a list of `Results` objects which offers a handy `summary()` method. For example, here is how we can see the first detected object in the first image:

```
>>> results[0].summary()[0]
{'name': 'sports ball',
 'class': 32,
 'confidence': 0.96214,
 'box': {'x1': 245.35733, 'y1': 286.03003, 'x2': 300.62509, 'y2': 343.57184}}
```



The Ultralytics library also provides a simple API to train a YOLO model on other common object detection datasets, or on your own dataset. See <https://docs.ultralytics.com/modes/train> for more details.

Several other pretrained object detection models are available via TorchVision. You can use them just like the pretrained classification models (e.g., ConvNeXt), except that each image prediction is represented as a dictionary containing two entries: "labels" (i.e., class IDs) and "boxes". The available models are listed here (see the [models page](#) for the full list of variants available):

#### *Faster R-CNN*<sup>36</sup>

This model has two stages: the image first goes through a CNN, then the output is passed to a *region proposal network* (RPN) that proposes bounding boxes that are most likely to contain an object; a classifier is then run for each bounding box, based on the cropped output of the CNN.

#### *SSD*<sup>37</sup>

SSD is a single-stage detector (“look once”) similar to YOLO.

#### *SSDlite*<sup>38</sup>

A lightweight version of SSD, well suited for mobile devices.

---

<sup>36</sup> Shaoqing Ren et al., “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”, *Proceedings of the 28th International Conference on Neural Information Processing Systems* 1 (2015): 91–99.

<sup>37</sup> Wei Liu et al., “SSD: Single Shot Multibox Detector”, *Proceedings of the 14th European Conference on Computer Vision* 1 (2016): 21–37.

### *RetinaNet*<sup>39</sup>

A single-stage detector which introduced a variant of the cross-entropy loss called the *focal loss* (see `torchvision.ops.sigmoid_focal_loss()`). This loss gives more weight to difficult samples and thereby improves performance on small objects and less frequent classes.

### *FCOS*<sup>40</sup>

A single-stage fully convolutional net which directly predicts bounding boxes without relying on anchor boxes.

So far, we've only considered detecting objects in single images. But what about videos? Objects must not only be detected in each frame, they must also be tracked over time. Let's take a quick look at object tracking now.

## Object Tracking

Object tracking is a challenging task: objects move, they may grow or shrink as they get closer or further away, their appearance may change as they turn around or move to different lighting conditions or backgrounds, they may be temporarily occluded by other objects, and so on.

One of the most popular object tracking systems is *DeepSORT*.<sup>41</sup> It is based on a combination of classical algorithms and deep learning:

- It uses *Kalman filters* to estimate the most likely current position of an object given prior detections, and assuming that objects tend to move at a constant speed.
- It uses a deep learning model to measure the resemblance between new detections and existing tracked objects.
- Lastly, it uses the *Hungarian algorithm* to map new detections to existing tracked objects (or to new tracked objects). This algorithm efficiently finds the combination of mappings that minimizes the distance between the detections and the predicted positions of tracked objects, while also minimizing the appearance discrepancy.

For example, imagine a red ball that just bounced off a blue ball traveling in the opposite direction. Based on the previous positions of the balls, the Kalman filter

---

<sup>38</sup> Mark Sandler et al., “MobileNetV2: Inverted Residuals and Linear Bottlenecks”, arXiv preprint arXiv:1801.04381 (2018).

<sup>39</sup> Tsung-Yi Lin et al., “Focal Loss for Dense Object Detection”, arXiv preprint arXiv:1708.02002 (2017).

<sup>40</sup> Zhi Tian et al., “FCOS: Fully Convolutional One-Stage Object Detection”, arXiv preprint arXiv:1904.01355 (2019).

<sup>41</sup> Nicolai Wojke et al., “Simple Online and Realtime Tracking with a Deep Association Metric”, arXiv preprint arXiv:1703.07402 (2017).

will predict that the balls will go through each other; indeed, it assumes that objects move at a constant speed, so it will not expect the bounce. If the Hungarian algorithm only considered positions, then it would happily map the new detections to the wrong balls, as if they had just gone through each other and swapped colors. But thanks to the resemblance measure, the Hungarian algorithm will notice the problem. Assuming the balls are not too similar, the algorithm will map the new detections to the correct balls.

The Ultralytics library supports object tracking. It uses the [Bot-SORT algorithm](#) by default: this algorithm is very similar to DeepSORT but it's faster and more accurate thanks to improvements such as camera-motion compensation and tweaks to the Kalman filter.<sup>42</sup> For example, we can track objects in a video using the YOLOv9 model we created earlier by executing the following code. In this example, we also print the ID of each tracked object at every frame, and we save a copy of the video with annotations (its path is displayed at the end):

```
my_video = "https://homl.info/cars.mp4"
results = model.track(source=my_video, stream=True, save=True)
for frame_results in results:
    summary = frame_results.summary() # similar summary as earlier + track id
    track_ids = [obj["track_id"] for obj in summary]
    print("Track ids:", track_ids)
```

So far we have located objects using bounding boxes. This is often sufficient, but sometimes you need to locate objects with much more precision—for example, to remove the background behind a person during a videoconference call. Let's see how to go down to the pixel level.

## Semantic Segmentation

In *semantic segmentation*, each pixel is classified according to the class of the object it belongs to (e.g., road, car, pedestrian, building, etc.), as shown in [Figure 12-27](#). Note that different objects of the same class are *not* distinguished. For example, all the bicycles on the righthand side of the segmented image end up as one big lump of pixels. The main difficulty in this task is that when images go through a regular CNN, they gradually lose their spatial resolution (due to the layers with strides greater than 1); so, a regular CNN may end up knowing that there's a person somewhere in the bottom left of the image, but it might not be much more precise than that.

---

<sup>42</sup> Nir Aharon et al., “BoT-SORT: Robust Associations Multi-Pedestrian Tracking”, arXiv preprint arXiv:2206.14651 (2022).



Figure 12-27. Semantic segmentation

Just like for object detection, there are many different approaches to tackle this problem, some quite complex. However, a fairly simple solution was proposed in the 2015 paper by Jonathan Long et al., that I mentioned earlier, on fully convolutional networks. The authors start by taking a pretrained CNN and turning it into an FCN. The CNN applies an overall stride of 32 to the input image (i.e., if you multiply all the strides), meaning the last layer outputs feature maps that are 32 times smaller than the input image. This is clearly too coarse, so they added a single *upsampling layer* that multiplies the resolution by 32.

There are several solutions available for upsampling (increasing the size of an image), such as bilinear interpolation, but that only works reasonably well up to  $\times 4$  or  $\times 8$ . Instead, they use a *transposed convolutional layer*:<sup>43</sup> this is equivalent to first stretching the image by inserting empty rows and columns (full of zeros), then performing a regular convolution (see Figure 12-28). Alternatively, some people prefer to think of it as a regular convolutional layer that uses fractional strides (e.g., the stride is 1/2 in Figure 12-28). The transposed convolutional layer can be initialized to perform something close to linear interpolation, but since it is a trainable layer, it will learn to do better during training. In PyTorch, you can use the `nn.ConvTranspose2d` layer.



In a transposed convolutional layer, the stride defines how much the input will be stretched, not the size of the filter steps, so the larger the stride, the larger the output (unlike for convolutional layers or pooling layers).

---

<sup>43</sup> This type of layer is sometimes referred to as a *deconvolution layer*, but it does *not* perform what mathematicians call a deconvolution, so this name should be avoided.

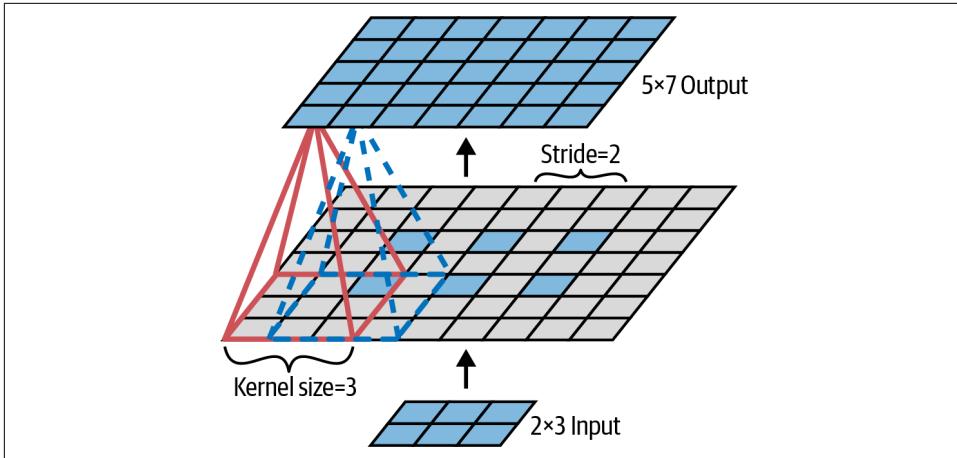


Figure 12-28. Upsampling using a transposed convolutional layer

## Other PyTorch Convolutional Layers

PyTorch also offers a few other kinds of convolutional layers:

### `nn.Conv1d`

A convolutional layer for 1D inputs, such as time series or text (sequences of letters or words), as you will see in [Chapter 13](#).

### `nn.Conv3d`

A convolutional layer for 3D inputs, such as 3D PET scans.

### *À-trous convolutional layer*

Setting the `dilation` hyperparameter of any convolutional layer to a value of 2 or more creates an *à-trous convolutional layer* (*à trous* is French for “with holes”). This is equivalent to using a regular convolutional layer with a filter dilated by inserting rows and columns of zeros (i.e., holes). For example, a  $1 \times 3$  filter equal to `[[1, 2, 3]]` may be dilated with a *dilation rate* of 4, resulting in a *dilated filter* of `[[1, 0, 0, 0, 2, 0, 0, 0, 3]]`. This lets the convolutional layer have a larger receptive field at no computational cost and using no extra parameters.

Using transposed convolutional layers for upsampling is OK, but still too imprecise. To do better, Long et al. added skip connections from lower layers: for example, they upsampled the output image by a factor of 2 (instead of 32), and they added the output of a lower layer that had this double resolution. Then they upsampled the result by a factor of 16, leading to a total upsampling factor of 32 (see [Figure 12-29](#)). This recovered some of the spatial resolution that was lost in earlier pooling layers. In their best architecture, they used a second similar skip connection to recover even

finer details from an even lower layer. In short, the output of the original CNN goes through the following extra steps: upsample  $\times 2$ , add the output of a lower layer (of the appropriate scale), upsample  $\times 2$ , add the output of an even lower layer, and finally upsample  $\times 8$ . It is even possible to scale up beyond the size of the original image: this can be used to increase the resolution of an image, which is a technique called *super-resolution*.

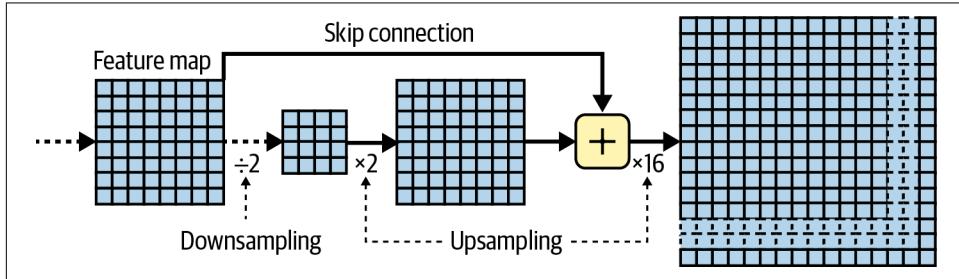


Figure 12-29. Skip layers recover some spatial resolution from lower layers



The FCN model is available in TorchVision, along with a couple other semantic segmentation models. See the notebook for a code example.

*Instance segmentation* is similar to semantic segmentation, but instead of merging all objects of the same class into one big lump, each object is distinguished from the others (e.g., it identifies each individual bicycle). For example the *Mask R-CNN* architecture, proposed in a [2017 paper<sup>44</sup>](#) by Kaiming He et al., extends the Faster R-CNN model by additionally producing a pixel mask for each bounding box. So not only do you get a bounding box around each object, with a set of estimated class probabilities, you also get a pixel mask that locates pixels in the bounding box that belong to the object. This model is available in TorchVision, pretrained on the COCO 2017 dataset.



TorchVision's transforms API v2 can apply to masks and videos, just like it applies to bounding boxes, thanks to the `Video` and `Mask` `Tensor`s.

As you can see, the field of deep computer vision is vast and fast-paced, with all sorts of architectures popping up every year. If you want to try the latest and greatest

<sup>44</sup> Kaiming He et al., “Mask R-CNN”, arXiv preprint arXiv:1703.06870 (2017).

models, check out the trending papers at <https://huggingface.co/papers>. Most of them used to be based on convolutional neural networks, but since 2020 another neural net architecture has entered the computer vision space: Transformers (which we will discuss in [Chapter 14](#)). The progress made over the last 15 years has been astounding, and researchers are now focusing on harder and harder problems, such as *adversarial learning* (which attempts to make the network more resistant to images designed to fool it), *explainability* (understanding why the network makes a specific classification), realistic *image generation* (which we will come back to in [Chapter 18](#)), *single-shot learning* (a system that can recognize an object after it has seen it just once), predicting the next frames in a video, combining text and image tasks, and more.

Now on to the next chapter, where we will look at how to process sequential data such as time series using recurrent neural networks and convolutional neural networks.

## Exercises

1. What are the advantages of a CNN over a fully connected DNN for image classification?
2. Consider a CNN composed of three convolutional layers, each with  $3 \times 3$  kernels, a stride of 2, and "same" padding. The lowest layer outputs 100 feature maps, the middle one outputs 200, and the top one outputs 400. The input images are RGB images of  $200 \times 300$  pixels:
  - a. What is the total number of parameters in the CNN?
  - b. If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance?
  - c. What about when training on a mini-batch of 50 images?
3. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?
4. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?
5. Can you name the main innovations in AlexNet, as compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, Xception, EfficientNet, and ConvNeXt?
6. What is a fully convolutional network? How can you convert a dense layer into a convolutional layer?
7. What is the main technical difficulty of semantic segmentation?
8. Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.

9. Use transfer learning for large image classification, going through these steps:
  - a. Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.). Alternatively, you can use an existing dataset, such as the one used in PyTorch's [transfer learning for computer vision tutorial](#).
  - b. Split it into a training set, a validation set, and a test set.
  - c. Build the input pipeline, apply the appropriate preprocessing operations, and optionally add data augmentation.
  - d. Fine-tune a pretrained model on this dataset.
10. Go through PyTorch's [object detection fine-tuning tutorial](#).

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.

# Processing Sequences Using RNNs and CNNs

Predicting the future is something you do all the time, whether you are finishing a friend’s sentence or anticipating the smell of coffee at breakfast. In this chapter we will discuss recurrent neural networks (RNNs)—a class of nets that can predict the future (well, up to a point). RNNs can analyze time series data, such as the number of daily active users on your website, the hourly temperature in your city, your home’s daily power consumption, the trajectories of nearby cars, and more. Once an RNN learns past patterns in the data, it is able to use its knowledge to forecast the future, assuming, of course, that past patterns still hold in the future.

More generally, RNNs can work on sequences of arbitrary lengths, rather than on fixed-sized inputs. For example, they can take sentences, documents, or audio samples as input, which makes them extremely useful for natural language processing applications such as automatic translation or speech-to-text.

In this chapter, we will first go through the fundamental concepts underlying RNNs and how to train them using backpropagation through time. Then, we will use them to forecast a time series. Along the way, we will look at the popular autoregressive moving average (ARMA) family of models, often used to forecast time series, and use them as baselines to compare with our RNNs. After that, we’ll explore the two main difficulties that RNNs face:

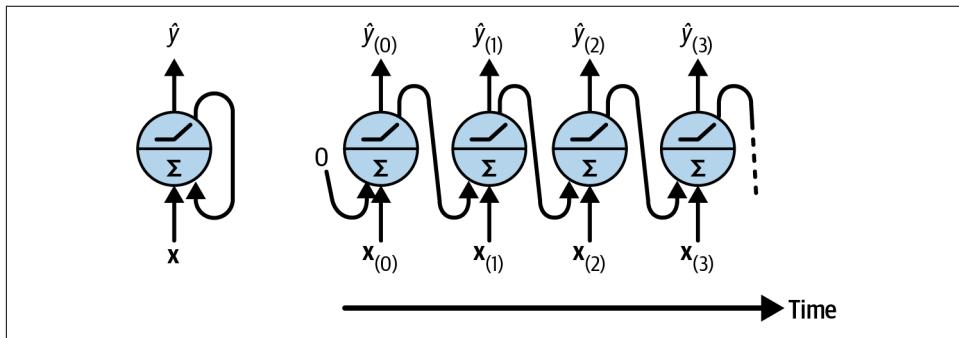
- Unstable gradients (discussed in [Chapter 11](#)), which can be alleviated using various techniques, including *recurrent dropout* and *recurrent layer normalization*.
- A (very) limited short-term memory, which can be extended using long short-term memory (LSTM) and gated recurrent unit (GRU) cells.

RNNs are not the only types of neural networks capable of handling sequential data. For small sequences, a regular dense network can do the trick, and for very long sequences, such as audio samples or text, convolutional neural networks can actually work quite well too. We will discuss both of these possibilities, and we will finish this chapter by implementing a WaveNet—a CNN architecture capable of handling sequences of tens of thousands of time steps. But we can do even better! In [Chapter 14](#), we will apply RNNs to natural language processing (NLP), and we will see how to boost them using attention mechanisms. Attention is at the core of transformers, which we will discover in [Chapter 15](#): they are now the state of the art for sequence processing, NLP, and even computer vision. But before we get there, let's start with the simplest RNNs!

## Recurrent Neurons and Layers

Up to now we have focused on feedforward neural networks, where the activations flow only in one direction, from the input layer to the output layer. A recurrent neural network looks very much like a feedforward neural network, except it also has connections pointing backward.

Let's look at the simplest possible RNN, composed of one neuron receiving inputs, producing an output, and sending that output back to itself (see [Figure 13-1](#), left). At each *time step*  $t$  (also called a *frame*), this *recurrent neuron* receives the inputs  $\mathbf{x}_{(t)}$  as well as its own output from the previous time step,  $\hat{y}_{(t-1)}$ . Since there is no previous output at the first time step, it is generally set to zero. We can represent this tiny network against the time axis (see [Figure 13-1](#), right). This is called *unrolling the network through time* (it's the same recurrent neuron represented once per time step).



*Figure 13-1. A recurrent neuron (left) unrolled through time (right)*

You can easily create a layer of recurrent neurons. At each time step  $t$ , every neuron receives both the input vector  $\mathbf{x}_{(t)}$  and the output vector from the previous time step  $\hat{\mathbf{y}}_{(t-1)}$ , as shown in [Figure 13-2](#). Note that both the inputs and outputs are now vectors (when there was just a single neuron, the output was a scalar).

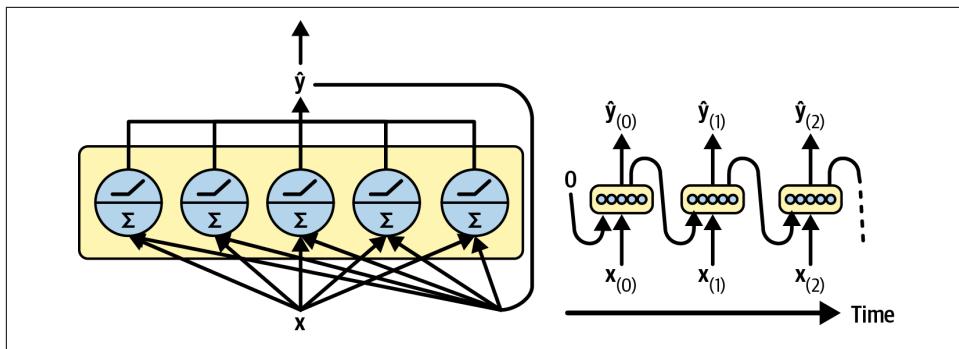


Figure 13-2. A layer of recurrent neurons (left) unrolled through time (right)

Each recurrent neuron has two sets of weights: one for the inputs  $x_{(t)}$  and the other for the outputs of the previous time step,  $\hat{y}_{(t-1)}$ . Let's call these weight vectors  $w_x$  and  $w_{\hat{y}}$ . If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices:  $W_x$  and  $W_{\hat{y}}$ .

The output vector of the whole recurrent layer can then be computed pretty much as you might expect, as shown in [Equation 13-1](#), where  $\mathbf{b}$  is the bias vector and  $\phi(\cdot)$  is the activation function (e.g., ReLU<sup>1</sup>).

*Equation 13-1. Output of a recurrent layer for a single instance*

$$\hat{y}_{(t)} = \varphi(W_x^T x_{(t)} + W_{\hat{y}}^T \hat{y}_{(t-1)} + \mathbf{b})$$

Just as with feedforward neural networks, we can compute a recurrent layer's output in one shot for an entire mini-batch by placing all the inputs at time step  $t$  into an input matrix  $X_{(t)}$  (see [Equation 13-2](#)).

*Equation 13-2. Outputs of a layer of recurrent neurons for all instances in a mini-batch*

$$\begin{aligned} \hat{Y}_{(t)} &= \varphi(X_{(t)} W_x + \hat{Y}_{(t-1)} W_{\hat{y}} + \mathbf{b}) \\ &= \varphi([X_{(t)} \quad \hat{Y}_{(t-1)}] \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} W_x \\ W_{\hat{y}} \end{bmatrix} \end{aligned}$$

<sup>1</sup> Note that many researchers prefer to use the hyperbolic tangent (tanh) activation function in RNNs rather than the ReLU activation function. For example, see Vu Pham et al.'s [2013 paper](#) "Dropout Improves Recurrent Neural Networks for Handwriting Recognition". ReLU-based RNNs are also possible, as shown in Quoc V. Le et al.'s [2015 paper](#) "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units".

In this equation:

- $\hat{Y}_{(t)}$  is an  $m \times n_{\text{neurons}}$  matrix containing the layer's outputs at time step  $t$  for each instance in the mini-batch ( $m$  is the number of instances in the mini-batch, and  $n_{\text{neurons}}$  is the number of neurons).
- $X_{(t)}$  is an  $m \times n_{\text{inputs}}$  matrix containing the inputs for all instances ( $n_{\text{inputs}}$  is the number of input features).
- $W_x$  is an  $n_{\text{inputs}} \times n_{\text{neurons}}$  matrix containing the connection weights for the inputs of the current time step.
- $W_y$  is an  $n_{\text{neurons}} \times n_{\text{neurons}}$  matrix containing the connection weights for the outputs of the previous time step.
- $b$  is a vector of size  $n_{\text{neurons}}$  containing each neuron's bias term.
- The weight matrices  $W_x$  and  $W_y$  are often concatenated vertically into a single weight matrix  $W$  of shape  $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$  (see the second line of [Equation 13-2](#)).
- The notation  $[X_{(t)} \hat{Y}_{(t-1)}]$  represents the horizontal concatenation of the matrices  $X_{(t)}$  and  $\hat{Y}_{(t-1)}$ .

Notice that  $\hat{Y}_{(t)}$  is a function of  $X_{(t)}$  and  $\hat{Y}_{(t-1)}$ , which is a function of  $X_{(t-1)}$  and  $\hat{Y}_{(t-2)}$ , which is a function of  $X_{(t-2)}$  and  $\hat{Y}_{(t-3)}$ , and so on. This makes  $\hat{Y}_{(t)}$  a function of all the inputs since time  $t = 0$  (that is,  $X_{(0)}, X_{(1)}, \dots, X_{(t)}$ ). At the first time step,  $t = 0$ , there are no previous outputs, so they are typically assumed to be all zeros.



The idea of introducing backward connections (i.e., loops) in artificial neural networks dates back to the very origins of ANNs, but the first modern RNN architecture was [proposed by Michael I. Jordan in 1986](#).<sup>2</sup> At each time step, his RNN would look at the inputs for that time step, plus its own outputs from the previous time step. This is called *output feedback*.

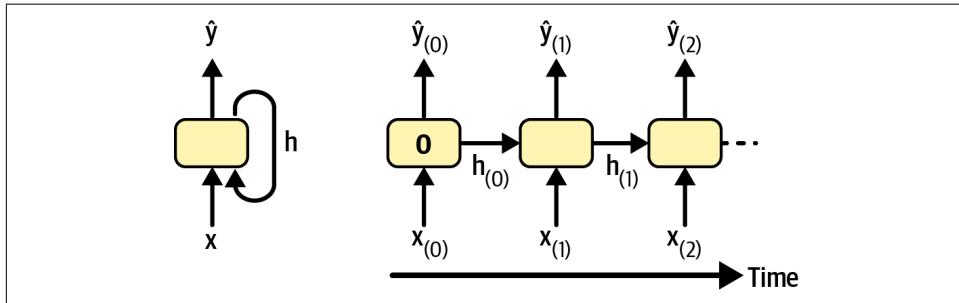
## Memory Cells

Since the output of a recurrent neuron at time step  $t$  is a function of all the inputs from previous time steps, you could say it has a form of *memory*. A part of a neural network that preserves some state across time steps is called a *memory cell* (or simply a *cell*). A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long, but this varies depending on the task). Later in this chapter, we will look at some more complex and

<sup>2</sup> Michael I. Jordan, "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine", *Proceedings of the Eighth Annual Conference of the Cognitive Science Society* (1986).

powerful types of cells capable of learning longer patterns (roughly 10 times longer, but again, this depends on the task).

A cell's state at time step  $t$ , denoted  $\mathbf{h}_{(t)}$  (the “ $h$ ” stands for “hidden”), is a function of some inputs at that time step and its state at the previous time step:  $\mathbf{h}_{(t)} = f(\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$ . Its output at time step  $t$ , denoted  $\hat{\mathbf{y}}_{(t)}$ , is also a function of the previous state and the current inputs, and typically it's just a function of the current state. In the case of the basic cells we have discussed so far, the output is just equal to the state, but in more complex cells this is not always the case, as shown in [Figure 13-3](#).



*Figure 13-3. A cell's hidden state and its output may be different*



The first modern RNN that fed back the hidden state rather than the outputs was [proposed by Jeffrey Elman in 1990](#).<sup>3</sup> This is called *state feedback*, and it's the most common approach today.

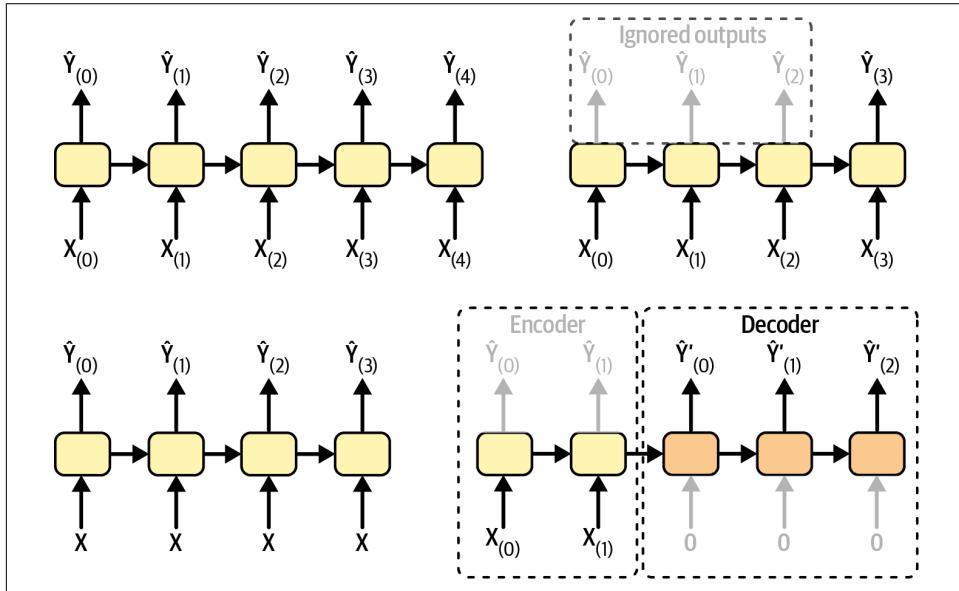
## Input and Output Sequences

An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see the top-left network in [Figure 13-4](#)). This type of *sequence-to-sequence network* is useful to forecast time series, such as your home's daily power consumption: you feed it the data over the last  $N$  days, and you train it to output the power consumption shifted by one day into the future (i.e., from  $N - 1$  days ago to tomorrow).

Alternatively, you could feed the network a sequence of inputs and ignore all outputs except for the last one (see the top-right network in [Figure 13-4](#)). This is a *sequence-to-vector network*. For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from 0 [hate] to 1 [love]).

<sup>3</sup> Jeffrey L. Elman, “Finding Structure in Time”, *Cognitive Science*, Volume 14, Issue 2 (1990).

Conversely, you could feed the network the same input vector over and over again at each time step and let it output a sequence (see the bottom-left network of [Figure 13-4](#)). This is a *vector-to-sequence network*. For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image.



*Figure 13-4. Sequence-to-sequence (top left), sequence-to-vector (top right), vector-to-sequence (bottom left), and encoder-decoder (bottom right) networks*

Lastly, you could have a sequence-to-vector network, called an *encoder*, followed by a vector-to-sequence network, called a *decoder* (see the bottom-right network of [Figure 13-4](#)). For example, this could be used for translating a sentence from one language to another. You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language. This two-step model, called an *encoder-decoder*,<sup>4</sup> works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented at the top left): the last words of a sentence can affect the first words of the translation, so you need to wait until you have seen the whole sentence before translating it. We will go through the implementation of an encoder-decoder in [Chapter 14](#) (as you will see, it is a bit more complex than what [Figure 13-4](#) suggests).

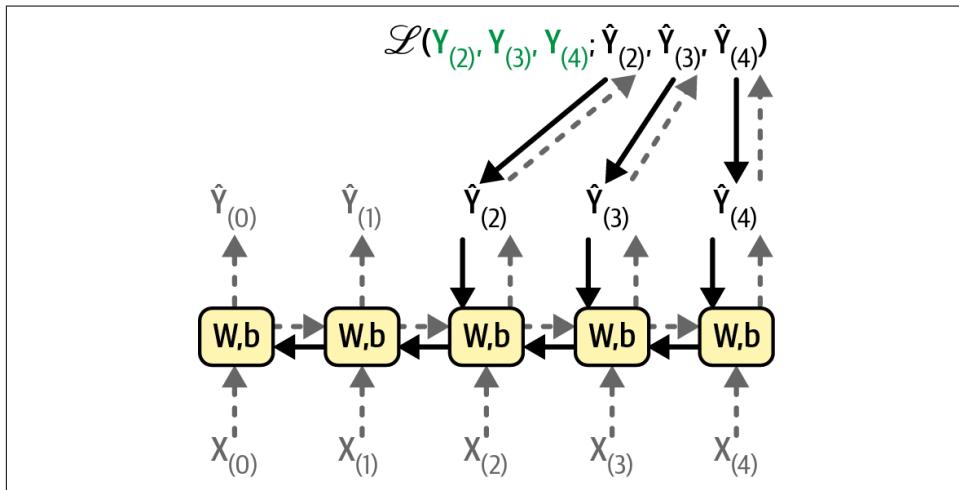
This versatility sounds promising, but how do you train a recurrent neural network?

<sup>4</sup> Nal Kalchbrenner and Phil Blunsom, “Recurrent Continuous Translation Models”, *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (2013): 1700–1709.

## Training RNNs

To train an RNN, the trick is to unroll it through time (like we just did) and then use regular backpropagation (see [Figure 13-5](#)). This strategy is called *backpropagation through time* (BPTT).

Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows). Then the output sequence is evaluated using a loss function  $\mathcal{L}(Y_{(0)}, Y_{(1)}, \dots, Y_{(T)}; \hat{Y}_{(0)}, \hat{Y}_{(1)}, \dots, \hat{Y}_{(T)})$  (where  $Y_{(i)}$  is the  $i^{\text{th}}$  target,  $\hat{Y}_{(i)}$  is the  $i^{\text{th}}$  prediction, and  $T$  is the max time step). Note that this loss function may ignore some outputs. For example, in a sequence-to-vector RNN, all outputs are ignored except for the very last one. In [Figure 13-5](#), the loss function is computed based on the last three outputs only. The gradients of that loss function are then propagated backward through the unrolled network (represented by the solid arrows). In this example, since the outputs  $\hat{Y}_{(0)}$  and  $\hat{Y}_{(1)}$  are not used to compute the loss, the gradients do not flow backward through them; they only flow through  $\hat{Y}_{(2)}$ ,  $\hat{Y}_{(3)}$ , and  $\hat{Y}_{(4)}$ . Moreover, since the same parameters  $W$  and  $b$  are used at each time step, their gradients will be tweaked multiple times during backprop. Once the backward phase is complete and all the gradients have been computed, BPTT can perform a gradient descent step to update the parameters (this is no different from regular backprop).



*Figure 13-5. Backpropagation through time*

Fortunately, PyTorch takes care of all of this complexity for you, as you will see. But before we get there, let's load a time series and start analyzing it using classical tools to better understand what we're dealing with, and to get some baseline metrics.

# Forecasting a Time Series

All right! Let's pretend you've just been hired as a data scientist by Chicago's Transit Authority. Your first task is to build a model capable of forecasting the number of passengers that will ride on bus and rail the next day. You have access to daily ridership data since 2001. Let's walk through how you would handle this. We'll start by loading and cleaning up the data:<sup>5</sup>

```
import pandas as pd
from pathlib import Path

path = Path("datasets/ridership/CTA_-_Ridership_-_Daily_Boarding_Totals.csv")
df = pd.read_csv(path, parse_dates=["service_date"])
df.columns = ["date", "day_type", "bus", "rail", "total"] # shorter names
df = df.sort_values("date").set_index("date")
df = df.drop("total", axis=1) # no need for total, it's just bus + rail
df = df.drop_duplicates() # remove duplicated months (2011-10 and 2014-07)
```

We load the CSV file, set short column names, sort the rows by date, remove the redundant total column, and drop duplicate rows. Now let's check what the first few rows look like:

```
>>> df.head()
      day_type    bus    rail
date
2001-01-01      U  297192  126455
2001-01-02      W  780827  501952
2001-01-03      W  824923  536432
2001-01-04      W  870021  550011
2001-01-05      W  890426  557917
```

On January 1st, 2001, 297,192 people boarded a bus in Chicago, and 126,455 boarded a train. The day\_type column contains W for Weekdays, A for Saturdays, and U for Sundays or holidays.

Now let's plot the bus and rail ridership figures over a few months in 2019, to see what it looks like (see Figure 13-6):

```
import matplotlib.pyplot as plt

df["2019-03":"2019-05"].plot(grid=True, marker=". ", figsize=(8, 3.5))
plt.show()
```

---

<sup>5</sup> The latest data from the Chicago Transit Authority is available at the [Chicago Data Portal](#).

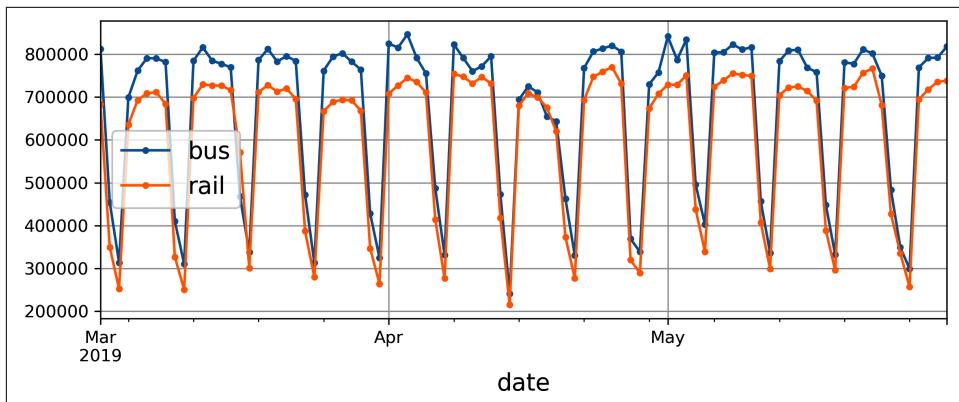


Figure 13-6. Daily ridership in Chicago

Note that Pandas includes both the start and end month in the range, so this plots the data from the 1st of March all the way up to the 31st of May. This is a *time series*: data with values at different time steps, usually at regular intervals. More specifically, since there are multiple values per time step, this is called a *multivariate time series*. If we only looked at the bus column, it would be a *univariate time series*, with a single value per time step. Predicting future values (i.e., forecasting) is the most typical task when dealing with time series, and this is what we will focus on in this chapter. Other tasks include imputation (filling in missing past values), classification, anomaly detection, and more.

Looking at Figure 13-6, we can see that a similar pattern is clearly repeated every week. This is called a weekly *seasonality*. In fact, it's so strong in this case that forecasting tomorrow's ridership by just copying the values from a week earlier will yield reasonably good results. This is called *naive forecasting*: simply copying a past value to make our forecast. Naive forecasting is often a great baseline, and it can even be tricky to beat in some cases.



In general, naive forecasting means copying the latest known value (e.g., forecasting that tomorrow will be the same as today). However, in our case, copying the value from the previous week works better, due to the strong weekly seasonality.

To visualize these naive forecasts, let's overlay the two time series (for bus and rail) as well as the same time series lagged by one week (i.e., shifted toward the right) using dotted lines. We'll also plot the difference between the two (i.e., the value at time  $t$  minus the value at time  $t - 7$ ); this is called *differencing* (see Figure 13-7):

```

diff_7 = df[["bus", "rail"]].diff(7)[ "2019-03": "2019-05"]

fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8, 5))
df.plot(ax=axs[0], legend=False, marker=".") # original time series
df.shift(7).plot(ax=axs[0], grid=True, legend=False, linestyle="--") # lagged
diff_7.plot(ax=axs[1], grid=True, marker=".") # 7-day difference time series
plt.show()

```

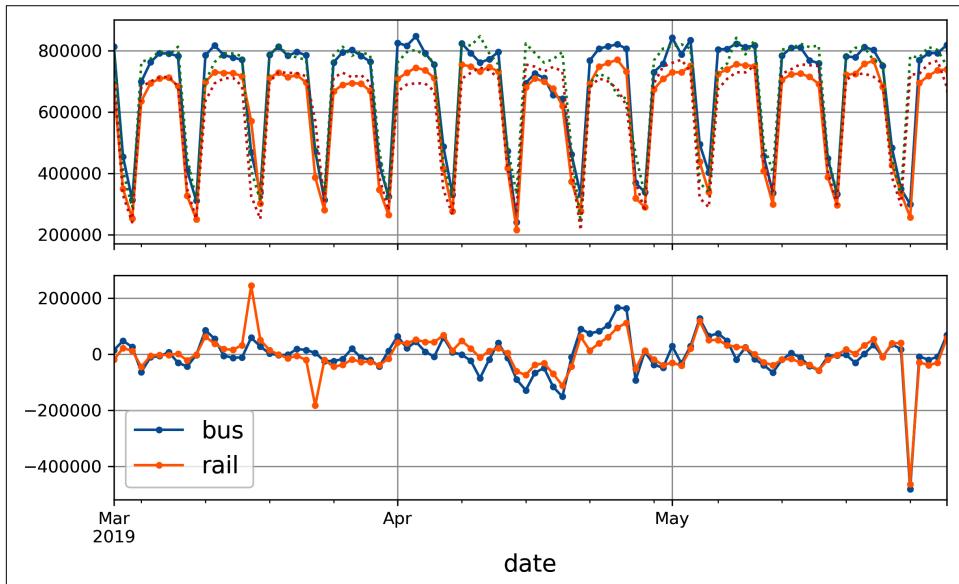


Figure 13-7. Time series overlaid with 7-day lagged time series (top), and difference between  $t$  and  $t - 7$  (bottom)

Not too bad! Notice how closely the lagged time series track the actual time series. When a time series is correlated with a lagged version of itself, we say that the time series is *autocorrelated*. As you can see, most of the differences are fairly small, except at the end of May. Maybe there was a holiday at that time? Let's check the `day_type` column:

```

>>> list(df.loc["2019-05-25": "2019-05-27"]["day_type"])
['A', 'U', 'U']

```

Indeed, there was a long weekend back then: the Monday was the Memorial Day holiday. We could use this column to improve our forecasts, but for now let's just measure the mean absolute error over the three-month period we're arbitrarily focusing on—March, April, and May 2019—to get a rough idea:

```

>>> diff_7.abs().mean()
bus      43915.608696
rail    42143.271739
dtype: float64

```

Our naive forecasts get a mean absolute error (MAE) of about 43,916 bus riders, and about 42,143 rail riders. It's hard to tell at a glance how good or bad this is, so let's put the forecast errors into perspective by dividing them by the target values:

```
>>> targets = df[['bus", "rail"]][["2019-03":"2019-05"]  
>>> (diff_7 / targets).abs().mean()  
bus      0.082938  
rail     0.089948  
dtype: float64
```

What we just computed is called the *mean absolute percentage error* (MAPE). It looks like our naive forecasts give us a MAPE of roughly 8.3% for bus and 9.0% for rail. It's interesting to note that the MAE for the rail forecasts looks slightly better than the MAE for the bus forecasts, while the opposite is true for the MAPE. That's because the bus ridership is larger than the rail ridership, so naturally the forecast errors are also larger, but when we put the errors into perspective, it turns out that the bus forecasts are actually slightly better than the rail forecasts.



The MAE, MAPE, and mean squared error (MSE) are among the most common metrics you can use to evaluate your forecasts. As always, choosing the right metric depends on the task. For example, if your project suffers quadratically more from large errors than from small ones, then the MSE may be preferable, as it strongly penalizes large errors.

Looking at the time series, there doesn't appear to be any significant monthly seasonality, but let's check whether there's any yearly seasonality. We'll look at the data from 2001 to 2019. To reduce the risk of data snooping, we'll ignore more recent data for now. Let's also plot a 12-month rolling average for each series to visualize long-term trends (see [Figure 13-8](#)):

```
period = slice("2001", "2019")  
df_monthly = df.select_dtypes(include="number").resample('ME').mean()  
rolling_average_12_months = df_monthly.loc[period].rolling(window=12).mean()  
  
fig, ax = plt.subplots(figsize=(8, 4))  
df_monthly[period].plot(ax=ax, marker=".")  
rolling_average_12_months.plot(ax=ax, grid=True, legend=False)  
plt.show()
```

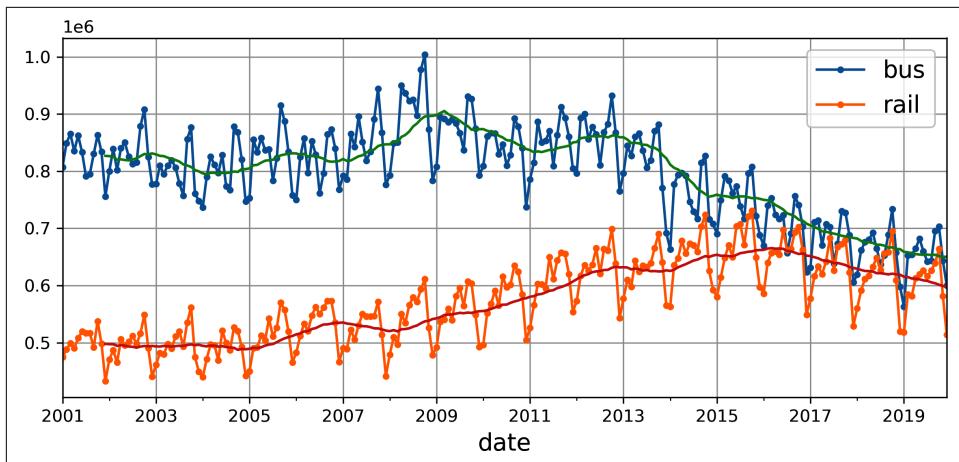


Figure 13-8. Yearly seasonality and long-term trends

Yep! There's definitely some yearly seasonality as well, although it is noisier than the weekly seasonality, and more visible for the rail series than the bus series: we see peaks and troughs at roughly the same dates each year. Let's check what we get if we plot the 12-month difference (see Figure 13-9):

```
df_monthly.diff(12)[period].plot(grid=True, marker=". ", figsize=(8, 3))
plt.show()
```

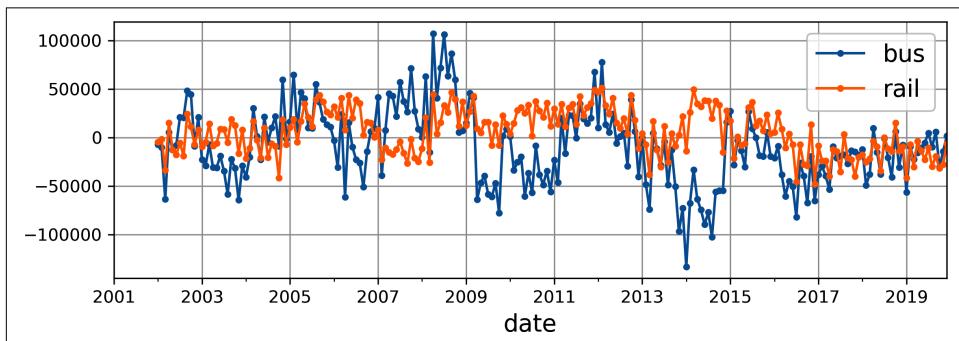


Figure 13-9. The 12-month difference

Notice how differencing not only removed the yearly seasonality, but it also removed the long-term trends. For example, the linear downward trend present in the time series from 2016 to 2019 became a roughly constant negative value in the differenced time series. In fact, differencing is a common technique used to remove trend and seasonality from a time series: it's easier to study a *stationary* time series, meaning one whose statistical properties remain the same over time, without any seasonality or trends. Once you're able to make accurate forecasts on the differenced time series,

it's easy to turn them into forecasts for the actual time series by just adding back the past values that were previously subtracted.

You may be thinking that we're only trying to predict tomorrow's ridership, so the long-term patterns matter much less than the short-term ones. You're right, but still, we may be able to improve performance slightly by taking long-term patterns into account. For example, daily bus ridership dropped by about 2,500 in October 2017, which represents about 570 fewer passengers each week, so if we were at the end of October 2017, it would make sense to forecast tomorrow's ridership by copying the value from last week, minus 570. Accounting for the trend will make your forecasts a bit more accurate on average.

Now that you're familiar with the ridership time series, as well as some of the most important concepts in time series analysis, including seasonality, trend, differencing, and moving averages, let's take a quick look at a very popular family of statistical models that are commonly used to analyze time series.

## The ARMA Model Family

We'll start with the *autoregressive moving average* (ARMA) model, developed by Herman Wold in the 1930s: it computes its forecasts using a simple weighted sum of lagged values and corrects these forecasts by adding a moving average, very much like we just discussed. Specifically, the moving average component is computed using a weighted sum of the last few forecast errors. [Equation 13-3](#) shows how the model makes its forecasts.

*Equation 13-3. Forecasting using an ARMA model*

$$\hat{y}(t) = \sum_{i=1}^p \alpha_i y(t-i) + \sum_{i=1}^q \theta_i \varepsilon(t-i)$$

with  $\varepsilon(t) = y(t) - \hat{y}(t)$

In this equation:

- $\hat{y}_{(t)}$  is the model's forecast for time step  $t$ .
- $y_{(t)}$  is the time series' value at time step  $t$ .
- The first sum is the weighted sum of the past  $p$  values of the time series, using the learned weights  $\alpha_i$ . The number  $p$  is a hyperparameter, and it determines how far back into the past the model should look. This sum is the *autoregressive* component of the model: it performs regression based on past values.

- The second sum is the weighted sum over the past  $q$  forecast errors  $\varepsilon_{(t)}$ , using the learned weights  $\theta_i$ . The number  $q$  is a hyperparameter. This sum is the moving average component of the model.

Importantly, this model assumes that the time series is stationary. If it is not, then differencing may help. Using differencing over a single time step will produce an approximation of the derivative of the time series: indeed, it will give the slope of the series at each time step. This means that it will eliminate any linear trend, transforming it into a constant value. For example, if you apply one-step differencing to the series [3, 5, 7, 9, 11], you get the differenced series [2, 2, 2, 2].

If the original time series has a quadratic trend instead of a linear trend, then a single round of differencing will not be enough. For example, the series [1, 4, 9, 16, 25, 36] becomes [3, 5, 7, 9, 11] after one round of differencing, but if you run differencing for a second round, then you get [2, 2, 2, 2]. So, running two rounds of differencing will eliminate quadratic trends. More generally, running  $d$  consecutive rounds of differencing computes an approximation of the  $d^{\text{th}}$  order derivative of the time series, so it will eliminate polynomial trends up to degree  $d$ . This hyperparameter  $d$  is called the *order of integration*.

Differencing is the central contribution of the *autoregressive integrated moving average* (ARIMA) model, introduced in 1970 by George Box and Gwilym Jenkins in their book *Time Series Analysis* (Wiley). This model runs  $d$  rounds of differencing to make the time series more stationary, then it applies a regular ARMA model. When making forecasts, it uses this ARMA model, then it adds back the terms that were subtracted by differencing.

One last member of the ARMA family is the *seasonal ARIMA* (SARIMA) model: it models the time series in the same way as ARIMA, but it additionally models a seasonal component for a given frequency (e.g., weekly), using the exact same ARIMA approach. It has a total of seven hyperparameters: the same  $p$ ,  $d$ , and  $q$  hyperparameters as ARIMA; plus additional  $P$ ,  $D$ , and  $Q$  hyperparameters to model the seasonal pattern; and lastly the period of the seasonal pattern, denoted  $s$ . The hyperparameters  $P$ ,  $D$ , and  $Q$  are just like  $p$ ,  $d$ , and  $q$ , but they are used to model the time series at  $t - s$ ,  $t - 2s$ ,  $t - 3s$ , etc.

Let's see how to fit a SARIMA model to the rail time series, and use it to make a forecast for tomorrow's ridership. We'll pretend today is the last day of May 2019, and we want to forecast the rail ridership for "tomorrow", the 1st of June, 2019. For this, we can use the `statsmodels` library, which contains many different statistical models, including the ARMA model and its variants, implemented by the ARIMA class:

```
from statsmodels.tsa.arima.model import ARIMA
origin, today = "2019-01-01", "2019-05-31"
rail_series = df.loc[origin:today]["rail"].asfreq("D")
```

```

model = ARIMA(rail_series,
              order=(1, 0, 0),
              seasonal_order=(0, 1, 1, 7))
model = model.fit()
y_pred = model.forecast() # returns 427,758.6

```

In this code example:

- We start by importing the ARIMA class, then we take the rail ridership data from the start of 2019 up to “today”, and we use `asfreq("D")` to set the time series’ frequency to daily. This doesn’t change the data at all in this case, since it’s already daily, but without this the ARIMA class would have to guess the frequency, and it would display a warning.
- Next, we create an ARIMA instance, passing it all the data until “today”, and we set the model hyperparameters: `order=(1, 0, 0)` means that  $p = 1$ ,  $d = 0$ , and  $q = 0$ ; and `seasonal_order=(0, 1, 1, 7)` means that  $P = 0$ ,  $D = 1$ ,  $Q = 1$ , and  $s = 7$ . Notice that the `statsmodels` API differs a bit from Scikit-Learn’s API, since we pass the data to the model at construction time, instead of passing it to the `fit()` method.
- Next, we fit the model, and we use it to make a forecast for “tomorrow”, the 1st of June, 2019.

The forecast is 427,759 passengers, when in fact there were 379,044. Yikes, we’re 12.9% off—that’s pretty bad. It’s actually slightly worse than naive forecasting, which forecasts 426,932, off by 12.6%. But perhaps we were just unlucky that day? To check this, we can run the same code in a loop to make forecasts for every day in March, April, and May, and compute the MAE over that period:

```

origin, start_date, end_date = "2019-01-01", "2019-03-01", "2019-05-31"
time_period = pd.date_range(start_date, end_date)
rail_series = df.loc[origin:end_date]["rail"].asfreq("D")
y_preds = []
for today in time_period.shift(-1):
    model = ARIMA(rail_series[origin:today], # train on data up to "today"
                  order=(1, 0, 0),
                  seasonal_order=(0, 1, 1, 7))
    model = model.fit() # note that we retrain the model every day!
    y_pred = model.forecast().iloc[0]
    y_preds.append(y_pred)

y_preds = pd.Series(y_preds, index=time_period)
mae = (y_preds - rail_series[time_period]).abs().mean() # returns 32,040.7

```

Ah, that’s much better! The MAE is about 32,041, which is significantly lower than the MAE we got with naive forecasting (42,143). So although the model is not perfect, it still beats naive forecasting by a large margin, on average.

At this point, you may be wondering how to pick good hyperparameters for the SARIMA model. There are several methods, but the simplest to understand and to get started with is the brute-force approach: just run a grid search. For each model you want to evaluate (i.e., each hyperparameter combination), you can run the preceding code example, changing only the hyperparameter values. Good  $p$ ,  $q$ ,  $P$ , and  $Q$  values are usually fairly small (typically 0 to 2, sometimes up to 5 or 6), and  $d$  and  $D$  are typically 0 or 1, sometimes 2. As for  $s$ , it's just the main seasonal pattern's period: in our case it's 7 since there's a strong weekly seasonality. The model with the lowest MAE wins. Of course, you can replace the MAE with another metric if it better matches your business objective. And that's it!



There are other more principled approaches to selecting good hyperparameters, based on analyzing the *autocorrelation function* (ACF) and *partial autocorrelation function* (PACF),<sup>6</sup> or minimizing the AIC or BIC metrics (introduced in [Chapter 8](#)) to penalize models that use too many parameters and reduce the risk of overfitting the data, but grid search is a good place to start.

## Preparing the Data for Machine Learning Models

Now that we have two baselines, naive forecasting and SARIMA, let's try to use the machine learning models we've covered so far to forecast this time series, starting with a basic linear model. Our goal will be to forecast tomorrow's ridership based on the ridership of the past 8 weeks of data (56 days). The inputs to our model will therefore be sequences (usually a single sequence per day once the model is in production), each containing 56 values from time steps  $t - 55$  to  $t$ . For each input sequence, the model will output a single value: the forecast for time step  $t + 1$ .

But what will we use as training data? Well, here's the trick: we will use every 56-day window from the past as training data, and the target for each window will be the value immediately following it. To do that, we need to create a custom dataset that will chop a given time series into all possible windows of a given length, each with its corresponding target:

```
class TimeSeriesDataset(torch.utils.data.Dataset):
    def __init__(self, series, window_length):
        self.series = series
        self.window_length = window_length

    def __len__(self):
        return len(self.series) - self.window_length

    def __getitem__(self, idx):
```

---

<sup>6</sup> For more details on the ACF-PACF approach, check out this very nice [post by Jason Brownlee](#).

```

if idx >= len(self):
    raise IndexError("dataset index out of range")
end = idx + self.window_length # 1st index after window
window = self.series[idx : end]
target = self.series[end]
return window, target

```

Let's test this class by applying it to a simple time series containing the numbers 0 to 5. We could represent this series in 1D using [0, 1, 2, 3, 4, 5], but the RNN modules expect each sequence to be 2D, with a shape of [*sequence length, dimensionality*]. For univariate time series, the dimensionality is simply 1, so we represent the time series as [[0], [1], [2], [3], [4], [5]]. In the following code below, the `TimeSeriesDataset` contains all the windows of length 3, each with its corresponding target (i.e., the first value after the window):

```

>>> my_series = torch.tensor([[0], [1], [2], [3], [4], [5]])
>>> my_dataset = TimeSeriesDataset(my_series, window_length=3)
>>> for window, target in my_dataset:
...     print("Window:", window, " Target:", target)
...
Window: tensor([[0], [1], [2]])  Target: tensor([3])
Window: tensor([[1], [2], [3]])  Target: tensor([4])
Window: tensor([[2], [3], [4]])  Target: tensor([5])

```

It looks like our `TimeSeriesDataset` class works fine! Now we can create a `DataLoader` for this tiny dataset, shuffling the windows and grouping them into batches of two:

```

>>> from torch.utils.data import DataLoader
>>> torch.manual_seed(0)
>>> my_loader = DataLoader(my_dataset, batch_size=2, shuffle=True)
>>> for X, y in my_loader:
...     print("X:", X, " y:", y)
...
X: tensor([[0], [1], [2]], [[2], [3], [4]])  y: tensor([3], [5])
X: tensor([[1], [2], [3]])  y: tensor([4])

```

The first batch contains the windows [[0], [1], [2]] and [[2], [3], [4]], along with their respective targets [3] and [5]; and the second batch contains only one window [[1], [2], [3]], with its target [4]. Indeed, when the length of a dataset is not a multiple of the batch size, the last batch is shorter.

OK, now that we have a way to convert a time series into a dataset that we can use to train ML models, let's go ahead and prepare our ridership dataset. First, we need to split our data into a training period, a validation period, and a test period. We will focus on the rail ridership for now. We will convert the data to 32-bit float tensors, and scale them down by a factor of one million to ensure the values end up near the 0–1 range; this plays nicely with the default weight initialization and learning rate. In

this code, we use `df[["rail"]]` instead of `df["rail"]` to ensure the resulting tensor has a shape of  $[series\ length, 1]$  rather than  $[series\ length]$ :

```
rail_train = torch.FloatTensor(df[["rail"]][("2016-01":"2018-12").values / 1e6]
rail_valid = torch.FloatTensor(df[["rail"]][("2019-01":"2019-05").values / 1e6)
rail_test = torch.FloatTensor(df[["rail"]][("2019-06":).values / 1e6)
```



When dealing with time series, you generally want to split across time. However, in some cases you may be able to split along other dimensions, which will give you a longer time period to train on. For example, if you have data about the financial health of 10,000 companies from 2001 to 2019, you might be able to split this data across the different companies. It's very likely that many of these companies will be strongly correlated, though (e.g., whole economic sectors may go up or down jointly), and if you have correlated companies across the training set and the test set, your test set will not be as useful, as its measure of the generalization error will be optimistically biased.

Next, let's use our `TimeSeriesDataset` class to create datasets for training, validation, and testing, and also create the corresponding data loaders. Since gradient descent expects the instances in the training set to be independent and identically distributed (IID), as we saw in [Chapter 4](#), we must set `shuffle` to `True`—this will shuffle the windows, but not their contents:

```
window_length = 56
train_set = TimeSeriesDataset(rail_train, window_length)
train_loader = DataLoader(train_set, batch_size=32, shuffle=True)
valid_set = TimeSeriesDataset(rail_valid, window_length)
valid_loader = DataLoader(valid_set, batch_size=32)
test_set = TimeSeriesDataset(rail_test, window_length)
test_loader = DataLoader(test_set, batch_size=32)
```

And now we're ready to build and train any regression model we want!

## Forecasting Using a Linear Model

Let's try a basic linear model first. We will use the Huber loss, which usually works better than minimizing the MAE directly, as discussed in [Chapter 9](#):

```
import torch.nn as nn
import torchmetrics

torch.manual_seed(42)
model = nn.Sequential(nn.Flatten(), nn.Linear(window_length, 1)).to(device)
loss_fn = nn.HuberLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.003, momentum=0.9)
metric = torchmetrics.MeanAbsoluteError().to(device)
[...] # train the PyTorch model (e.g., using train() function from Chapter 10)
```

Note that we must use an `nn.Flatten` layer before the `nn.Linear` layer, because the inputs have a shape of  $[batch\ size, window\ length, dimensionality]$ , but the `nn.Linear` layer expects inputs of shape  $[batch\ size, features]$ . If you train this model, you will see that it reaches a validation MAE of 37,726 (your mileage may vary). That's better than naive forecasting, but worse than the SARIMA model.<sup>7</sup>

Can we do better with an RNN? Well, let's see!

## Forecasting Using a Simple RNN

Let's implement a simple RNN containing a single recurrent layer (see Figure 13-2) plus a final `nn.Linear` layer that will take the last hidden state as input and output the model's forecast:

```
class SimpleRnnModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.memory_cell = nn.Sequential(
            nn.Linear(input_size + hidden_size, hidden_size),
            nn.Tanh()
        )
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, X):
        batch_size, window_length, dimensionality = X.shape
        X_time_first = X.transpose(0, 1)
        H = torch.zeros(batch_size, self.hidden_size, device=X.device)
        for X_t in X_time_first:
            XH = torch.cat((X_t, H), dim=1)
            H = self.memory_cell(XH)
        return self.output(H)

torch.manual_seed(42)
univar_model = SimpleRnnModel(input_size=1, hidden_size=32, output_size=1)
```

Let's go through this code:

- The constructor takes three arguments: the input size, the hidden size, and the output size. In our case, the input size is set to 1 when we create the model (on the very last line) since we are dealing with a univariate time series. The hidden size is the number of recurrent neurons. We set it to 32 in this example, but this is a hyperparameter you can tune. The output size is 1 since we're only forecasting a single value.

---

<sup>7</sup> Note that the validation period starts on the 1st of January 2019, so the first prediction is for the 26th of February 2019, eight weeks later. When we evaluated the baseline models, we used predictions starting on the 1st of March instead, but this should be close enough.

- The constructor first creates the memory cell, which will be used once per time step: it's a sequential module composed of an `nn.Linear` layer and the tanh activation function. You can use another activation function here, but it's common to use the tanh activation function because it tends to be more stable than other activation functions in RNNs.
- Next, we create an `nn.Linear` layer that will be used to take the last hidden state and produce the final output. This is needed because the hidden state has one dimension per recurrent neuron (32 in this example), while the target has just one dimension since we're dealing with a univariate time series and we're only trying to forecast a single future value. Moreover, the tanh activation function only outputs values between  $-1$  and  $+1$ , while the values we need to forecast occasionally exceed  $+1$ .
- The `forward()` method will be passed input batches produced by our data loader, so each batch will have a shape of  $[batch\ size, window\ length, dimensionality]$ , with *dimensionality* = 1.
- The hidden state  $H$  is initialized to zeros: for each input window, there's one zero per recurrent neuron, so the hidden state's shape is  $[batch\ size, hidden\_size]$ .
- Next, we iterate over each time step. For this, we must swap the first two dimensions of  $X$  using `permute(0, 1)`. As a result, the input tensor  $X_t$  at each time step has a shape of  $[batch\ size, dimensionality]$ .
- At each time step, we want to feed both the current inputs  $X_t$  and the hidden state  $H$  to the memory cell. For this, we must first concatenate  $X_t$  and  $H$  along the first dimension, resulting in a tensor  $XH$  of shape  $[batch\ size, input\_size + hidden\ size]$ . Then we can pass  $XH$  to the memory cell to get the new hidden state.
- After the loop,  $H$  represents the final hidden state. We pass it through the output `nn.Linear` layer, and we get our final prediction of shape  $[batch\ size, output\ size]$ .

In short, this model initializes the hidden state to zeros, then it goes through each time step and applies the memory cell to both the current inputs and the last hidden state, which gives it the new hidden state. It repeats this process until the last time step, then it passes the last hidden state through a linear layer to get the actual forecasts. All of this is performed simultaneously for every sequence in the batch.

So that's our first recurrent model! It's a sequence-to-vector model. Since there's a single output neuron in this case, the output vector for each input sequence has a size of 1.

Now if you move this model to the GPU, then train and evaluate it just like the previous one, you will find that its validation MAE reaches 30,659. That's the best model we've trained so far, and it even beats the SARIMA model; we're doing pretty well!



We've only normalized the time series, without removing trend and seasonality, and yet the model still performs well. This is convenient, as it makes it possible to quickly search for promising models without worrying too much about preprocessing. However, to get the best performance, you may want to try making the time series more stationary, for example, using differencing.

PyTorch comes with an `nn.RNN` module that can greatly simplify the implementation of our `SimpleRnnModel`. The following implementation is (almost) equivalent to the previous one:

```
class SimpleRnnModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, X):
        outputs, last_state = self.rnn(X)
        return self.output(outputs[:, -1])
```

The code is much shorter than before. Let's go through it:

- In the constructor, we now create an `nn.RNN` module instead of building a memory cell. We specify the input size and the hidden size, just like we did earlier, and we also set `batch_first=True` because our input batches have the batch dimension first. If we didn't set `batch_first=True`, the `nn.RNN` module would assume that the time dimension comes first (i.e., it would expect the input batches to have a shape of `[window length, batch size, dimensionality]` instead of `[batch size, window length, dimensionality]`).
- The constructor also creates an output layer, exactly like in our previous implementation.
- In the `forward()` method, we pass the input batch directly to the `nn.RNN` module. This takes care of everything: internally, it initializes the hidden state with zeros, and it processes each time step using a simple memory cell based on a linear layer and an activation function (`tanh` by default), much like we did earlier.
- Note that the `nn.RNN` module returns two things:
  - `outputs` is a tensor containing the outputs of the top recurrent layer at every time step. Right now we have a single recurrent layer, but in the next section we will see that the `nn.RNN` module supports multiple recurrent layers. Since we are dealing with a simple RNN, the outputs are just the hidden states of the top recurrent layer at each time step. The `outputs` tensor has a shape of `[batch size, window length, hidden size]` (if we didn't set `batch_first=True`, then the first two dimensions would be swapped).

- `last_state` contains the hidden state of each recurrent layer after the very last time step. Its shape is `[number of layers, batch size, hidden size]`. In our case, there's a single recurrent layer, so the size of the first dimension is 1.
- In the end, we take the last output (which is also the last state of the top recurrent layer) and we pass it through our `nn.Linear` output layer.

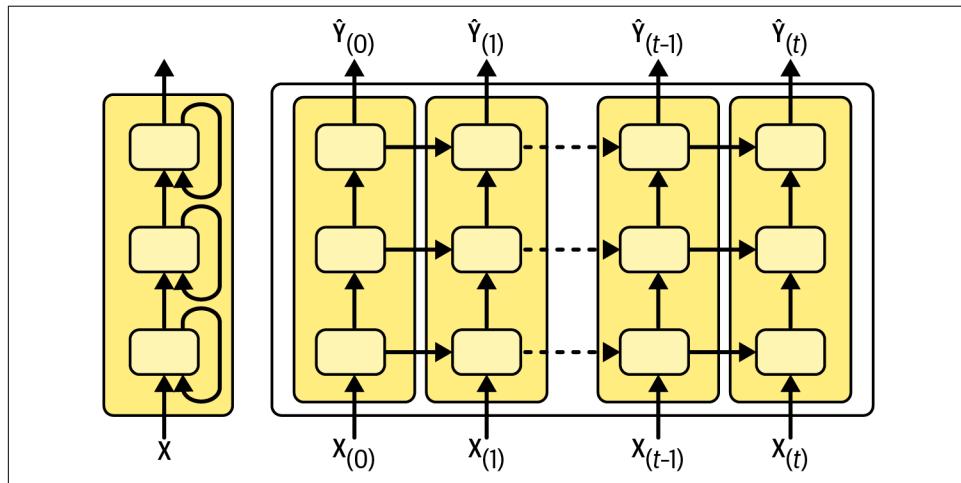
If you train this model, you will get a similar result as before, but generally much faster because the `nn.RNN` module is well optimized. In particular, when using an Nvidia GPU, the `nn.RNN` module leverages the cuDNN library which provides highly optimized implementations of various neural net architectures, including several RNN architectures.



The `nn.RNN` module uses two bias parameters: one for the inputs, and the other for the hidden states. It just adds them up, so this really doesn't improve the model at all, but this extra parameter is required by the cuDNN library. This explains why you won't get exactly the same results as before.

## Forecasting Using a Deep RNN

It is quite common to stack multiple layers of cells, as shown in [Figure 13-10](#). This gives you a *deep RNN*.



*Figure 13-10. A deep RNN (left) unrolled through time (right)*

Implementing a deep RNN with PyTorch is straightforward: just set the `num_layers` argument to the desired number of recurrent layers when creating the `nn.RNN` module. For example, if you set `num_layers=3` when creating the `nn.RNN` module in the previous model's constructor, you get a three-layer RNN (the rest of the code remains unchanged):

```
self.rnn = nn.RNN(input_size, hidden_size, num_layers=3, batch_first=True)
```

If you train and evaluate this model, you will find that it reaches an MAE of 29,273. That's our best model so far!

## Forecasting Multivariate Time Series

An important quality of neural networks is their flexibility: in particular, they can deal with multivariate time series with almost no change to their architecture. For example, let's try to forecast the rail time series using both the rail and bus data as input. In fact, let's also throw in the day type! Since we can always know in advance whether tomorrow is going to be a weekday, a weekend, or a holiday, we can shift the day type series one day into the future, so that the model is given tomorrow's day type as input. For simplicity, we'll do this processing using Pandas:

```
df_mulvar = df[["rail", "bus"]] / 1e6 # use both rail & bus series as input
df_mulvar["next_day_type"] = df["day_type"].shift(-1) # we know tomorrow's type
df_mulvar = pd.get_dummies(df_mulvar, dtype=float) # one-hot encode day type
```

Now `df_mulvar` is a DataFrame with five columns: the rail and bus data, plus three columns containing the one-hot encoding of the next day's type (recall that there are three possible day types, W, A, and U). Next, we can proceed much like we did earlier. First we split the data into three periods, scale it down by a factor of one million, and convert it to tensors:

```
mulvar_train = torch.FloatTensor(df_mulvar["2016-01":"2018-12"].values / 1e6)
mulvar_valid = torch.FloatTensor(df_mulvar["2019-01":"2019-05"].values / 1e6)
mulvar_test = torch.FloatTensor(df_mulvar["2019-06":].values / 1e6)
```

Then we need to create the PyTorch datasets. If we used the `TimeSeriesDataset` for this, the targets would include the next day's rail and bus ridership, as well as the one-hot encoding of the following day type. Since we only want to predict the rail ridership for now, we must tweak the `TimeSeriesDataset` to keep only the first value in the target, which is the rail ridership. One way to do this is to create a new `MulvarTimeSeriesDataset` class that extends the `TimeSeriesDataset` class and tweaks the `__getitem__()` method to filter the target:

```
class MulvarTimeSeriesDataset(TimeSeriesDataset):
    def __getitem__(self, idx):
        window, target = super().__getitem__(idx)
        return window, target[:1]
```

Next, we can create the datasets and the data loaders, much like we did earlier:

```
mulvar_train_set = MulvarTimeSeriesDataset(mulvar_train, window_length)
mulvar_train_loader = DataLoader(mulvar_train_set, batch_size=32, shuffle=True)
[...] # create the datasets and data loaders for the validation and test sets
```

If you look at the batches produced by the data loaders, you will find that the input shape is [32, 56, 5], and the target shape is [32, 1]. Perfect!

So we can finally create the RNN:

```
torch.manual_seed(42)
mulvar_model = SimpleRnnModel(input_size=5, hidden_size=32, output_size=1)
mulvar_model = mulvar_model.to(device)
```

Notice that this model is identical to the `univar_model` RNN we built earlier, except `input_size=5`: at each time step, the model now receives five inputs instead of one. This model actually reaches a validation MAE of 23,227. Now we're making big progress!

In fact, it's not too hard to make the RNN forecast both the rail and bus ridership. You just need to return `target[:2]` instead of `target[:1]` in the `MulvarTimeSeriesDataset` class, and set `output_size=2` when creating the `SimpleRnnModel`, that's all there is to it!

As we discussed in [Chapter 9](#), using a single model for multiple related tasks often results in better performance than using a separate model for each task, since features learned for one task may be useful for the other tasks, and also because having to perform well across multiple tasks prevents the model from overfitting (it's a form of regularization). However, it depends on the task, and in this particular case the multitask RNN that forecasts both the bus and the rail ridership doesn't perform quite as well as dedicated models that forecast one or the other (using all five columns as input). Still, it reaches a validation MAE of 26,441 for rail and 26,178 for bus, which is still pretty good.

## Forecasting Several Time Steps Ahead

So far we have only predicted the value at the next time step, but we could just as easily have predicted the value several steps ahead by changing the targets appropriately (e.g., to predict the ridership 2 weeks from now, we could just change the targets to be the value 14 days ahead instead of 1 day ahead). But what if we want to predict the next 14 values with a single model?

The first option is to take the `univar_model` RNN we trained earlier for the rail time series, make it predict the next value, and add that value to the inputs, acting as if the predicted value had actually occurred. We then use the model again to predict the following value, and so on, as in the following code:

```

n_steps = 14
univar_model.eval()
with torch.no_grad():
    X = rail_valid[:window_length].unsqueeze(dim=0).to(device)
    for step_ahead in range(n_steps):
        y_pred_one = univar_model(X)
        X = torch.cat([X, y_pred_one.unsqueeze(dim=0)], dim=1)

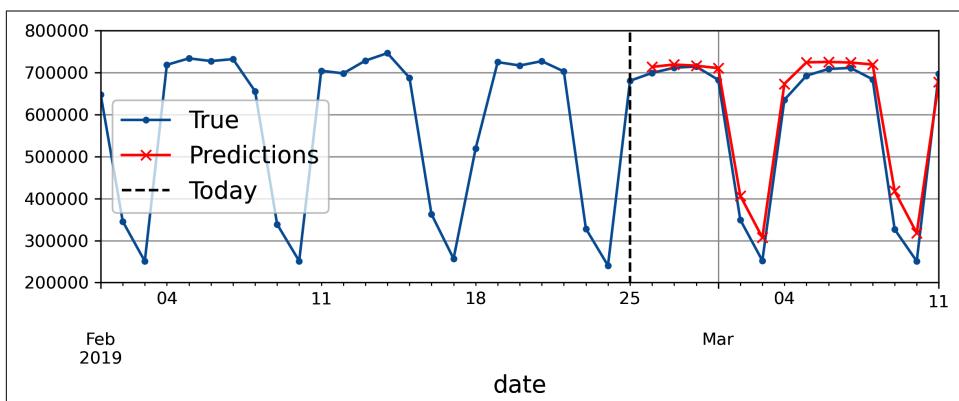
Y_pred = X[0, -n_steps:, 0]

```

In this code, we take the rail ridership of the first 56 days of the validation period, we add a batch dimension of size 1 using the `unsqueeze()` method (since our `univar_model` expects 3D inputs), then we move the tensor to the GPU. Now the shape of `X` is [1, 56, 1]. Then we repeatedly use the model to forecast the next value, and we append each forecast to the input series, along the time axis (`dim=1`). Since each prediction has a shape of [1, 1], we must use `unsqueeze()` again to add a batch dimension of size 1 before we can concatenate it to `X`. In the end, `X` has a shape of [1, 56 + 14, 1], and our final forecasts are the last 14 values of `X`. The resulting forecasts are plotted in [Figure 13-11](#).



If the model makes an error at one time step, then the forecasts for the following time steps are impacted as well: the errors tend to accumulate. So, it's preferable to use this technique only for a small number of steps.



*Figure 13-11. Forecasting 14 steps ahead, 1 step at a time*

The second option is to train an RNN to predict the next 14 values in one shot. We can still use a sequence-to-vector model, but it will output 14 values instead of 1. However, we first need to change the targets to be vectors containing the next 14 values. For this, we can create the following class:

```

class ForecastAheadDataset(TimeSeriesDataset):
    def __len__(self):
        return len(self.series) - self.window_length - 14 + 1

    def __getitem__(self, idx):
        end = idx + self.window_length # 1st index after window
        window = self.series[idx : end]
        target = self.series[end : end + 14, 0] # 0 = rail ridership
        return window, target

```



I've hardcoded the number 14, but in a real project you should make this configurable (e.g., just like the `window_length`).

This class inherits from the `TimeSeriesDataset` class and tweaks its `__len__()` and `__getitem__()` methods. The target is now a tensor containing the next 14 rail ridership values, rather than just the next value. We can once again create a training set, a validation set, and a test set, based on the multivariate time series we built earlier:

```

ahead_train_set = ForecastAheadDataset(mulvar_train, window_length)
ahead_train_loader = DataLoader(ahead_train_set, batch_size=32, shuffle=True)
[...] # create the datasets and data loaders for the validation and test sets

```

Lastly, we can create a simple RNN, just like the `mulvar_model`, but with `output_size=14`:

```

torch.manual_seed(42)
ahead_model = SimpleRnnModel(input_size=5, hidden_size=32, output_size=14)
ahead_model = ahead_model.to(device)

```

After training this model, you can predict the next 14 values at once, like this:

```

ahead_model.eval()
with torch.no_grad():
    window = mulvar_valid[:window_length] # shape [56, 5]
    X = window.unsqueeze(dim=0)          # shape [1, 56, 5]
    Y_pred = ahead_model(X.to(device))   # shape [1, 14]

```

This approach works quite well. Its forecasts for the next day are obviously better than its forecasts for 14 days into the future, but it doesn't accumulate errors like the previous approach did. Now let's see whether a sequence-to-sequence model can do even better.



You can combine both approaches to forecast many steps ahead: use a model that forecasts the next 14 days in one shot, then append the forecasts to the inputs and run the model again to get forecasts for the following 14 days, and so on.<sup>8</sup>

## Forecasting Using a Sequence-to-Sequence Model

Instead of training the model to forecast the next 14 values only at the very last time step, we can train it to forecast the next 14 values at each and every time step. To be clear, at time step 0 the model will output a vector containing the forecasts for time steps 1 to 14, then at time step 1 the model will forecast time steps 2 to 15, and so on. In other words, the targets are sequences of consecutive windows, shifted by one time step at each time step. The target for each input window is not a vector anymore, but a sequence of the same length as the inputs, containing a 14-dimensional vector at each step. Given an input batch of shape  $[batch\ size, window\ length, input\ size]$ , the output will have a shape of  $[batch\ size, window\ length, output\_size]$ . This is no longer a sequence-to-vector RNN, it's a sequence-to-sequence (or *seq2seq*) RNN.



It may be surprising that the targets contain values that appear in the inputs (except for the last time step). Isn't that cheating? Fortunately, not at all: at each time step, an RNN only knows about past time steps; it cannot look ahead. It is said to be a *causal* model.

You may be wondering why we would want to train a seq2seq model when we're really only interested in forecasting future values, which are output by our model at the very last time step. And you're right: after training, you can actually ignore all outputs except for the very last time step. The main advantage of this technique is that the loss will contain a term for the output of the RNN at each and every time step, not just for the output at the last time step. This means there will be many more error gradients flowing through the model, and they won't have to flow through time as much since they will come from the output of each time step, not just the last one. This can both stabilize training and speed up convergence. Moreover, since the model must make predictions at each time step, it will see input sequences of varying lengths, which can reduce the risk of overfitting the model to the specific window length used during training. Well, at least that's the hope! Let's give this technique a try on the rail ridership time series. As usual, we first need to prepare the dataset:

---

<sup>8</sup> We cannot use the `ahead_model` for this because it needs both the rail and bus ridership as input, but it only forecasts the rail ridership.

```

class Seq2SeqDataset(ForecastAheadDataset):
    def __getitem__(self, idx):
        end = idx + self.window_length # 1st index after window
        window = self.series[idx : end]
        target_period = self.series[idx + 1 : end + 14, 0]
        target = target_period.unfold(dimension=0, size=14, step=1)
        return window, target

```

Our new `Seq2SeqDataset` class inherits from the `ForecastAheadDataset` class and overrides the `__getitem__()` method: the input window is defined just like before, but the target is now a sequence of consecutive windows, shifted by one time step at each time step. The `unfold()` method is where the magic happens: it takes a tensor and produces sliding blocks from it. For this, it repeatedly slides along the given dimension by the given number of steps and extracts a block of the given size. For example:

```

>>> torch.tensor([0, 1, 2, 3, 4, 5]).unfold(dimension=0, size=4, step=1)
tensor([[0, 1, 2, 3],
        [1, 2, 3, 4],
        [2, 3, 4, 5]])

```

Once again we must create a training set, a validation set, and a test set, as well as the corresponding data loaders:

```

seq_train_set = Seq2SeqDataset(mulvar_train, window_length)
seq_train_loader = DataLoader(seq_train_set, batch_size=32, shuffle=True)
[...] # create the datasets and data loaders for the validation and test sets

```

And lastly, we can build the sequence-to-sequence model:

```

class Seq2SeqRnnModel(SimpleRnnModel):
    def forward(self, X):
        outputs, last_state = self.rnn(X)
        return self.output(outputs)

torch.manual_seed(42)
seq_model = Seq2SeqRnnModel(input_size=5, hidden_size=32, output_size=14)
seq_model = seq_model.to(device)

```

We inherit from the `SimpleRnnModel` class, and we override the `forward()` method. Instead of applying the linear `self.output` layer only to the outputs of the last time step, as we did before, we now apply it to the outputs of every time step. It may surprise you that this works at all. So far, we have only applied `nn.Linear` layers to 2D inputs of shape  $[batch\ size, features]$ , but here the `outputs` tensor has a shape of  $[batch\ size, window\ length, hidden\ size]$ : it's 3D, not 2D! Luckily, this works fine as the `nn.Linear` layer will automatically be applied to each time step, so the model's predictions will have a shape of  $[batch\ size, window\ length, output\ size]$ : just what we need.

Under the hood, the `nn.Linear` layer relies on `torch.matmul()` for matrix multiplication. This function efficiently supports multiplying arrays of more than two dimensions. For example, you can multiply an array of shape [2, 3, 5, 7] with an array of shape [2, 3, 7, 11]. Indeed, these two arrays can both be seen as  $2 \times 3$  grids of matrices, and `torch.matmul()` simply multiplies the corresponding matrices in both grids. Since multiplying a  $5 \times 7$  matrix with a  $7 \times 11$  matrix produces a  $5 \times 11$  matrix, the final result is a  $2 \times 3$  grid of  $5 \times 11$  matrices, represented as a tensor of shape [2, 3, 5, 11]. Broadcasting is also supported; for example, you can multiply an array of shape [10, 56, 32] with an array of shape [32, 14]: each of the ten  $56 \times 32$  matrices in the first array will be multiplied by the same  $32 \times 14$  matrix in the second array, and you will get a tensor of shape [10, 56, 14]. That's what happens when you pass a 3D input to an `nn.Linear` layer.



Another way to get the exact same result is to replace the `nn.Linear` output layer with an `nn.Conv1d` layer using a kernel size of one (i.e., `Conv1d(32, 14, kernel_size=1)`). However, you would have to swap the last two dimensions of both the inputs and the outputs, treating the time dimension as a spatial dimension.

The training code is the same as usual. During training, all the model's outputs are used, but after training, only the outputs of the very last time step matter, and the rest can be ignored (as mentioned earlier). For example, we can forecast the rail ridership for the next 14 days like this:

```
seq_model.eval()
with torch.no_grad():
    some_window = mulvar_valid[:window_length] # shape [56, 5]
    X = some_window.unsqueeze(dim=0)           # shape [1, 56, 5]
    Y_preds = seq_model(X.to(device))         # shape [1, 56, 14]
    Y_pred = Y_preds[:, -1]                   # shape [1, 14]
```

If you evaluate this model's forecasts for  $t + 1$ , you will find a validation MAE of 23,350, which is very good. Of course, the model is not as accurate for more distant forecasts. For example, the MAE for  $t + 14$  is 35,315.

Simple RNNs can be quite good at forecasting time series or handling other kinds of sequences, but they do not perform as well on long time series or sequences. Let's discuss why and see what we can do about it.

## Handling Long Sequences

To train an RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network. Just like any deep neural network, it may suffer from the unstable gradients problem, discussed in [Chapter 11](#): it may take forever to train, or training may be unstable. Moreover, when an RNN processes a

long sequence, it will gradually forget the first inputs in the sequence. Let's look at both these problems, starting with the unstable gradients problem.

## Fighting the Unstable Gradients Problem

Many of the tricks we used in deep nets to alleviate the unstable gradients problem can also be used for RNNs: good parameter initialization, faster optimizers, dropout, and so on. However, nonsaturating activation functions (e.g., ReLU) may not help as much here. In fact, they may actually lead the RNN to be even more unstable during training. Why? Well, suppose gradient descent updates the weights in a way that increases the outputs slightly at the first time step. Because the same weights are used at every time step, the outputs at the second time step may also be slightly increased, and those at the third, and so on until the outputs explode—and a nonsaturating activation function does not prevent that.

You can reduce this risk by using a smaller learning rate, or you can use a saturating activation function like the hyperbolic tangent (this explains why it's the default).

In much the same way, the gradients themselves can explode. If you notice that training is unstable, you may want to monitor the size of the gradients and perhaps use gradient clipping.

Moreover, batch normalization cannot be used as efficiently with RNNs as with deep feedforward nets. In fact, you cannot use it between time steps, only between recurrent layers. To be more precise, it is technically possible to add a BN layer to a memory cell so that it will be applied at each time step (both on the inputs for that time step and on the hidden state from the previous step). However, this implies that the same BN layer will be used at each time step, with the same parameters, regardless of the actual scale and offset of the inputs and hidden state. In practice, this does not yield good results, as was demonstrated by César Laurent et al. in a [2015 paper](#).<sup>9</sup> The authors found that BN was slightly beneficial only when it was applied to the layer's inputs, not to the hidden states. In other words, it was slightly better than nothing when applied between recurrent layers (i.e., vertically in [Figure 13-10](#)), but not within recurrent layers (i.e., horizontally).

Layer norm (introduced in [Chapter 11](#)) tends to work a bit better than BN within recurrent layers. It is usually applied just before the activation function, at each time step. Sadly, PyTorch's `nn.RNN` module does not support LN, so you have to implement the RNN's loop manually (as we did earlier), and apply the `nn.LayerNorm` module at each iteration. This is not too hard, but you do lose the simplicity and speed of the `nn.RNN` module. For example, you can take the first version of our `SimpleRnnModel`

---

<sup>9</sup> César Laurent et al., "Batch Normalized Recurrent Neural Networks", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2016): 2657–2661.

class and add an `nn.LayerNorm` module to the memory cell, just before the `tanh` activation function:

```
self.memory_cell = nn.Sequential(  
    nn.Linear(input_size + hidden_size, hidden_size),  
    nn.LayerNorm(hidden_size),  
    nn.Tanh()  
)
```



Layer norm does not always help; you just have to try it. In general, it works better in *gated RNNs* such as LSTM and GRU (discussed shortly). It is also more likely to help when the time series is preprocessed to remove any seasonality or trend.

Similarly, if you wish to apply dropout between each time step, you must write a custom RNN since the `nn.RNN` module does not support that. However, it does support adding a dropout layer after every recurrent layer: simply set the `dropout` hyperparameter to the desired dropout rate (it defaults to zero).

With these techniques, you can alleviate the unstable gradients problem and train an RNN much more efficiently. Now let's look at how to deal with the short-term memory problem.



When forecasting time series, it is often useful to have some error bars along with your predictions. For this, one approach is to use MC dropout (introduced in [Chapter 11](#)).

## Tackling the Short-Term Memory Problem

Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step. After a while, the RNN's state contains virtually no trace of the first inputs. This can be a showstopper. Imagine Dory the fish<sup>10</sup> trying to translate a long sentence; by the time she's finished reading it, she has no clue how it started. To tackle this problem, various types of cells with long-term memory have been introduced. They have proven so successful that the basic cells are not used much anymore. Let's first look at the most popular of these long-term memory cells: the LSTM cell.

---

<sup>10</sup> A character from the animated movies *Finding Nemo* and *Finding Dory* who has short-term memory loss.

## LSTM cells

The *long short-term memory* (LSTM) cell was proposed in 1997<sup>11</sup> by Sepp Hochreiter and Jürgen Schmidhuber and gradually improved over the years by several researchers, such as Alex Graves, Haşim Sak,<sup>12</sup> and Wojciech Zaremba.<sup>13</sup> You can simply use the `nn.LSTM` module instead of the `nn.RNN` module; it's a drop-in replacement, and it usually performs much better: training converges faster, and the model detects longer-term patterns in the data.

So how does this magic work? Well, the LSTM architecture is shown in [Figure 13-12](#). If you don't look at what's inside the box, the LSTM cell looks exactly like a regular cell, except that its state is split into two vectors:  $\mathbf{h}_{(t)}$  and  $\mathbf{c}_{(t)}$  ("c" stands for "cell"). You can think of  $\mathbf{h}_{(t)}$  as the short-term state and  $\mathbf{c}_{(t)}$  as the long-term state.

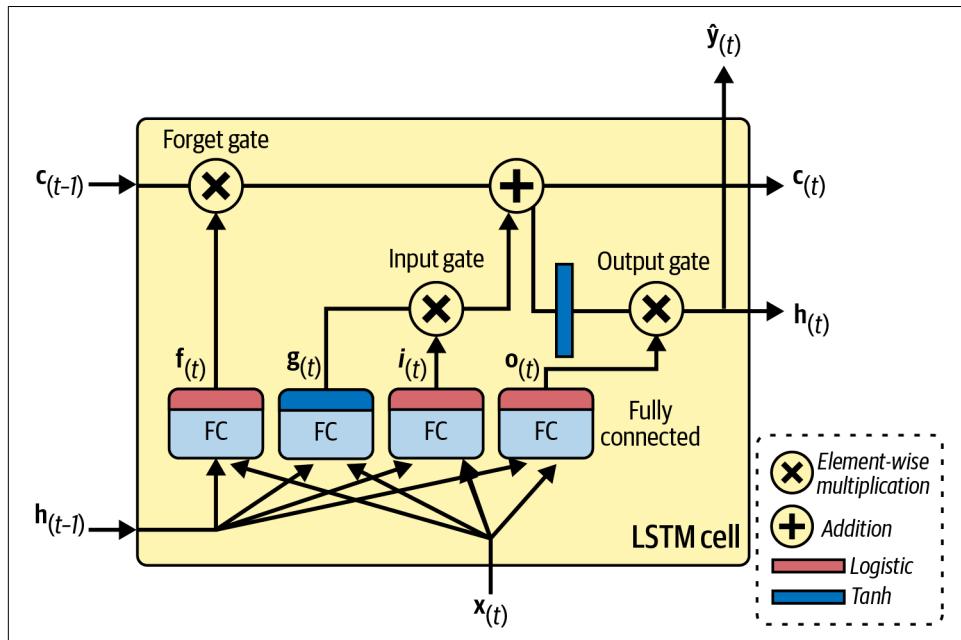


Figure 13-12. An LSTM cell

Now let's open the box! The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it. As the long-term state

<sup>11</sup> Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory", *Neural Computation* 9, no. 8 (1997): 1735–1780.

<sup>12</sup> Haşim Sak et al., "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition", arXiv preprint arXiv:1402.1128 (2014).

<sup>13</sup> Wojciech Zaremba et al., "Recurrent Neural Network Regularization", arXiv preprint arXiv:1409.2329 (2014).

$\mathbf{c}_{(t-1)}$  traverses the network from left to right, you can see that it first goes through a *forget gate*, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*). The result  $\mathbf{c}_{(t)}$  is sent straight out without any further transformation. So at each time step, some memories are dropped and some memories are added. Moreover, after the addition operation, the long-term state is copied and passed through the tanh function, and the result is filtered by the *output gate*. This produces the short-term state  $\mathbf{h}_{(t)}$  (which is equal to the cell's output for this time step,  $\mathbf{y}_{(t)}$ ). Now let's look at where new memories come from and how the gates work.

First, the current input vector  $\mathbf{x}_{(t)}$  and the previous short-term state  $\mathbf{h}_{(t-1)}$  are fed to four different fully connected layers. They all serve a different purpose:

- The main layer is the one that outputs  $\mathbf{g}_{(t)}$ . It has the usual role of analyzing the current inputs  $\mathbf{x}_{(t)}$  and the previous (short-term) state  $\mathbf{h}_{(t-1)}$ . In a simple RNN cell, there is nothing other than this layer, and its output goes straight out to  $\mathbf{y}_{(t)}$  and  $\mathbf{h}_{(t)}$ . But in an LSTM cell, this layer's output does not go straight out; instead its most important parts are stored in the long-term state (and the rest is dropped).
- The three other layers are *gate controllers*. Since they use the logistic activation function, the outputs range from 0 to 1. As you can see, the gate controllers' outputs are fed to element-wise multiplication operations: if they output 0s they close the gate, and if they output 1s they open it. Specifically:
  - The *forget gate* (controlled by  $\mathbf{f}_{(t)}$ ) controls which parts of the long-term state should be erased.
  - The *input gate* (controlled by  $\mathbf{i}_{(t)}$ ) controls which parts of  $\mathbf{g}_{(t)}$  should be added to the long-term state.
  - Finally, the *output gate* (controlled by  $\mathbf{o}_{(t)}$ ) controls which parts of the long-term state should be read and output at this time step, both to  $\mathbf{h}_{(t)}$  and to  $\mathbf{y}_{(t)}$ .

In short, an LSTM cell can learn to recognize an important input (that's the role of the input gate), store it in the long-term state, preserve it for as long as it is needed (that's the role of the forget gate), and extract it whenever it is needed (that's the role of the output gate), all while being fully differentiable. This explains why these cells have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

[Equation 13-4](#) summarizes how to compute the cell's long-term state, its short-term state, and its output at each time step for a single instance (the equations for a whole mini-batch are very similar).

*Equation 13-4. LSTM computations*

$$\begin{aligned}\mathbf{i}_{(t)} &= \sigma(\mathbf{W}_{xi}^T \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \mathbf{h}_{(t-1)} + \mathbf{b}_i) \\ \mathbf{f}_{(t)} &= \sigma(\mathbf{W}_{xf}^T \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \mathbf{h}_{(t-1)} + \mathbf{b}_f) \\ \mathbf{o}_{(t)} &= \sigma(\mathbf{W}_{xo}^T \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \mathbf{h}_{(t-1)} + \mathbf{b}_o) \\ \mathbf{g}_{(t)} &= \tanh(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \mathbf{h}_{(t-1)} + \mathbf{b}_g) \\ \mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\ \mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)})\end{aligned}$$

In this equation:

- $\mathbf{W}_{xi}$ ,  $\mathbf{W}_{xf}$ ,  $\mathbf{W}_{xo}$ , and  $\mathbf{W}_{xg}$  are the weight matrices of each of the four layers for their connection to the input vector  $\mathbf{x}_{(t)}$ .
- $\mathbf{W}_{hi}$ ,  $\mathbf{W}_{hf}$ ,  $\mathbf{W}_{ho}$ , and  $\mathbf{W}_{hg}$  are the weight matrices of each of the four layers for their connection to the previous short-term state  $\mathbf{h}_{(t-1)}$ .
- $\mathbf{b}_i$ ,  $\mathbf{b}_f$ ,  $\mathbf{b}_o$ , and  $\mathbf{b}_g$  are the bias terms for each of the four layers.



Try replacing `nn.RNN` with `nn.LSTM` in the previous models and see what performance you can reach on the ridership dataset (a bit of hyperparameter tuning may be required).

Just like for simple RNNs, if you want to add layer normalization or dropout at each time step, you must implement the recurrent loop manually. One option is to use [Equation 13-4](#), but a simpler option is to use the `nn.LSTMCell` module, which runs a single time step. For example, here is a simple implementation:

```
class LstmModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.memory_cell = nn.LSTMCell(input_size, hidden_size)
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, X):
        batch_size, window_length, dimensionality = X.shape
        X_time_first = X.transpose(0, 1)
```

```

H = torch.zeros(batch_size, self.hidden_size, device=X.device)
C = torch.zeros(batch_size, self.hidden_size, device=X.device)
for X_t in X_time_first:
    H, C = self.memory_cell(X_t, (H, C))
return self.output(H)

```

This is very similar to the first implementation of our `SimpleRnnModel`, but we are now using an `nn.LSTMCell` at each time step, and the hidden state is now split in two parts: the short-term  $H$  and the long-term  $C$ .

There are several variants of the LSTM cell. One particularly popular variant is the GRU cell, which we will look at now.

### GRU cells

The *gated recurrent unit* (GRU) cell (see [Figure 13-13](#)) was proposed by Kyunghyun Cho et al. in a [2014 paper](#)<sup>14</sup> that also introduced the encoder-decoder network we discussed earlier.

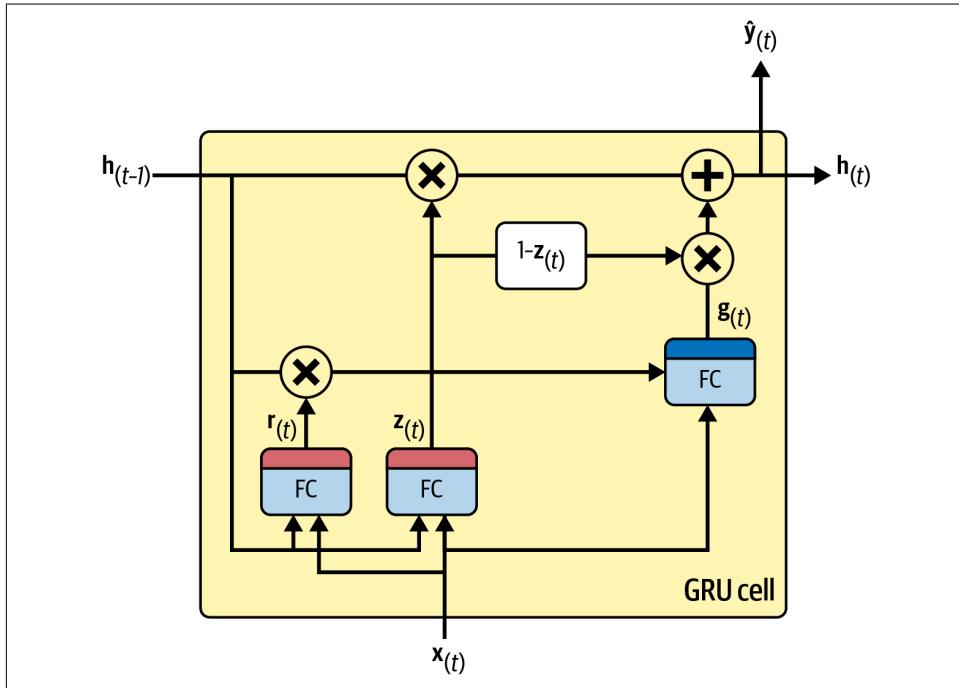


Figure 13-13. GRU cell

<sup>14</sup> Kyunghyun Cho et al., “Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation”, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (2014): 1724–1734.

The GRU cell is a simplified version of the LSTM cell, and it often performs just as well.<sup>15</sup> These are the main simplifications:

- Both state vectors are merged into a single vector  $\mathbf{h}_{(t)}$ .
- A single gate controller  $\mathbf{z}_{(t)}$  controls both the forget gate and the input gate. If the gate controller outputs a 1, the forget gate is open ( $= 1$ ) and the input gate is closed ( $1 - 1 = 0$ ). If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first.
- There is no output gate; the full state vector is output at every time step. However, there is a new gate controller  $\mathbf{r}_{(t)}$  that controls which part of the previous state will be shown to the main layer ( $\mathbf{g}_{(t)}$ ).

[Equation 13-5](#) summarizes how to compute the cell's state at each time step for a single instance.

*Equation 13-5. GRU computations*

$$\begin{aligned}\mathbf{z}_{(t)} &= \sigma(\mathbf{W}_{xz}^T \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \mathbf{h}_{(t-1)} + \mathbf{b}_z) \\ \mathbf{r}_{(t)} &= \sigma(\mathbf{W}_{xr}^T \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \mathbf{h}_{(t-1)} + \mathbf{b}_r) \\ \mathbf{g}_{(t)} &= \tanh(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g) \\ \mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}\end{aligned}$$

PyTorch provides an `nn.GRU` layer; using it is just a matter of replacing `nn.RNN` or `nn.LSTM` with `nn.GRU`. It also provides an `nn.GRUCell` in case you want to create a custom RNN based on a GRU cell (just replace `nn.LSTMCell` with `nn.GRUCell` in the previous example, and get rid of `C`).

LSTM and GRU are one of the main reasons behind the success of RNNs. Yet while they can tackle much longer sequences than simple RNNs, they still have a fairly limited short-term memory, and they have a hard time learning long-term patterns in sequences of 100 time steps or more, such as audio samples, long time series, or long sentences. One way to solve this is to shorten the input sequences, for example, using 1D convolutional layers.

---

<sup>15</sup> See Klaus Greff et al., “[LSTM: A Search Space Odyssey](#)”, *IEEE Transactions on Neural Networks and Learning Systems* 28, no. 10 (2017): 2222–2232. This paper seems to show that all LSTM variants perform roughly the same.

## Using 1D convolutional layers to process sequences

In Chapter 12, we saw that a 2D convolutional layer works by sliding several fairly small kernels (or filters) across an image, producing multiple 2D feature maps (one per kernel). Similarly, a 1D convolutional layer slides several kernels across a sequence, producing a 1D feature map per kernel. Each kernel will learn to detect a single very short sequential pattern (no longer than the kernel size). If you use 10 kernels, then the layer's output will be composed of 10 1D sequences (all of the same length), or equivalently you can view this output as a single 10D sequence. This means that you can build a neural network composed of a mix of recurrent layers and 1D convolutional layers (or even 1D pooling layers). However, as mentioned earlier, you must swap the last two dimensions of the `nn.Conv1d` layer's inputs and outputs, since the `nn.Conv1d` layer expects inputs of shape `[batch size, input features, sequence length]`, and produces outputs of shape `[batch size, output features, sequence length]`.



If you use a 1D convolutional layer with a stride of 1 and "same" padding, then the output sequence will have the same length as the input sequence. But if you use "valid" padding or a stride greater than 1, then the output sequence will be shorter than the input sequence, so make sure you adjust the targets accordingly.

For example, the following model is composed of a 1D convolutional layer, followed by a GRU layer, and lastly a linear output layer, all of which input and output batches of sequences (i.e., 3D tensors). The `nn.Conv1d` layer downsamples the input sequences by a factor of 2, using a stride of 2. The kernel size is as large as the stride (larger, in fact), so all inputs will be used to compute the layer's output, and therefore the model can learn to preserve the most useful information, dropping only the unimportant details. In the `forward()` method, we just chain the layers, but we permute the last two dimensions before and after the `nn.Conv1d` layer, and we ignore the hidden states returned by the `nn.GRU` layer:

```
class DownsamplingModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.conv = nn.Conv1d(input_size, hidden_size, kernel_size=4, stride=2)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, X):
        Z = X.permute(0, 2, 1) # treat time as a spatial dimension
        Z = self.conv(Z)
        Z = Z.permute(0, 2, 1) # swap back time & features dimensions
        Z = torch.relu(Z)
        Z, _states = self.gru(Z)
        return self.linear(Z)
```

```
torch.manual_seed(42)
dseq_model = DownsamplingModel(input_size=5, hidden_size=32, output_size=14)
dseq_model = dseq_model.to(device)
```

By shortening the time series, the convolutional layer helps the GRU layer detect longer patterns, so we can afford to double the window length to 112 days. Note that we must also crop off the first three time steps from the targets: indeed, the kernel's size is 4, so the first output of the convolutional layer will be based on the input time steps 0 to 3, therefore the first forecasts must be for time steps 4 to 17 (instead of time steps 1 to 14). Moreover, we must downsample the targets by a factor of 2 because of the stride. For all this, we need a new `Dataset` class, so let's create a subclass of the `Seq2SeqDataset` class:

```
class DownsampledDataset(Seq2SeqDataset):
    def __getitem__(self, idx):
        window, target = super().__getitem__(idx)
        return window, target[3::2] # crop the first 3 targets and downsample

    window_length = 112
    dseq_train_set = DownsampledDataset(rail_train, window_length)
    dseq_train_loader = DataLoader(dseq_train_set, batch_size=32, shuffle=True)
    [...] # create the datasets and data loaders for the validation and test sets
```

And now the model can be trained as usual. We've successfully mixed convolutional layers and recurrent layers. But what if we used only 1D convolutional layers and dropped the recurrent layers entirely?

## WaveNet

In a [2016 paper](#),<sup>16</sup> Aaron van den Oord and other DeepMind researchers introduced a novel architecture called *WaveNet*. They stacked 1D convolutional layers, doubling the dilation rate (how spread apart each neuron's inputs are) at every layer: the first convolutional layer gets a glimpse of just two time steps at a time, while the next one sees four time steps (its receptive field is four time steps long), the next one sees eight time steps, and so on (see [Figure 13-14](#)). This way, the lower layers learn short-term patterns, while the higher layers learn long-term patterns. Thanks to the doubling dilation rate, the network can process extremely large sequences very efficiently.

---

<sup>16</sup> Aaron van den Oord et al., “WaveNet: A Generative Model for Raw Audio”, arXiv preprint arXiv:1609.03499 (2016).

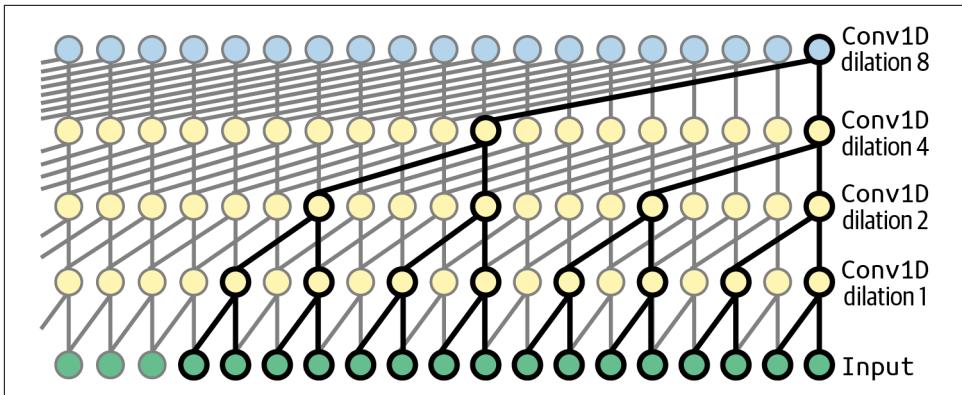


Figure 13-14. WaveNet architecture

The authors of the paper actually stacked 10 convolutional layers with dilation rates of 1, 2, 4, 8, ..., 256, 512, then they stacked another group of 10 identical layers (also with dilation rates 1, 2, 4, 8, ..., 256, 512), then again another identical group of 10 layers. They justified this architecture by pointing out that a single stack of 10 convolutional layers with these dilation rates will act like a super-efficient convolutional layer with a kernel of size 1,024 (except way faster, more powerful, and using significantly fewer parameters). They also left-padded the input sequences with a number of zeros equal to the dilation rate before every layer to preserve the same sequence length throughout the network. Padding on the left rather than on both sides is important, as it ensures that the convolutional layer does not peek into the future when making predictions. This makes it a causal model.

Let's implement a simplified WaveNet to tackle the same sequences as earlier.<sup>17</sup> We will start by creating a custom `CausalConv1d` module that acts just like an `nn.Conv1d` module, except the inputs get padded on the left side by the appropriate amount to ensure the sequence preserves the same length:

```
import torch.nn.functional as F

class CausalConv1d(nn.Conv1d):
    def forward(self, X):
        padding = (self.kernel_size[0] - 1) * self.dilation[0]
        X = F.pad(X, (padding, 0))
        return super().forward(X)
```

In this code, we inherit from the `nn.Conv1d` class and we override the `forward()` method. In it, we calculate the size of the left-padding we need, and we pad the sequences using the `pad()` function before calling the base class's `forward()` method.

<sup>17</sup> The complete WaveNet uses a few more tricks, such as skip connections like in a ResNet, and *gated activation units* similar to those found in a GRU cell.

The `pad()` function takes two arguments: the tensor to pad (`X`), and a tuple of ints that indicates how much to pad to the left and right in the last dimension (i.e., the time dimension).

Now we're ready to build the WaveNet model itself:

```
class WavenetModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        layers = []
        for dilation in (1, 2, 4, 8) * 2:
            conv = CausalConv1d(input_size, hidden_size, kernel_size=2,
                                dilation=dilation)
            layers += [conv, nn.ReLU()]
            input_size = hidden_size
        self.convs = nn.Sequential(*layers)
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, X):
        Z = X.permute(0, 2, 1)
        Z = self.convs(Z)
        Z = Z.permute(0, 2, 1)
        return self.output(Z)

torch.manual_seed(42)
wavenet_model = WavenetModel(input_size=5, hidden_size=32, output_size=14)
wavenet_model = wavenet_model.to(device)
```

In the constructor, we create eight `CausalConv1d` layers with various dilation rates (1, 2, 4, 8, then again 1, 2, 4, 8), each followed by the `ReLU` activation function. We chain all these modules in an `nn.Sequential` module `self.convs`. We also create the output `nn.Linear` layer. In the forward method, we permute the last two dimensions of the inputs, as we did earlier, we then pass them through the convolutional layers, then we permute the last two dimensions back to their original order, and we pass the result through the output layer. Thanks to the causal padding, every convolutional layer outputs a sequence of the same length as its input sequence, so the targets we use during training can be the full 112-day sequences; no need to crop them or downsample them. Thus, we can train the model using the data loaders we built for the `Seq2SeqModel` (i.e., `seq_train_loader` and `seq_valid_loader`).

The models we've discussed in this section offer similar performance for the ridership forecasting task, but they may vary significantly depending on the task and the amount of available data. In the WaveNet paper, the authors achieved state-of-the-art performance on various audio tasks (hence the name of the architecture), including text-to-speech tasks, producing very realistic voices across several languages. They also used the model to generate music, one audio sample at a time. This feat is all the more impressive when you realize that a single second of audio can contain tens of thousands of time steps—even LSTMs and GRUs cannot handle such long sequences.



If you evaluate our best Chicago ridership models on the test period, starting in 2020, you will find that they perform much worse than expected! Why is that? Well, that's when the Covid-19 pandemic started, which greatly affected public transportation. As mentioned earlier, these models will only work well if the patterns they learned from the past continue in the future. In any case, before deploying a model to production, verify that it works well on recent data. And once it's in production, make sure to monitor its performance regularly.

With that, you can now tackle all sorts of time series! In [Chapter 14](#), we will continue to explore RNNs, and we will see how they can tackle various NLP tasks as well.

## Exercises

1. Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN, and a vector-to-sequence RNN?
2. How many dimensions must the inputs of an RNN layer have? What does each dimension represent? What about its outputs?
3. How can you build a deep sequence-to-sequence RNN in PyTorch?
4. Suppose you have a daily univariate time series, and you want to forecast the next seven days using an RNN. Which architecture should you use?
5. What are the main difficulties when training RNNs? How can you handle them?
6. Can you sketch the LSTM cell's architecture?
7. Why would you want to use 1D convolutional layers in an RNN?
8. Which neural network architecture could you use to classify videos?
9. Try to tweak the `Seq2SeqModel` model to forecast both rail and bus ridership for the next 14 days. The model will now need to predict 28 values instead of 14.
10. Download the [Bach chorales](#) dataset and unzip it. It is composed of 382 chorales composed by Johann Sebastian Bach. Each chorale is 100 to 640 time steps long, and each time step contains 4 integers, where each integer corresponds to a note's index on a piano (except for the value 0, which means that no note is played). Train a model—recurrent, convolutional, or both—that can predict the next time step (four notes), given a sequence of time steps from a chorale. Then use this model to generate Bach-like music, one note at a time: you can do this by giving the model the start of a chorale and asking it to predict the next time step, then appending these time steps to the input sequence and asking the model for the next note, and so on. Also make sure to check out [Google's Coconet model](#), which was used for a nice Google doodle about Bach.

11. Train a classification model for the [QuickDraw dataset](#), which contains millions of sketches of various objects. Start by downloading the simplified data for a few classes (e.g., `ant.ndjson`, `axe.ndjson`, and `bat.ndjson`). Each NDJSON file contains one JSON object per line, which you can parse using Python's `json.loads()` function. This will give you a list of sketches, where each sketch is represented as a Python dictionary. In each dictionary, the "drawing" entry contains a list of pen strokes. You can convert this list to a 3D float tensor where the dimensions are [*strokes*, *x coordinates*, *y coordinates*]. Since an RNN takes a single sequence as input, you will need to concatenate all the strokes for each sketch into a single sequence. It's best to add an extra feature to allow the RNN to know how far along each stroke it currently is (e.g., from 0 to 1). In other words, the model will receive a sequence where each time step has three features: the *x* and *y* coordinates of the pen, and the progress ratio along the current stroke.
12. Create a dataset containing short audio recordings of you saying “yes” or “no”, and train a binary classification RNN on it. For example, you could:
  - a. Use an audio recording software such as Audacity to record yourself saying “yes” as many times as your patience allows, with short pauses between each word. Create a similar recording for the word “no”. Try to cover the various ways you might realistically pronounce these words in real life.
  - b. Load each WAV file using the `torchaudio.load()` function from the Torch-Audio library. This will return a tensor containing the audio, as well as an integer indicating the number of samples per second. The audio tensor has a shape of [*channels*, *samples*]: one channel for mono, two for stereo. Convert stereo to mono by averaging over the channel dimension.
  - c. Chop each recording into individual words by splitting at the silences. You can do this using the `torchaudio.transforms.Vad` transform (Voice Activity Detection).
  - d. Since the sequences are so long, it's hard to directly train an RNN on them, so it helps to convert the audio to a spectrogram first. For this, you can use the `torchaudio.transforms.MelSpectrogram` transform, which is well suited for voice. The output is a dramatically shorter sequence, with many more channels.
  - e. Now try building and training a binary classification RNN on your yes/no dataset! Consider sharing your dataset and model with the world (e.g., via the Hugging Face Hub).

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.

# Natural Language Processing with RNNs and Attention

When Alan Turing imagined his famous [Turing test](#)<sup>1</sup> in 1950, he proposed a way to evaluate a machine’s ability to match human intelligence. He could have tested for many things, such as the ability to recognize cats in pictures, play chess, compose music, or escape a maze, but, interestingly, he chose a linguistic task. More specifically, he devised a *chatbot* capable of fooling its interlocutor into thinking it was human.<sup>2</sup> This test does have its weaknesses: a set of hardcoded rules can fool unsuspecting or naive humans (e.g., the machine could give vague predefined answers in response to some keywords, it could pretend that it is joking or drunk to get a pass on its weirdest answers, or it could escape difficult questions by answering them with its own questions), and many aspects of human intelligence are utterly ignored (e.g., the ability to interpret nonverbal communication such as facial expressions, or to learn a manual task). But the test does highlight the fact that mastering language is arguably *Homo sapiens*’s greatest cognitive ability.

Until recently, state-of-the-art natural language processing (NLP) models were pretty much all based on recurrent neural networks (introduced in [Chapter 13](#)). However, in recent years, RNNs have been replaced with transformers, which we will explore in [Chapter 15](#). That said, it’s still important to learn how RNNs can be used for NLP tasks, if only because it helps better understand transformers. Moreover, most of the techniques we will discuss in this chapter are also useful with Transformer

---

<sup>1</sup> Alan Turing, “Computing Machinery and Intelligence”, *Mind* 49 (1950): 433–460.

<sup>2</sup> Of course, the word *chatbot* came much later. Turing called his test the *imitation game*: machine A and human B chat with human interrogator C via text messages; the interrogator asks questions to figure out which one is the machine (A or B). The machine passes the test if it can fool the interrogator, while the human B must try to help the interrogator.

architectures (e.g., tokenization, beam search, attention mechanisms, and more). Plus, RNNs have recently made a surprise comeback in the form of state space models (SSMs) (see “State-Space Models (SSMs)” at <https://homl.info>).

This chapter is organized in three sections. In the first section, we will start by building a *character RNN*, or *char-RNN*, trained to predict the next character in a sentence. On the way, we will learn about trainable embeddings. Our char-RNN will be our first tiny *language model*, capable of generating original text.

In the second section, we will turn to text classification, and more specifically sentiment analysis, which aims to predict how positive or negative some text is. Our model will read movie reviews and estimate the rater’s feeling about the movie. This time, instead of splitting the text into individual characters, we will split it into *tokens*: a token is a small piece of text from a fixed-sized vocabulary, such as the top 10,000 most common words in the English language, or the most common subwords (e.g., “smartest” = “smart” + “est”), or even individual characters or bytes. To split the text into tokens, we will use a *tokenizer*. This section will also introduce popular Hugging Face libraries: the *Datasets* library to download datasets, the *Tokenizers* library for tokenizers, and the *Transformers* library for popular models, downloaded automatically from the *Hugging Face Hub*. Hugging Face is a hugely influential company and open source community, and it plays a central role in the open source AI space, especially in NLP.

The final boss of this chapter will be neural machine translation (NMT), the topic of the third and last section: we will build an encoder-decoder model capable of translating English to Spanish. This will lead us to *attention mechanisms*, which we will apply to our encoder-decoder model to improve its capacity to handle long input texts. As their name suggests, attention mechanisms are neural network components that learn to select the part of the inputs that the model should focus on at each time step. They directly led to the transformers revolution, as we will see in the next chapter.

Let’s start with a simple and fun char-RNN model that can write like Shakespeare (sort of).

## Generating Shakespearean Text Using a Character RNN

In a famous [2015 blog post](#) titled “The Unreasonable Effectiveness of Recurrent Neural Networks”, Andrej Karpathy showed how to train an RNN to predict the next character in a sentence. This *char-RNN* can then be used to generate novel text, one character at a time. Here is a small sample of the text generated by a char-RNN model after it was trained on all of Shakespeare’s works:

PANDARUS:

Alas, I think he shall be come approached and the day

When little strain would be attain'd into being never fed,  
And who is but a chain and subjects of his death,  
I should not sleep.

Not exactly a masterpiece, but it is still impressive that the model was able to learn words, grammar, proper punctuation, and more, just by learning to predict the next character in a sentence. This is our first example of a *language model*. In the remainder of this section we'll build a char-RNN step by step, starting with the creation of the dataset.

## Creating the Training Dataset

First, let's download a subset of Shakespeare's works (about 25%). The data is loaded from Andrej Karpathy's [char-rnn project](#):

```
from pathlib import Path
import urllib.request

def download_shakespeare_text():
    path = Path("datasets/shakespeare/shakespeare.txt")
    if not path.is_file():
        path.parent.mkdir(parents=True, exist_ok=True)
        url = "https://homl.info/shakespeare"
        urllib.request.urlretrieve(url, path)
    return path.read_text()

shakespeare_text = download_shakespeare_text()
```

Let's print the first few lines:

```
>>> print(shakespeare_text[:80])
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.
```

Looks like Shakespeare, all right!

Neural networks work with numbers, not text, so we need a way to encode text into numbers. In general, this is done by splitting the text into *tokens*, such as words or characters, and assigning an integer ID to each possible token. For example, let's split our text into characters, and assign an ID to each possible character. We first need to find the list of characters used in the text. This will constitute our token *vocabulary*:

```
>>> vocab = sorted(set(shakespeare_text.lower()))
>>> ''.join(vocab)
"\n !$&',-.3:;?abcdefghijklmnopqrstuvwxyz"
```

Note that we call `lower()` to ignore case and thereby reduce the vocabulary size. We must now assign a token ID to each character. For this, we can just use its index in the vocabulary. To decode the output of our model, we will also need a way to go from a token ID to a character:

```
>>> char_to_id = {char: index for index, char in enumerate(vocab)}
>>> id_to_char = {index: char for index, char in enumerate(vocab)}
>>> char_to_id["a"]
13
>>> id_to_char[13]
'a'
```

Next, let's create two helper functions to encode text to tensors of token IDs, and to decode them back to text:

```
import torch

def encode_text(text):
    return torch.tensor([char_to_id[char] for char in text.lower()])

def decode_text(char_ids):
    return ''.join([id_to_char[char_id.item()] for char_id in char_ids])
```

Let's try them out:

```
>>> encoded = encode_text("Hello, world!")
>>> encoded
tensor([20, 17, 24, 24, 27, 6, 1, 35, 27, 30, 24, 16, 2])
>>> decode_text(encoded)
'hello, world!'
```

Next, let's prepare the dataset. Right now, we have a single, extremely long sequence of characters containing Shakespeare's works. Just like we did in [Chapter 13](#), we can turn this long sequence into a dataset of windows that we can then use to train a sequence-to-sequence RNN. The targets will be similar to the inputs, but shifted by one time step into the "future". For example, one sample in the dataset may be a sequence of character IDs representing the text "to be or not to b" (without the final "e"), and the corresponding target—a sequence of character IDs representing the text "o be or not to be" (with the final "e", but without the leading "t"). Let's create our dataset class:

```
from torch.utils.data import Dataset, DataLoader

class CharDataset(Dataset):
    def __init__(self, text, window_length):
        self.encoded_text = encode_text(text)
        self.window_length = window_length

    def __len__(self):
        return len(self.encoded_text) - self.window_length
```

```

def __getitem__(self, idx):
    if idx >= len(self):
        raise IndexError("dataset index out of range")
    end = idx + self.window_length
    window = self.encoded_text[idx : end]
    target = self.encoded_text[idx + 1 : end + 1]
    return window, target

```

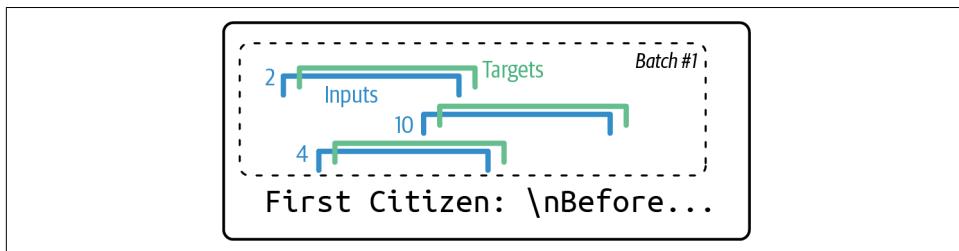
And now let's create the data loaders, as usual. Since the text is quite large, we can afford to use roughly 90% for training (i.e., one million characters), and just 5% for validation, and 5% for testing (60,000 characters each):

```

window_length = 50
batch_size = 512 # reduce if your GPU cannot handle such a large batch size
train_set = CharDataset(shakespeare_text[:1_000_000], window_length)
valid_set = CharDataset(shakespeare_text[1_000_000:1_060_000], window_length)
test_set = CharDataset(shakespeare_text[1_060_000:], window_length)
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=batch_size)
test_loader = DataLoader(test_set, batch_size=batch_size)

```

Each batch will be composed of 512 50-character windows, where each character is represented by its token ID, and where each window comes with its 50-character target window (offset by one character). Note that the training batches are shuffled at each epoch (see [Figure 14-1](#)).



*Figure 14-1. Each training batch is composed of shuffled windows, along with their shifted targets. In this figure, the window length is 10 instead of 50.*



We set the window length to 50, but you can try tuning it. It's easier and faster to train RNNs on shorter input sequences, but the RNN will not be able to learn any pattern longer than the window length, so don't make it too small.

While we could technically feed the token IDs directly to a neural network without any further preprocessing, it wouldn't work very well. Indeed, as we saw in [Chapter 2](#), most ML models—including neural networks—assume that similar inputs represent similar things; unfortunately, similar IDs may represent totally unrelated tokens, and conversely, distant IDs may represent similar tokens. The neural net would be

biased in a weird way, and it would have great difficulty overcoming this bias during training.

One solution is to use one-hot encoding, since all one-hot vectors are equally distant from one another. However, when the vocabulary is large, one-hot vectors are equally large. In our case, the vocabulary contains just 39 characters, so each character would be represented by a 39-dimensional one-hot vector. That's still manageable, but if we were dealing with words instead of characters, the vocabulary size could be in the tens of thousands, so one-hot encoding would be out of the question. Luckily, since we are dealing with neural networks, we have a better option: embeddings.

## Embeddings

An embedding is a dense representation of some higher-dimensional data, typically a categorical feature. If there are 50,000 possible categories, then one-hot encoding produces a 50,000-dimensional sparse vector (i.e., containing mostly zeros). In contrast, an embedding is a comparatively small dense vector; for example, with just 300 dimensions.



The embedding size is a hyperparameter you can tune. As a rule of thumb, a good embedding size is often close to the square root of the number of categories.

In deep learning, embeddings are usually initialized randomly, and they are then trained by gradient descent, along with the other model parameters. For example, if we wanted to train a neural network on the California housing dataset (see [Chapter 2](#)), we could represent the `ocean_proximity` categorical feature using embeddings. The "NEAR BAY" category could be represented initially by a random vector such as `[0.831, 0.696]`, while the "NEAR OCEAN" category might be represented by another random vector such as `[0.127, 0.868]` (in this example we are using 2D embeddings).

Since these embeddings are trainable, they will gradually improve during training; and as they represent fairly similar categories in this example, 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 14-2](#)). 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* (you will see other types of representation learning in [Chapter 18](#)).

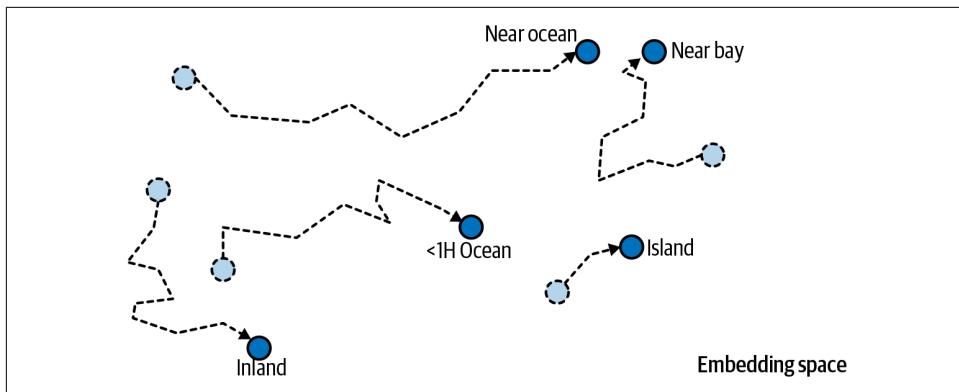


Figure 14-2. Embeddings will gradually improve during training

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, as we will see later in this chapter.

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<sup>3</sup>](#) 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* were clustered together.

It's not just about proximity, though: word embeddings are 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 14-3](#)). In other words, the word embeddings encode the concept of gender! Similarly, you can compute *Madrid* – *Spain* + *France*, and the result is close to *Paris*, which seems to show that the notion of capital city is also encoded in the embeddings.

<sup>3</sup> Tomáš Mikolov et al., “Distributed Representations of Words and Phrases and Their Compositionality”, *Proceedings of the 26th International Conference on Neural Information Processing Systems 2* (2013): 3111–3119.

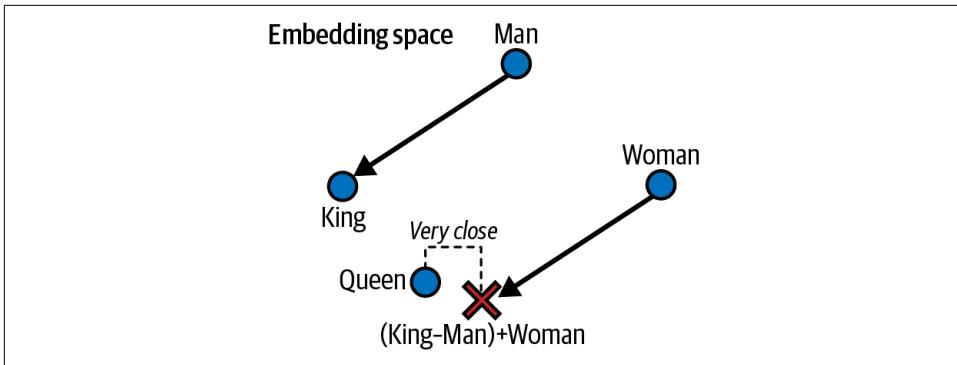


Figure 14-3. Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts



Word embeddings can have some meaningful structure, as the “King – Man + Woman” shows. However, they are also noisy and often hard to interpret. I’ve added some code at the end of the notebook so you can judge for yourself.

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 paper<sup>4</sup>](#) by Malvina Nissim et al. Nevertheless, ensuring fairness in deep learning algorithms is an important and active research topic.

PyTorch provides an `nn.Embedding` module, which wraps an *embedding matrix*: this matrix has one row per possible category (e.g., one row for each token in the vocabulary) and one column per embedding dimension. The embedding dimensionality is a hyperparameter you can tune. By default, the embedding matrix is initialized randomly.

To convert a category ID to an embedding, the `nn.Embedding` layer just looks up and returns the corresponding row. That’s all there is to it! For example, let’s initialize an `nn.Embedding` layer with five categories and 3D embeddings, and use it to encode some categories:

```
>>> import torch.nn as nn
>>> torch.manual_seed(42)
>>> embed = nn.Embedding(5, 3) # 5 categories x 3D embeddings
>>> embed(torch.tensor([[3, 2], [0, 2]]))
```

<sup>4</sup> Malvina Nissim et al., “Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor”, arXiv preprint arXiv:1905.09866 (2019).

```

tensor([[[ 0.2674,  0.5349,  0.8094],
        [ 2.2082, -0.6380,  0.4617]],

       [[ 0.3367,  0.1288,  0.2345],
        [ 2.2082, -0.6380,  0.4617]]], grad_fn=<EmbeddingBackward0>)

```

As you can see, category 3 gets encoded as the 3D vector `[0.2674, 0.5349, 0.8094]`, category 2 gets encoded (twice) as the 3D vector `[2.2082, -0.6380, 0.4617]`, and category 0 gets encoded as the 3D vector `[0.3367, 0.1288, 0.2345]` (categories 1 and 4 were not used in this example). Since the layer is not trained yet, these encodings are just random.

Note that an embedding layer is mathematically equivalent to one-hot encoding followed by a linear layer (with no bias parameter). For example, if you create a linear layer with `nn.Linear(5, 3, bias=False)` and pass it the one-hot vector `torch.tensor([[0., 0., 0., 1., 0.]])`, you get a vector equal to row #3 of the linear layer's transposed weight matrix (which acts as an embedding matrix). That's because all rows in the transposed weight matrix get multiplied by zero, except for row #3 which gets multiplied by 1, so the result is just row #3. However, it's much more efficient to use `nn.Embedding(5, 3)` and pass it `torch.tensor([3])`: this looks up row #3 in the embedding matrix without the need for one-hot encoding, and without all the pointless multiplications by zero.

OK, now that you know about embeddings, you are ready to build the Shakespeare model.

## Building and Training the Char-RNN Model

Since our dataset is reasonably large, and modeling language is quite a difficult task, we need more than a simple RNN with a few recurrent neurons. Let's build and train a model with a two-layer `nn.GRU` module (introduced in [Chapter 13](#)), with 128 units per layer, and a bit of dropout. You can try tweaking the number of layers and units later, if needed:

```

class ShakespeareModel(nn.Module):
    def __init__(self, vocab_size, n_layers=2, embed_dim=10, hidden_dim=128,
                 dropout=0.1):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.gru = nn.GRU(embed_dim, hidden_dim, num_layers=n_layers,
                         batch_first=True, dropout=dropout)
        self.output = nn.Linear(hidden_dim, vocab_size)

```

```

def forward(self, x):
    embeddings = self.embed(x)
    outputs, _states = self.gru(embeddings)
    return self.output(outputs).permute(0, 2, 1)

torch.manual_seed(42)
model = ShakespeareModel(len(vocab)).to(device)

```

Let's go over this code:

- We use an `nn.Embedding` layer as the first layer, to encode the character IDs. As we just saw, the `nn.Embedding` layer's number of input dimensions is the number of categories, so in our case it's the number of distinct character IDs. The embedding size is a hyperparameter you can tune—we'll set it to 10 for now. Whereas the inputs of the `nn.Embedding` layer will be integer tensors of shape  $[batch\ size, window\ length]$ , the outputs of the `nn.Embedding` layer will be float tensors of shape  $[batch\ size, window\ length, embedding\ size]$ .
- The `nn.GRU` layer has 10 inputs (i.e., the embedding size), 128 outputs (i.e., the hidden size), two layers, and as usual we must specify `batch_first=True` because otherwise the layer assumes that the batch dimension comes after the time dimension.
- We use an `nn.Linear` layer for the output layer: it must have 39 units because there are 39 distinct characters in the text, and we want to output a logit for each possible character (at each time step).
- In the `forward()` method, we just call these layers one by one. Note that the `nn.GRU` layer's output shape is  $[batch\ size, window\ length, hidden\ size]$ , and the `nn.Linear` layer's output shape is  $[batch\ size, window\ length, vocabulary\ size]$ , but as we saw in [Chapter 13](#), the `nn.CrossEntropyLoss` and `Accuracy` modules that we will use for training both expect the class dimension (i.e., `vocab_size`) to be the second dimension, not the last one. This is why we must permute the last two dimensions of the `nn.Linear` layer's output. Note that the `nn.GRU` layer also returns the final hidden states, but we ignore them.<sup>5</sup>

Now you can now train and evaluate the model as usual, using the `nn.CrossEntropyLoss` and the `Accuracy` metric.

And now let's use our model to predict the next character in a sentence:

```

model.eval() # don't forget to switch the model to evaluation mode!
text = "To be or not to b"
encoded_text = encode_text(text).unsqueeze(dim=0).to(device)

```

---

<sup>5</sup> It's a convention in Python to name unused variables with an underscore prefix.

```
with torch.no_grad():
    Y_logits = model(encoded_text)
    predicted_char_id = Y_logits[0, :, -1].argmax().item()
    predicted_char = id_to_char[predicted_char_id] # correctly predicts "e"
```

We first encode the text, add a batch dimension of size 1, and move the tensor to the GPU. Then we call our model and get logits for each time step. We're only interested in logits for the final time step (hence the `-1`), and we want to know which token ID has the highest logit, so we use `argmax()`. We then use `item()` to extract the token ID from the tensor. Lastly, we convert the token ID to a character, and that's our prediction.

The model correctly predicts “e”, great! Now let’s use this model to pretend we’re Shakespeare!



If you are running this code on Colab with a GPU activated, then training will take a few hours. You can reduce the number of epochs if you don’t want to wait that long, but of course the model’s accuracy will probably be lower. If the Colab session times out, make sure to reconnect quickly, or else the Colab runtime will be destroyed.

## Generating Fake Shakespearean Text

To generate new text using the char-RNN model, we could feed it some text, make the model predict the most likely next letter, add it to the end of the text, then give the extended text to the model to guess the next letter, and so on. This is called *greedy decoding*. But in practice this often leads to the same words being repeated over and over again. Instead, we can sample the next character randomly, using the model’s estimated probability distribution: if the model estimates a probability  $p$  for a given token, then this token will be sampled with probability  $p$ . This process will generate more diverse and interesting text since the most likely token won’t always be sampled. To sample the next token, we can use the `torch.multinomial()` function, which samples random class indices, given a list of class probabilities. For example:

```
>>> torch.manual_seed(42)
>>> probs = torch.tensor([[0.5, 0.4, 0.1]]) # probas = 50%, 40%, and 10%
>>> samples = torch.multinomial(probs, replacement=True, num_samples=8)
>>> samples
tensor([[0, 0, 0, 0, 1, 0, 2, 2]])
```

To have more control over the diversity of the generated text, we can divide the logits by a number called the *temperature*, which we can tweak as we wish. A temperature close to zero favors high-probability characters, while a high temperature gives all characters an equal probability. Lower temperatures are typically preferred when generating fairly rigid and precise text, such as mathematical equations, while higher

temperatures are preferred when generating more diverse and creative text. Let's write a `next_char()` helper function that will use this approach to pick the next character to add to the input text:

```
import torch.nn.functional as F

def next_char(model, text, temperature=1):
    encoded_text = encode_text(text).unsqueeze(dim=0).to(device)
    with torch.no_grad():
        Y_logits = model(encoded_text)
        Y_probas = F.softmax(Y_logits[0, :, -1] / temperature, dim=-1)
        predicted_char_id = torch.multinomial(Y_probas, num_samples=1).item()
    return id_to_char[predicted_char_id]
```

Next, we can write another small helper function that will repeatedly call `next_char()` to get the next character and append it to the given text:

```
def extend_text(model, text, n_chars=80, temperature=1):
    for _ in range(n_chars):
        text += next_char(model, text, temperature)
    return text
```

We are now ready to generate some text! Let's try low, medium, and high temperatures:

```
>>> print(extend_text(model, "To be or not to b", temperature=0.01))
To be or not to be the state
and the contrary of the state and the sea,
the common people of the
>>> print(extend_text(model, "To be or not to b", temperature=0.4))
To be or not to be the better from the cause
that thou think you may be so be gone.

romeo:
that
>>> print(extend_text(model, "To be or not to b", temperature=100))
To be or not to b-c3;m-rkn&:x:uyve:b&hi n;n-h;wt3k
&cixxh:a!kq$c$ 3 ncq$ ;;wq cp:!xq;yh
!3
d!nhi.
```

Notice the repetitions when the temperature is low: “the state and the” appears twice. The intermediate temperature led to more convincing results, although Romeo wasn't very talkative today. But in the last example the temperature was way too high—we fried Shakespeare. To generate more convincing text, a common technique is to sample only from the top  $k$  characters, or only from the smallest set of top characters whose total probability exceeds some threshold: this is called *top-p sampling*, or *nucleus sampling*. Alternatively, you could try using *beam search*, which we will discuss later in this chapter, or using more `nn.GRU` layers and more neurons per layer, training for longer, and adding more regularization if needed.



The model is currently incapable of learning patterns longer than `window_length`, which is just 50 characters. You could try making this window larger, but it would also make training harder, and even LSTM and GRU cells cannot handle very long sequences.<sup>6</sup>

Interestingly, although a char-RNN model is just trained to predict the next character, this seemingly simple task actually requires it to learn some higher-level tasks as well. For example, to find the next character after “Great movie, I really”, it’s helpful to understand that the sentence is positive, so what follows is more likely to be the letter “l” (for “loved”) rather than “h” (for “hated”). In fact, a [2017 paper](#)<sup>7</sup> by Alec Radford and other OpenAI researchers describes how the authors trained a big char-RNN-like model on a large dataset, and found that one of the neurons acted as an excellent sentiment analysis classifier. Although the model was trained without any labels, the *sentiment neuron*—as they called it—reached state-of-the-art performance on sentiment analysis benchmarks (this foreshadowed and motivated unsupervised pretraining in NLP).

Speaking of which, let’s say farewell to Shakespeare and turn to the second part of this chapter: sentiment analysis.

## Sentiment Analysis Using Hugging Face Libraries

One of the most common applications of NLP is text classification—especially sentiment analysis. If image classification on the MNIST dataset is the “Hello, world!” of computer vision, then sentiment analysis on the IMDb reviews dataset is the “Hello, world!” of natural language processing. The IMDb dataset consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing) extracted from the famous [Internet Movie Database](#), along with a simple binary target for each review indicating whether it is negative (0) or positive (1). Just like MNIST, the IMDb reviews dataset is popular for good reasons: it is simple enough to be tackled on a laptop in a reasonable amount of time, but challenging enough to be fun and rewarding.

To download the IMDb dataset, we will use the Hugging Face *Datasets* library, which gives easy access to hundreds of thousands of datasets hosted on the Hugging Face Hub. It is preinstalled on Colab; otherwise it can be installed using `pip install datasets`. We’ll use 80% of the original training set for training, and the remaining 20% for validation, using the `train_test_split()` method to split the set:

---

<sup>6</sup> Another technique to capture longer patterns is to use a stateful RNN. It’s a bit more complex and not used as much, but if you’re interested I’ve included a section in this chapter’s notebook.

<sup>7</sup> Alec Radford et al., “Learning to Generate Reviews and Discovering Sentiment”, arXiv preprint arXiv:1704.01444 (2017).

```

from datasets import load_dataset

imdb_dataset = load_dataset("imdb")
split = imdb_dataset["train"].train_test_split(train_size=0.8, seed=42)
imdb_train_set, imdb_valid_set = split["train"], split["test"]
imdb_test_set = imdb_dataset["test"]

```

Let's inspect a couple of reviews:

```

>>> imdb_train_set[1]["text"]
"The Rookie' was a wonderful movie about the second chances life holds [...]"
>>> imdb_train_set[1]["label"]
1
>>> imdb_train_set[16]["text"]
"Lillian Hellman's play, adapted by Dashiell Hammett with help from Hellman,
becomes a curious project to come out of gritty Warner Bros. [...] It seems to
take forever for this drama to find its focus, [...], it seems a little
patronizing [...] Lukas has several speeches in the third-act which undoubtedly
won him the Academy Award [...] this tasteful, tactful movie [...] It should be
a heady mix, but instead it's rather dry-eyed and inert. ** from ****"
>>> imdb_train_set[16]["label"]
0

```

The first review immediately says that it's a wonderful movie; no need to read any further: it's clearly positive (label = 1). The second review is much harder to classify: it contains a detailed description of the movie, sprinkled with both positive and negative comments. Luckily, the conclusion is quite clearly negative, making the task much easier (label = 0). Still, it's not a trivial task.

A simple char-RNN model would struggle; we need a more powerful tokenization technique. So let's focus on tokenization before we return to sentiment analysis.

## Tokenization Using the Hugging Face Tokenizers Library

In a [2016 paper](#),<sup>8</sup> Rico Sennrich et al. from the University of Edinburgh explored several methods to tokenize and detokenize text at the subword level. This way, even if your model encounters a rare word it has never seen before, it can still reasonably guess what it means. For example, even if the model never saw the word “smartest” during training, if it learned the word “smart” and it also learned that the suffix “est” means “the most”, it can infer the meaning of “smartest”. One of the techniques the authors evaluated is *byte pair encoding* (BPE), introduced by Philip Gage in 1994 (initially for data compression). BPE works by splitting the whole training set into individual characters, then at each iteration it finds the most frequent pair of adjacent tokens and adds it to the vocabulary. It repeats this process until the vocabulary reaches the desired size.

---

<sup>8</sup> Rico Sennrich et al., “Neural Machine Translation of Rare Words with Subword Units”, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* 1 (2016): 1715–1725.

The [Hugging Face Tokenizers library](#) includes highly efficient implementations of several popular tokenization algorithms, including BPE. It is preinstalled on Colab (or you can install it with `pip install tokenizers`). Here's how to train a BPE model on the IMDb dataset:

```
import tokenizers

bpe_model = tokenizers.models.BPE(unk_token=<unk>)
bpe_tokenizer = tokenizers.Tokenizer(bpe_model)
bpe_tokenizer.pre_tokenizer = tokenizers.pre_tokenizers.Whitespace()
special_tokens = [<pad>, <unk>]
bpe_trainer = tokenizers.trainers.BpeTrainer(vocab_size=1000,
                                              special_tokens=special_tokens)
train_reviews = [review["text"].lower() for review in imdb_train_set]
bpe_tokenizer.train_from_iterator(train_reviews, bpe_trainer)
```

Let's walk through this code:

- We import the Tokenizers library, and we create a BPE model, specifying an unknown token "<unk>" which will be used later if we try to tokenize some text containing tokens that the model never saw during training: the unknown tokens will be replaced with the "<unk>" token.
- We then create a Tokenizer based on the BPE model.
- The Tokenizers library lets you specify optional preprocessing and postprocessing steps, and it also provides common preprocessors and postprocessors. In this example, we use the `Whitespace` preprocessor which splits the text at spaces (and drops the spaces), and also separates groups of letters and groups of nonletters. For example "Hello, world!!! will be split into ["Hello", ",", "world", "!!!"]. The BPE algorithm will then run on these individual chunks, which dramatically speeds up training and improves token quality (at least when the text is in English) by providing reasonable word boundaries.
- We then define a list of special tokens: a padding token "<pad>" that will come in handy when we create batches of texts of different lengths, and the unknown token we have already discussed.
- We create a `BpeTrainer`, specifying the maximum vocabulary size and the list of special tokens. The trainer will add the special tokens at the beginning of the vocabulary, so "<pad>" will be token 0, and "<unk>" will be token 1.
- Next we create a list of all the text in the IMBD training set.
- Lastly, we train the tokenizer on this list, using the `BpeTrainer`. A few seconds later, the BPE tokenizer is ready to be used!

Now let's use our BPE tokenizer to tokenize some text:

```
>>> some_review = "what an awesome movie! 😊\n\n>>> bpe_encoding = bpe_tokenizer.encode(some_review)\n>>> bpe_encoding\nEncoding(num_tokens=8, attributes=[ids, type_ids, tokens, offsets,\n    attention_mask, special_tokens_mask, overflowing])
```

The `encode()` method returns an `Encoding` object that contains eight tokens. Let's look at these tokens and their IDs:

```
>>> bpe_encoding.tokens\n['what', 'an', 'aw', 'es', 'ome', 'movie', '!', '<unk>']\n>>> bpe_token_ids = bpe_encoding.ids\n>>> bpe_token_ids\n[303, 139, 373, 149, 240, 211, 4, 1]
```

Notice that frequent words like “what” and “movie” have been identified by the BPE model and are represented by a single token, while less frequent words like “awesome” are split into multiple tokens. Also note that the smiley was not part of the training data, so it gets replaced with the unknown token “`<unk>`”.

The tokenizer provides a `get_vocab()` method which returns a dictionary mapping each token to its ID. You can also use the `token_to_id()` method to map a single token, or conversely use the `id_to_token()` method to go from ID to token. However, you will more often use the `decode()` method to convert a list of token IDs into a string:

```
>>> bpe_tokenizer.decode(bpe_token_ids)\n'what an aw es ome movie !'
```

The tokenizer keeps track of each token's start and end offset in the original string, which can come in handy, especially for debugging:

```
>>> bpe_encoding.offsets\n[(0, 4), (5, 7), (8, 10), (10, 12), (12, 15), (16, 21), (21, 22), (23, 24)]
```

It's also possible to encode a whole batch of strings at once. For example, let's encode the first three reviews of the training set:

```
>>> bpe_tokenizer.encode_batch(train_reviews[:3])\n[Encoding(num_tokens=281, attributes=[ids, type_ids, tokens, [...]]),\n Encoding(num_tokens=114, attributes=[ids, type_ids, tokens, [...]]),\n Encoding(num_tokens=285, attributes=[ids, type_ids, tokens, [...]]),
```

If we want to create a single integer tensor containing the token IDs of all three reviews, we must first ensure that they all have the same number of tokens, which is not the case right now. For this, we can ask the tokenizer to pad the shorter reviews with the padding token ID until they are as long as the longest review in the batch.

We can also ask the tokenizer to truncate any sequence longer than some maximum length, since RNNs don't handle very long sequences very well anyway:

```
bpe_tokenizer.enable_padding(pad_id=0, pad_token="<pad>")
bpe_tokenizer.enable_truncation(max_length=500)
```

Now let's encode the batch again. This time all sequences will have the same number of tokens, so we can create a tensor containing all the token IDs:

```
>>> bpe_encodings = bpe_tokenizer.encode_batch(train_reviews[:3])
>>> bpe_batch_ids = torch.tensor([encoding.ids for encoding in bpe_encodings])
>>> bpe_batch_ids
tensor([[159, 402, 176, 246, 61, [...], 215, 156, 586, 0, 0, 0, 0, 0],
        [10, 138, 198, 289, 175, [...], 0, 0, 0, 0, 0, 0, 0, 0],
        [289, 15, 209, 398, 177, [...], 50, 29, 22, 17, 24, 18, 24]])
```

Notice how the first and second review were padded with 0s, which is our padding token ID. Each `Encoding` object also includes an `attention_mask` attribute containing a 1 for each nonpadding token, and a 0 for each padding token. This can be used in your models to easily ignore the padded time steps: just multiply a tensor with the attention mask. In some cases you will prefer to have the list of sequence lengths (ignoring padding). Here's how to get both the attention mask tensor and the sequence lengths:

```
>>> attention_mask = torch.tensor([encoding.attention_mask
...                                     for encoding in bpe_encodings])
...
>>> attention_mask
tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, [...], 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, [...], 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, [...], 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
>>> lengths = attention_mask.sum(dim=-1)
>>> lengths
tensor([281, 114, 285])
```

You may have noted that spaces were not handled very well by our tokenizer. In particular, the word "awesome" came back as "aw es ome", and "movie!" came back as "movie !". This is because the `Whitespace` pre-tokenizer dropped all spaces, therefore the BPE tokenizer doesn't know where spaces should go and it just adds spaces between all tokens. To fix this, we can replace the `Whitespace` pre-tokenizer with the `ByteLevel` pre-tokenizer: it replaces all spaces with a special character `\u202f` so the BPE model doesn't lose track of them. For example, if you use this pre-tokenizer and you encode and decode the text "what an awesome movie! 😊", you will get: "Gwhat G\u202f\u00c3\u00e1n \u202f\u00c3\u00e1w es ome \u202f\u00c3\u00e1movie !". After removing the spaces, then replacing every `\u202f` with a space, you get " what an awesome movie!". This is almost perfect, except for the extra space at the start—which is easily removed—and the lost emoji, which was replaced with an unknown token because it's not in the vocabulary, and dropped by the `decode()` method.

As its name suggests, the `ByteLevel` pre-tokenizer allows the BPE model to work at the byte level, rather than the character level: unsurprisingly, this is called Byte-level BPE (BBPE). For example, the 😊 emoji will be converted to four bytes, using Unicode’s UTF-8 encoding. This means that BBPE will never output an unknown token if its vocabulary contains all 256 possible bytes, since any text can be broken down into its individual bytes whenever longer tokens are not found in the vocabulary. This makes BBPE well suited when the corpus contains rare characters such as emojis.

Another important variant of BPE is `WordPiece`,<sup>9</sup> proposed by Google in 2016. This tokenization algorithm is very similar to BPE, but instead of adding the most frequent adjacent pair of tokens to the vocabulary at each iteration, it adds the pair with the highest score. This score is computed using [Equation 14-1](#): the `frequency(AB)` term is just like in BPE—it boosts pairs that are frequent in the corpus. However, the denominator reduces the score of a pair when the individual tokens are themselves frequent. This normalization tends to favor more useful and meaningful tokens than BPE, and the algorithm often produces shorter encoded sequences than BPE or BBPE.

*Equation 14-1. WordPiece score for a pair AB composed of tokens A and B*

$$\text{score}(AB) = \frac{\text{frequency}(AB)}{\text{freq}(A) \cdot \text{freq}(B)} \cdot \text{len}(\text{vocab})$$

To train a WordPiece tokenizer using the `Tokenizers` library, you can use the same code as for BPE, but replace the `BPE` model with `WordPiece`, and the `BpeTrainer` with `WordPieceTrainer`. If you encode and decode the same review as earlier, you will get “what an aw esome movie !”. Notice that WordPiece adds a prefix to tokens that are inside a word, which makes it easy to reconstruct the original string: just remove “ #” (as well as spaces before punctuations). Note that the smiley emoji once again disappeared because it was not in the vocabulary.

One last popular tokenization algorithm we will discuss is Unigram LM (Language Model), introduced in a [2018 paper](#)<sup>10</sup> by Taku Kudo at Google. This technique is a bit different than the previous ones: it starts out with a very large vocabulary containing every frequent word, subword, and character in the training corpus, then it gradually removes the least useful tokens until it reaches the desired vocabulary size. To determine how useful a token is, this method makes one big simplifying assumption: it assumes that the corpus was sampled randomly from the vocabulary,

---

<sup>9</sup> Yonghui Wu et al., “Google’s Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation”, arXiv preprint arXiv:1609.08144 (2016).

<sup>10</sup> Taku Kudo, “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates”, arXiv preprint arXiv:1804.10959 (2018).

one token at a time (hence the name Unigram LM), and that every token was sampled independently from the others. Therefore, this tokenizer model assumes that the probability of sampling the pair AB is equal to the probability of sampling A times the probability of sampling B. Given this assumption, it can estimate the probability of sampling the whole training corpus. At each iteration, the training algorithm attempts to remove tokens without reducing this overall probability too much.

For example, suppose that the vocabulary contains the tokens “them”, “the”, and “m”, respectively, with 1%, 5%, and 2% probability. This means that the word “them” has a 1% chance of being sampled as the single token “them”, or a  $5\% \times 2\% = 0.1\%$  chance of being sampled as the pair “the” + “m”. Overall, the word “them” has a  $1\% + 0.1\% = 1.1\%$  chance of being sampled. If we remove the token “them” from the vocabulary, then the probability of sampling the word “them” drops down to 0.1%. If instead we remove either “m” or “the”, then the probability only drops down to 1% since we can still sample the single token “them”. So if the training corpus only contained the word “them”, then the algorithm would prefer to drop either “the” or “m”. Of course, in reality the corpus contains many other words that contain these two tokens, so the algorithm will likely find other less useful tokens to drop.



Unigram LM is great for languages that don't use spaces to separate words, like English does. For example, Chinese text does not use spaces between words, Vietnamese uses spaces even within words, and German often attaches multiple words together, without spaces.

The same paper also proposed a novel regularization technique called *subword regularization*, which improves generalization and robustness by introducing some randomness in tokenization while training the NLP model (not the tokenizer model). For example, assuming the vocabulary contains the tokens “them”, “the”, and “m”, and you choose to use subword regularization, then the word “them” will sometimes be tokenized as “the” + “m”, and sometimes as “them”. This technique works best with *morphologically rich languages*, meaning languages where words carry a lot of grammatical information through affixes, inflections, and internal modifications (such as Arabic, Finnish, German, Hungarian, Polish, or Turkish), as opposed to languages that rely on word order or additional helper words (such as English, Chinese, Thai, or Vietnamese).

Unfortunately, the Tokenizers library does not natively support subword regularization, so you either have to implement it yourself, or you can use Google's *SentencePiece* library (`pip install sentencepiece`) which provides an open source

implementation. This project is described in a [2018 paper](#)<sup>11</sup> by Taku Kudo and John Richardson.

**Table 14-1** summarizes the three main tokenizers used today.

*Table 14-1. Overview of the three main tokenizers*

Feature	BBPE	WordPiece	Unigram LM
How	Merge most frequent pairs	Merge pairs that maximize data likelihood	Remove least likely tokens
Pros	Fast, simple, great for multilingual	Good balance of efficiency and token quality	Most meaningful, shortest sequences
Cons	Can produce awkward splits	Less robust than BBPE for multilingual	Slower to train and tokenize
Used By	GPT, Llama, RoBERTa, BLOOM	BERT, DistilBERT, ELECTRA	T5, ALBERT, mBART

Training your own tokenizer is useful in many situations; for example, if you are dealing with domain-specific text, such as medical, legal, or engineering documents full of jargon, or if the text is written in a low-resource language or dialect, or if it's code written in a new programming language, and so on. However, in most cases you will want to simply reuse a pretrained tokenizer.

## Reusing Pretrained Tokenizers

To download a pretrained tokenizer, we will use the [Hugging Face Transformers library](#). This library provides many popular models for NLP, computer vision, audio processing, and more. Pretrained weights are available for almost all of these models, and the library can automatically download them from the Hugging Face Hub. The models were originally all based on the *Transformer architecture* (which we will discuss in detail in [Chapter 15](#)), hence the name of the library, but other kinds of models are now available as well, such as CNNs. Lastly, each model comes with all the tools it needs, including tokenizers for NLP models: in a single line of code, you can have a fully functional, high-performance model for a given task, as we will see later in this chapter.

For now, let's just grab the pretrained tokenizer from some NLP model. For example, the following code downloads the pretrained BBPE tokenizer used by the GPT-2 model (a text generation model), and it uses this tokenizer to encode the first 3 IMDb reviews, truncating the encoded sequences if they exceed 500 tokens:

```
import transformers

gpt2_tokenizer = transformers.AutoTokenizer.from_pretrained("gpt2")
```

---

<sup>11</sup> Taku Kudo and John Richardson, "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing", arXiv preprint arXiv:1808.06226 (2018).

```
gpt2_encoding = gpt2_tokenizer(train_reviews[:3], truncation=True,  
                                max_length=500)
```

Notice that we use the tokenizer object like a function. The result is a dictionary-like object of type `BatchEncoding`. You can get the token IDs using the "input\_ids" key. It returns a Python list of lists of token IDs. For example, let's look at the first 10 token IDs of the first encoded review, and use the tokenizer to decode them, using its `decode()` method:

```
>>> gpt2_token_ids = gpt2_encoding["input_ids"][:10]
>>> gpt2_token_ids
[14247, 35030, 1690, 423, 257, 1688, 8046, 13, 484, 1690]
>>> gpt2_tokenizer.decode(gpt2_token_ids)
'stage adaptations often have a major fault. they often'
```

If you would prefer to use a pretrained WordPiece tokenizer, you can reuse the tokenizer of any LLM that was pretrained using WordPiece, such as BERT (another popular NLP model, which stands for Bidirectional Encoder Representations from Transformers). This tokenizer has a padding token (unlike the previous tokenizer, since GPT-2 didn't need it), so we can specify `padding=True` when encoding a batch of reviews: as usual, the shortest texts will be padded to the length of the longest one using the padding token. This allows us to also specify `return_tensors="pt"` to get a PyTorch tensor instead of a Python list of lists of token IDs: very convenient! So let's encode the first three IMDb reviews:

```
bert_tokenizer = transformers.AutoTokenizer.from_pretrained("bert-base-uncased")
bert_encoding = bert_tokenizer(train_reviews[:3], padding=True,
                               truncation=True, max_length=500,
                               return_tensors="pt")
```



The name "bert-base-uncased" refers to a *model checkpoint*: this particular checkpoint is a case-insensitive BERT model, pretrained on English text. Other checkpoints are available, such as "bert-large-cased" if you want a larger and case-sensitive BERT model, or "bert-base-multilingual-uncased" if you want an uncased model pretrained on over 100 languages. For now we are just using the model's tokenizer.

The resulting token IDs and attention masks are nicely padded tensors:

Notice that each token ID sequence starts with token 101 ([CLS]), and ends with token 102 ([SEP]) (ignoring padding tokens). These tokens are needed by the BERT model (as we will see in [Chapter 15](#)), but unless your model needs them too, you can drop them by setting `add_special_tokens=False` when calling the tokenizer.

What about a pretrained Unigram LM tokenizer? Well, many models were trained using Unigram LM, such as ALBERT, T5, or XML-R models, just to name a few. For example:

```
albert_tokenizer = transformers.AutoTokenizer.from_pretrained("albert-base-v2")
albert_encoding = albert_tokenizer(train_reviews[:3], padding=True, [...])
```

The Transformers library also provides an object that can wrap your own tokenizer (from the Tokenizers library) and give it the same API as the pretrained tokenizers (from the Transformers library). For example, let's wrap the BPE tokenizer we trained earlier:

```
hf_tokenizer = transformers.PreTrainedTokenizerFast(
    tokenizer_object=bpe_tokenizer)
hf_encodings = hf_tokenizer(train_reviews[:3], padding=True, [...])
```

With that, we have all the tokenization tools we need, so let's go back to sentiment analysis.

## Building and Training a Sentiment Analysis Model

Our sentiment analysis model must be trained using batches of tokenized reviews. However, the datasets we created did not take care of tokenization. One option would be to update them (e.g., using the `map()` method), but it's just as simple to handle tokenization in the data loaders. To do this, we can pass a function to the `DataLoader` constructor using its `collate_fn` argument: the data loader will call this function for every batch, passing it a list of dataset samples. Our function will take this batch, tokenize the reviews, truncate and pad them if needed, and return a `BatchEncoding` object containing PyTorch tensors for the token IDs and attention masks, along with another tensor containing the labels. For tokenization, we will simply use the pretrained WordPiece tokenizer we just loaded:

```
def collate_fn(batch, tokenizer=bert_tokenizer):
    reviews = [review["text"] for review in batch]
    labels = [[review["label"]]] for review in batch]
    encodings = tokenizer(reviews, padding=True, truncation=True,
                          max_length=200, return_tensors="pt")
    labels = torch.tensor(labels, dtype=torch.float32)
    return encodings, labels

batch_size = 256
imdb_train_loader = DataLoader(imdb_train_set, batch_size=batch_size,
                               collate_fn=collate_fn, shuffle=True)
imdb_valid_loader = DataLoader(imdb_valid_set, batch_size=batch_size,
```

```
        collate_fn=collate_fn)
imdb_test_loader = DataLoader(imdb_test_set, batch_size=batch_size,
                             collate_fn=collate_fn)
```

Now we're ready to create our sentiment analysis model:

```
class SentimentAnalysisModel(nn.Module):
    def __init__(self, vocab_size, n_layers=2, embed_dim=128, hidden_dim=64,
                 pad_id=0, dropout=0.2):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim,
                                 padding_idx=pad_id)
        self.gru = nn.GRU(embed_dim, hidden_dim, num_layers=n_layers,
                          batch_first=True, dropout=dropout)
        self.output = nn.Linear(hidden_dim, 1)

    def forward(self, encodings):
        embeddings = self.embed(encodings["input_ids"])
        _outputs, hidden_states = self.gru(embeddings)
        return self.output(hidden_states[-1])
```

As you can see, this model is very similar to our Shakespeare model, but with a few important differences:

- When creating the `nn.Embedding` layer, we set its `padding_idx` argument to our padding ID. This ensures that the padding ID gets embedded as a nontrainable zero vector to reduce the impact of padding tokens on the loss.
- Since this is a sequence-to-vector model, not a sequence-to-sequence model, we only need the last output of the top GRU layer to make our final prediction (through the output `nn.Linear` layer). We could have used `outputs[:, -1]` instead of `hidden_states[-1]`, as they are equal.
- The output `nn.Linear` layer has a single output dimension because it's a binary classification model. The final output will be a 2D tensor with a single column containing one logit per review, positive for positive reviews, and negative for negative reviews.
- The `forward()` method takes a `BatchEncoding` object as input, containing the token IDs (possibly padded and truncated).

We can then train this model using the `nn.BCEWithLogitsLoss` since this is a binary classification task. It reaches close to 85% accuracy on the validation set, which is reasonably good, although the best models reach human level, slightly above 90% accuracy. It's probably not possible to go much higher than that; because many reviews are ambiguous, classifying them feels like flipping a coin.

One problem with our model is the fact that we are not fully ignoring the padding tokens. Indeed, if a review ends with many padding tokens, the `nn.GRU` module will have to process them, and by the time it gets through all of them, it might have

forgotten what the review was all about. To avoid this, we can use a *packed sequence* instead of a regular tensor. A packed sequence is a special data structure designed to efficiently represent a batch of sequences of variable lengths.<sup>12</sup> You can use the `pack_padded_sequence()` function to convert a tensor containing padded sequences to a packed sequence object, and conversely you can use the `pad_packed_sequence()` function whenever you want to convert a packed sequence object to a padded tensor:

```
>>> from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
>>> sequences = torch.tensor([[1, 2, 0, 0], [5, 6, 7, 8]])
>>> packed = pack_padded_sequence(sequences, lengths=(2, 4),
...                                enforce_sorted=False, batch_first=True)
...
>>> packed
PackedSequence(data=tensor([5, 1, 6, 2, 7, 8]), [...])
>>> padded, lengths = pad_packed_sequence(packed, batch_first=True)
>>> padded, lengths
(tensor([[1, 2, 0, 0],
       [5, 6, 7, 8]]),
 tensor([2, 4]))
```

By default, the `pack_padded_sequence()` function assumes that the sequences in the batch are ordered from the longest to the shortest. If this is not the case, you must set `enforce_sorted=False`. Moreover, the function also assumes that the time dimension comes before the batch dimension. If the batch dimension is first, you must set `batch_first=True`.

PyTorch's recurrent layers support packed sequences: they efficiently process the sequences, stopping at the end of each sequence. So let's update our sentiment analysis model to use packed sequences. In the `forward()` method, just replace the `self.gru(embeddings)` line with the following code:

```
lengths = encodings["attention_mask"].sum(dim=1)
packed = pack_padded_sequence(embeddings, lengths=lengths.cpu(),
                             batch_first=True, enforce_sorted=False)
_outputs, hidden_states = self.gru(packed)
```

This code starts by computing the length of each sequence in the batch, just like we did earlier, then it packs the embeddings tensor and passes the packed sequence to the `nn.GRU` module. With that, the model will properly handle sequences without being bothered by any padding tokens. You don't actually need to set `padding_idx` anymore when creating the `nn.Embedding` layer, but it doesn't hurt, and it makes debugging a bit easier, so I prefer to keep it.

---

<sup>12</sup> Nested tensors serve a similar purpose and are more convenient to use, but they are still in prototype stage at the time of writing. See <https://pytorch.org/docs/stable/nested.html> for more details.

Another way to improve our model is to let it look at the review in both directions: left to right, and right to left. Let's see how this works.



If you pass a packed sequence to an `nn.GRU` module, its outputs will also be a packed sequence, and you will need to convert it back to a padded tensor before you can pass it to the next layers. Luckily, we don't need these outputs for our sentiment analysis model, only the hidden states.

## Bidirectional RNNs

At each time step, a regular recurrent layer only looks at past and present inputs before generating its output. In other words, it is *causal*, meaning it cannot look into the future. This type of RNN makes sense when forecasting time series, or in the decoder of a sequence-to-sequence (seq2seq) model. But for tasks like text classification, or in the encoder of a seq2seq model, it is often preferable to look ahead at the next words before encoding a given word.

For example, consider the phrases “the right arm”, “the right person”, and “the right to speak”: to properly encode the word “right”, you need to look ahead. One solution is to run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left, then combine their outputs at each time step, typically by concatenating them. This is what a *bidirectional recurrent layer* does (see [Figure 14-4](#)).

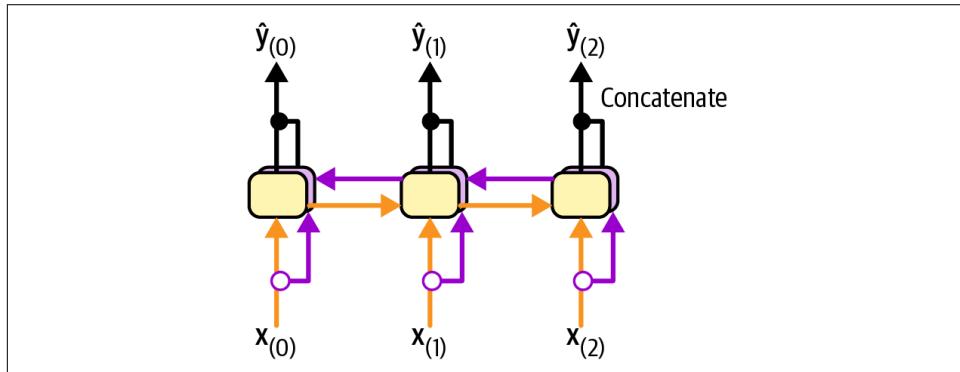


Figure 14-4. A bidirectional recurrent layer

To make our sentiment analysis model bidirectional, we can just set `bidirectional=True` when creating the `nn.GRU` layer (this also works with the `nn.RNN` and `nn.LSTM` modules).

However, once we do that, we must adjust our model a bit. In particular, we must double the input dimension of the output `nn.Linear` layer, since the hidden states will double in size:

```
self.output = nn.Linear(2 * hidden_dim, 1)
```

We must also concatenate the forward and backward hidden states of the GRU's top layer before passing the result to the output layer. For this, we can replace the last line of the `forward()` method (i.e., `return self.output(hidden_states[-1])`) with the following code:

```
n_dims = self.output.in_features
top_states = hidden_states[-2:].permute(1, 0, 2).reshape(-1, n_dims)
return self.output(top_states)
```

Let's see how the middle line works:

- Until now, the shape of the hidden states returned by the `nn.GRU` module was *[number of layers, batch size, hidden size]*, so [2, 256, 64] in our case. But when we set `bidirectional=True`, we doubled the first dimension size, so we now have a shape of [4, 256, 64]: the tensor contains the hidden states for layer 1 forward, layer 1 backward, layer 2 forward, and layer 2 backward. Since we only want the top layer's hidden states, both forward and backward, we must get `hidden_states[-2:]`.
- We also need to concatenate the forward and backward states. One way to do this is to permute the first two dimensions of the top hidden states using `permute(1, 0, 2)` to get the shape [256, 2, 64], then reshape the result using `reshape(-1, n_dims)` (where `n_dims` equals 128) to get the desired shape: [256, 2 \* 64].



In this model we only use the last hidden states, ignoring the outputs at each time step. If you ever want to use the outputs of a bidirectional module, be aware that its last dimension's size will be doubled.

You can try training this model, but you will not see any improvement in this case, because the first model actually overfit the training set, and this new version makes it even worse: it reaches over 99% accuracy on the training set, but just 84% on the validation set. To fix this, you could try to regularize the model a bit more, reduce the size of the model, or increase the size of the training set.

But let's instead try something different: using pretrained embeddings.

## Reusing Pretrained Embeddings and Language Models

Our model was able to learn useful embeddings for thousands of tokens, based on just 25,000 movie reviews: that's quite impressive! Imagine how good the embeddings would be if we had billions of reviews to train on. The good news is that we can reuse word embeddings even when they were trained on some other (very) large text corpus, even if it was not composed of movie reviews, and even if they were not trained for sentiment analysis. After all, the word "amazing" generally has the same meaning whether you use it to talk about movies or anything else.

Since we used pretrained tokens for the BERT model, we might as well try using its embedding layer. First, we need to download the pretrained model using the `AutoModel.from_pretrained()` function from the Transformers library, then we can directly access its embeddings layer:

```
>>> bert_model = transformers.AutoModel.from_pretrained("bert-base-uncased")
>>> bert_model.embeddings.word_embeddings
Embedding(30522, 768, padding_idx=0)
```

As you can see, this BERT model is implemented using PyTorch, and it contains a regular `nn.Embedding` layer. We could just replace our model's `nn.Embedding` layer with this one (and retrain our model), but we can keep models cleanly separated by initializing our own `nn.Embedding` layer with a copy of the pretrained embedding matrix. This can be done using the `Embedding.from_pretrained()` function:

```
class SentimentAnalysisModelPreEmbeds(nn.Module):
    def __init__(self, pretrained_embeddings, n_layers=2, hidden_dim=64,
                 dropout=0.2):
        super().__init__()
        weights = pretrained_embeddings.weight.data
        self.embed = nn.Embedding.from_pretrained(weights, freeze=True)
        embed_dim = weights.shape[-1]
    [...] # the rest of the model is exactly like earlier

imdb_model_bert_embeds = SentimentAnalysisModelPreEmbeds(
    bert_model.embeddings.word_embeddings).to(device)
```

Note that we set `freeze=True` when creating the `nn.Embedding` layer: this makes it nontrainable and ensures that the pretrained embeddings won't be damaged by large gradients at the beginning of training. You can train the model for a few epochs like this, then make the embedding layer trainable and continue training, letting the model fine-tune the embeddings for our task.

Pretrained word embeddings have been popular for quite a while, starting with Google's [Word2vec embeddings](#) (2013), Stanford's [GloVe embeddings](#) (2014), Facebook's [FastText embeddings](#) (2016), and more. However, this approach has its limits. In particular, a word has a single representation, no matter the context. For example, the word "right" is encoded the same way in "left and right" and "right and wrong", even

though it means two very different things. To address this limitation, a [2018 paper<sup>13</sup>](#) by Matthew Peters introduced *Embeddings from Language Models* (ELMo): these are contextualized word embeddings learned from the internal states of a deep bidirectional RNN language model. In other words, instead of just using pretrained word embeddings in your model, you can reuse several layers of a pretrained language model.

At roughly the same time, the [Universal Language Model Fine-Tuning \(ULMFiT\) paper<sup>14</sup>](#) by Jeremy Howard and Sebastian Ruder demonstrated the effectiveness of unsupervised pretraining for NLP tasks. The authors trained an LSTM language model on a huge text corpus using self-supervised learning (i.e., generating the labels automatically from the data), then they fine-tuned it on various tasks. Their model outperformed the state of the art on six text classification tasks by a large margin (reducing the error rate by 18%–24% in most cases). Moreover, the authors showed that a pretrained model fine-tuned on just 100 labeled examples could achieve the same performance as one trained from scratch on 10,000 examples. Before the ULMFiT paper, using pretrained models was only the norm in computer vision; in the context of NLP, pretraining was limited to word embeddings. This paper marked the beginning of a new era in NLP: today, reusing pretrained language models is the norm.

For example, why not reuse the entire pretrained BERT model for our sentiment analysis model? To use the BERT model, the Transformers library lets us call it like a function, passing it the tokenized reviews:

```
>>> bert_encoding = bert_tokenizer(train_reviews[:3], padding=True,
...                                     max_length=200, truncation=True,
...                                     return_tensors="pt")
...
>>> bert_output = bert_model(**bert_encoding)
>>> bert_output.last_hidden_state.shape
torch.Size([3, 200, 768])
```

BERT’s output includes an attribute named `last_hidden_state`, which contains contextualized embeddings for each token. The word “last” in this case refers to the last layer, not the last time step (BERT is a transformer, not an RNN). This `last_hidden_state` tensor has a shape of [*batch size*, *max sequence length*, *hidden size*]. Let’s use these contextualized embeddings in a sentiment analysis model:

---

<sup>13</sup> Matthew Peters et al., “Deep Contextualized Word Representations”, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2018): 2227–2237.

<sup>14</sup> Jeremy Howard and Sebastian Ruder, “Universal Language Model Fine-Tuning for Text Classification”, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics* 1 (2018): 328–339.

```

class SentimentAnalysisModelBert(nn.Module):
    def __init__(self, n_layers=2, hidden_dim=64, dropout=0.2):
        super().__init__()
        self.bert = transformers.AutoModel.from_pretrained("bert-base-uncased")
        embed_dim = self.bert.config.hidden_size
        self.gru = nn.GRU(embed_dim, hidden_dim, [...])
        self.output = nn.Linear(hidden_dim, 1)

    def forward(self, encodings):
        contextualized_embeddings = self.bert(**encodings).last_hidden_state
        lengths = encodings["attention_mask"].sum(dim=1)
        packed = pack_padded_sequence(contextualized_embeddings, [...])
        _outputs, hidden_states = self.gru(packed)
        return self.output(hidden_states[-1])

```

Note that we don't need to make the `nn.GRU` module bidirectional since the contextualized embeddings already looked ahead.

If you freeze the BERT model (e.g., using `model.bert.requires_grad_(False)`) and train the rest of the model, you will notice a significant performance boost, reaching over 88% accuracy. Wonderful!

Another option is to use only the contextualized embedding for the very first token, which is the *class token* [CLS]. Indeed, during pretraining, the BERT model had to perform a text classification task based solely on this token's contextualized embedding (we will discuss BERT pretraining in more detail in [Chapter 15](#)). As a result, it learned to summarize the most important features of the text into this embedding. This simplifies our model quite a bit, since we can get rid of the `nn.GRU` module altogether, and the `forward()` method becomes much shorter:

```

def forward(self, encodings):
    bert_output = self.bert(**encodings)
    return self.output(bert_output.last_hidden_state[:, 0])

```

In fact, the BERT model contains an extra hidden layer on top of the class embedding, composed of an `nn.Linear` module and an `nn.Tanh` module. This hidden layer is called the *pooler*. To use it, just replace `bert_output.last_hidden_state[:, 0]` with `bert_output.pooler_output`. You may also want to unfreeze the pooler after a few epochs to fine-tune it for the IMDb task.

So we started by reusing only the pretrained tokenizer, then we reused the pretrained embeddings, then most of the pretrained BERT model, and finally the full model, adding only an `nn.Linear` layer on top of the pooler. We can actually go one step further and just use an off-the-shelf class for sentence classification.

## Task-Specific Classes

To tackle our binary classification task using BERT, we can use the `BertForSequenceClassification` class provided by the Transformers library. It's just a BERT model

plus a classification head on top. All you need to do to create this model is specify the pretrained BERT checkpoint you want to use, the number of output units for your classification task, and optionally the data type (we'll use 16-bit floats to fit on small GPUs):

```
from transformers import BertForSequenceClassification

torch.manual_seed(42)
bert_for_binary_clf = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", num_labels=2, dtype=torch.float16).to(device)
```



The Transformers library contains many task-specific classes based on various pretrained models, such as `BertForQuestionAnswering` or `RobertaForSequenceClassification` (see [Chapter 15](#)). You can also use `AutoModelForSequenceClassification` to let the library pick the right class for you, based on the requested model checkpoint (e.g., if you ask for `"bert-base-uncased"`, you will get an instance of `BertForSequenceClassification`). Similar `AutoModelFor[...]` classes are available for other tasks.

Until now we have always used a single output for binary classification, so why did we set `num_labels=2`? Well, for simplicity Hugging Face prefers to treat binary classification exactly like multiclass classification, so this model will output two logits instead of one, and it must be trained using the `nn.CrossEntropyLoss` instead of `nn.BCELoss` or `nn.BCEWithLogitsLoss`. If you want to convert the logits to estimated probabilities, you must use `torch.softmax()` rather than `torch.sigmoid()`.

Let's call this model on a very positive review:

```
>>> encoding = bert_tokenizer(["This was a great movie!"])
>>> with torch.no_grad():
...     output = bert_for_binary_clf(
...         input_ids=torch.tensor(encoding["input_ids"], device=device),
...         attention_mask=torch.tensor(encoding["attention_mask"], device=device))
...
>>> output.logits
tensor([[-0.0120,  0.6304]], device='cuda:0', dtype=torch.float16)
>>> torch.softmax(output.logits, dim=-1)
tensor([[0.3447, 0.6553]], device='cuda:0', dtype=torch.float16)
```

We first tokenize the review, then we call the model, it returns a `ModelOutput` object containing the logits, and we convert these logits to estimated probabilities using the `torch.softmax()` function. Ouch! The model classified this review as negative with 65.53% confidence! Indeed, the BERT model inside `BertForSequenceClassification` is pretrained, but not the classification head, so we're going to get terrible performance until we actually train this model on the IMDb dataset.

If you pass labels when calling this model (or any other model from the Transformers library), then it also computes the loss and returns it in the `ModelOutput` object. For example:

```
>>> with torch.no_grad():
...     output = bert_for_binary_clf(
...         input_ids=torch.tensor(encoding["input_ids"], device=device),
...         attention_mask=torch.tensor(encoding["attention_mask"], device=device),
...         labels=torch.tensor([1], device=device))
...
>>> output.loss
tensor(0.4226, device='cuda:0', dtype=torch.float16)
```

Since `num_labels` is greater than 1, the model computes the `nn.CrossEntropyLoss` (which is implemented as `nn.LogSoftmax` followed by `nn.NLLLoss`—that's why we see `grad_fn=<NllLossBackward0>`). If we had used `num_labels=1`, then the model would have used the `nn.MSELoss` instead; this can be useful for regression tasks.

We could now train this model using our own training code, as we did so far, but the Transformers library provides a convenient *Trainer API*, so let's check it out.

## The Trainer API

The Trainer API lets you fine-tune a model on your own dataset with very little boilerplate code. It can save model checkpoints during training, apply early stopping, distribute the computations across GPUs, log metrics, take care of padding, batching, shuffling, and more. Let's use the Trainer API to train our IMDb model.

The Trainer API works directly with dataset objects, not data loaders, but it expects the datasets to contain tokenized text, not strings, so we must take care of tokenization. We can do this quite simply using the dataset's `map()` method (this method is implemented by the Datasets library; it's not available on pure PyTorch datasets):

```
def tokenize_batch(batch):
    return bert_tokenizer(batch["text"], truncation=True, max_length=200)

tok_imdb_train_set = imdb_train_set.map(tokenize_batch, batched=True)
tok_imdb_valid_set = imdb_valid_set.map(tokenize_batch, batched=True)
tok_imdb_test_set = imdb_test_set.map(tokenize_batch, batched=True)
```

Since we set `batched=True`, the `map()` method passes batches of reviews to the `tokenize_batch()` method: this is optional, but it significantly speeds up this preprocessing step. The `tokenize_batch()` method tokenizes the given batch of reviews, and the resulting fields are added to each instance by the `map()` method. This includes fields such as `token_ids` and `attention_mask`, which the model expects.

To evaluate our model, we can write a simple function that takes an object with two attributes: `label_ids` and `predictions`:

```
def compute_accuracy(pred):
    return {"accuracy": (pred.label_ids == pred.predictions.argmax(-1)).mean()}
```



Alternatively, you can use metrics provided by the Hugging Face *Evaluate library*: they are designed to work nicely with the Transformers library. Alternatively, although the Trainer API does not support the streaming metrics from the TorchMetrics library, you can still use them if you wrap them inside a function.

Next, we must specify our training configuration in a `TrainingArguments` object:

```
from transformers import TrainingArguments

train_args = TrainingArguments(
    output_dir="my_imdb_model", num_train_epochs=2,
    per_device_train_batch_size=128, per_device_eval_batch_size=128,
    eval_strategy="epoch", logging_strategy="epoch", save_strategy="epoch",
    load_best_model_at_end=True, metric_for_best_model="accuracy",
    report_to="none")
```

We specify that the logs and model checkpoints must be saved in the `my_imdb_model` directory; training should run for 2 epochs (you can increase this if you want); the batch size is 128 for both training and evaluation (you can tweak this depending on the amount of VRAM you have); we want evaluation, logging, and saving to take place at the end of each epoch; and the best model should be loaded at the end of training based on the validation accuracy. Lastly, the `report_to` argument lets you specify one or more tools that the training code will report logs to, such as TensorBoard or [Weights & Biases \(W&B\)](#). This can be useful to visualize the learning curves. For simplicity, I set `report_to="none"` to turn reporting off.

Lastly, we create a `Trainer` object and pass it the model, along with the training arguments, the training and validation sets, the evaluation function, plus a data collator which will take care of padding. Finally, we call the trainer's `train()` method, and we're done! The model reaches about 90% accuracy on the validation set after just two epochs:

```
from transformers import DataCollatorWithPadding, Trainer

trainer = Trainer(
    bert_for_binary_clf, train_args, train_dataset=tok_imdb_train_set,
    eval_dataset=tok_imdb_valid_set, compute_metrics=compute_accuracy,
    data_collator=DataCollatorWithPadding(bert_tokenizer))
train_output = trainer.train()
```

Great, you now know how to download a pretrained model like BERT and fine-tune it on your own dataset! But what if you don't have a dataset at all, and you just want to use a pretrained model that was already fine-tuned for sentiment analysis? For this, you can use the *pipelines API*.

## Hugging Face Pipelines

The Transformers library provides a very convenient API to download and use pre-trained pipelines for various tasks. Each pipeline contains a pretrained model along with its corresponding preprocessing and post-processing modules. For example let's create a sentiment analysis pipeline and run it on the first 10 IMDb reviews in the training set:

```
>>> from transformers import pipeline
>>> model_name = "distilbert-base-uncased-finetuned-sst-2-english"
>>> classifier_imdb = pipeline("sentiment-analysis", model=model_name,
...                                truncation=True, max_length=512)
...
>>> classifier_imdb(train_reviews[:10])
[{'label': 'POSITIVE', 'score': 0.9996108412742615},
 {'label': 'POSITIVE', 'score': 0.9998623132705688},
 [...]
 {'label': 'POSITIVE', 'score': 0.9978922009468079},
 {'label': 'NEGATIVE', 'score': 0.9997020363807678}]
```

Well, it could hardly be any easier, could it? Just create a pipeline by specifying the task and the model to use, and a couple of other parameters, depending on the task, and off you go! In this example, each review gets a "POSITIVE" or "NEGATIVE" label, along with a score equal to the model's estimated probability for that label. This particular model actually reaches 88.2% accuracy on the validation set, which is reasonably good. Here a few points to note:

- If you don't specify a model, the `pipeline()` function will use the default model for the chosen task. For sentiment analysis, at the time of writing, it's the model we chose: it's a DistilBERT model—a scaled down version of BERT—with an uncased tokenizer, trained on the English Wikipedia and a corpus of English books, and fine-tuned on the Stanford Sentiment Treebank v2 (SST 2) task.
- The pipeline automatically uses the GPU if you have one. If you have several GPUs, you can specify which one to use by setting the pipeline's `device` argument to the GPU index.
- The models from the Transformers library are always in evaluation mode by default (no need to call `eval()`).
- The score is for the chosen label, not for the positive class. In particular, since this is a binary classification task, the score cannot be lower than 0.5 (or else the model would have picked the other label).

## Bias and Fairness

If you try classifying “I am from the USA” and “I am from Iraq”, you will see that the former is classified as very positive, while the latter is classified as very negative. The model is also positive about Thailand but negative about Vietnam. You can try this model with your own country or city; the result may surprise you. Such a bias generally comes in large part from the training data itself: in this case, there were plenty of references to the wars in Iraq and Vietnam in the model’s training data, creating a negative bias. This bias was then amplified during the fine-tuning process since the model was forced to choose between just two classes: positive or negative. If you use a model that was fine-tuned on a dataset with an extra neutral class, then the country bias mostly disappears.

The training data is not the only source of bias: the model’s architecture, the type of loss or regularization used for training, the optimizer—all of these can affect what the model ends up learning. Understanding bias in AI and mitigating its negative effects is still an area of active research, but in any case you should probably pause and think before you rush to deploy a model to production. For example, if you train a model to score resumes, you must ensure that it’s fair. So make sure you evaluate the model’s performance not just on average over the whole test set, but across various subsets as well; for example, you may find that although the model works very well on average, its performance is abysmal for some categories of people. You may also want to run counterfactual tests; for example, check that the model’s predictions do not change when you simply switch someone’s gender or place of birth. The solution depends on the problem: it may require rebalancing the dataset, fine-tuning on a different dataset, switching to another pretrained model, tweaking the model’s architecture or hyperparameters, etc.

Lastly, even if you manage to train a perfectly fair model, it could be *used* in a biased way. For example, the recruiters may use it only for some category of people and not others.

The model we chose is well-suited for general-purpose sentiment analysis, such as movie reviews, but other models are better suited for specific use cases, such as social media posts (e.g., trained on a large dataset of tweets, then fine-tuned on a sentiment analysis dataset). To find the best model for your use case, you can search the list of available models on the [Hugging Face Hub](#). However, there are over 80,000 models available in the “text-classification” category alone, so you will need to use the filters to narrow down the options. In particular, start by filtering on the task, and sort by trending or most-liked models. You can also continue to filter by language and dataset, if necessary. Prefer models from reputable sources (e.g., models from users `huggingface`, `facebook`, `google`, `cardiffnlp`, and so on), and if the model includes executable code, make absolutely sure you trust the user (and if you do, set `trust_remote_code=True` when calling the `pipeline()` function).

There are many text classification tasks other than sentiment analysis. For example, a model fine-tuned on the multi-genre natural language inference (MultiNLI) dataset can classify a pair of texts (each ending with a separation token [SEP]) into three classes: contradiction (if the texts contradict each other), entailment (if the first text entails the second), or neutral otherwise. For example:

```
>>> model_name = "huggingface/distilbert-base-uncased-finetuned-mnli"
>>> classifier_mnli = pipeline("text-classification", model=model_name)
>>> classifier_mnli([
...     "She loves me. [SEP] She loves me not. [SEP]",
...     "Alice just woke up. [SEP] Alice is awake. [SEP]",
...     "I like dogs. [SEP] Everyone likes dogs. [SEP]")]
...
[{'label': 'contradiction', 'score': 0.9717152714729309},
 {'label': 'entailment', 'score': 0.9119168519973755},
 {'label': 'neutral', 'score': 0.9509281516075134}]
```

Many other NLP tasks are also available via the pipeline API, such as question answering, summarization, sentence similarity, text generation, token classification, translation, and more. And it doesn't stop there! There are also many computer vision tasks, such as image classification, image segmentation, object detection, image-to-text, text-to-image, depth estimation, and even audio tasks, such as audio classification, speech-to-text, text-to-speech, and so on. Make sure to check out the full list at <https://huggingface.co/tasks>.



Before you download a model, make sure you trust the hosting platform (e.g., the Hugging Face Hub) and the model's author: the model may contain executable code, which could be malicious. It could also produce biased outputs, or it may have been trained with copyrighted or sensitive private data which might be leaked to your users, or it might even have *poisoned weights* which could make it produce harmful content (e.g., propaganda) only for some types of inputs, otherwise behaving normally.

Time to step back. So far we have looked at text generation using a char-RNN, and sentiment analysis using various subword tokenization methods, pretrained embeddings, and even entire pretrained models. Along the way, we discussed embeddings, tokenizers, and the Hugging Face libraries. In the next section, we will explore another important NLP task: *neural machine translation* (NMT). Specifically, we will build an encoder-decoder model capable of translating English to Spanish, and we will see how to boost its performance using beam search and attention mechanisms. ¡Vamos!

# An Encoder-Decoder Network for Neural Machine Translation

Let's begin with a relatively simple sequence-to-sequence NMT model<sup>15</sup> that will translate English text to Spanish (see Figure 14-5).

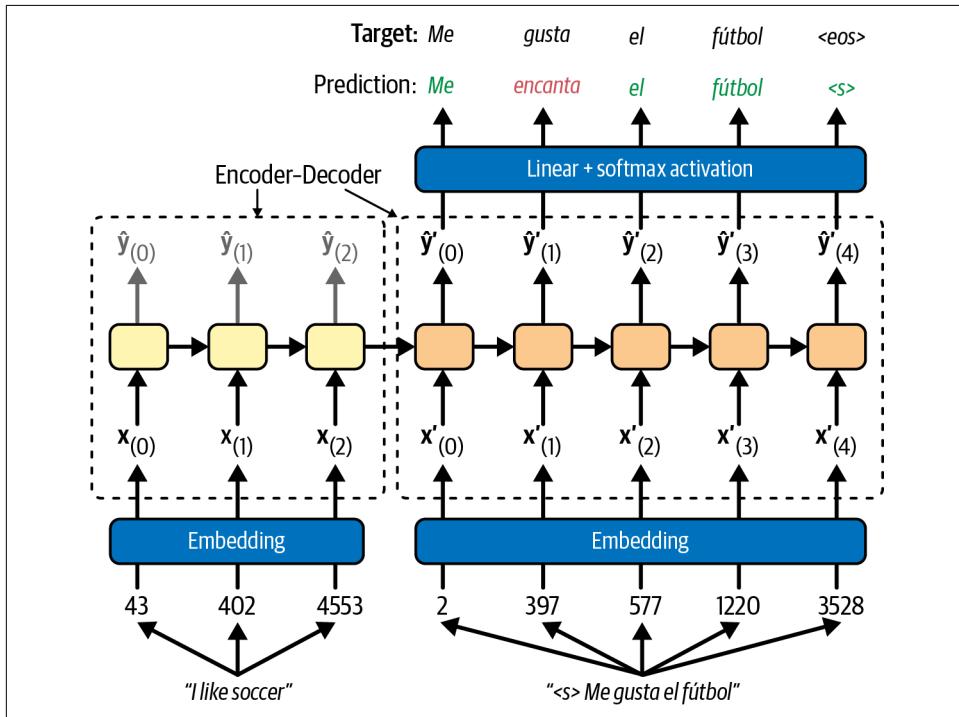


Figure 14-5. A simple machine translation model

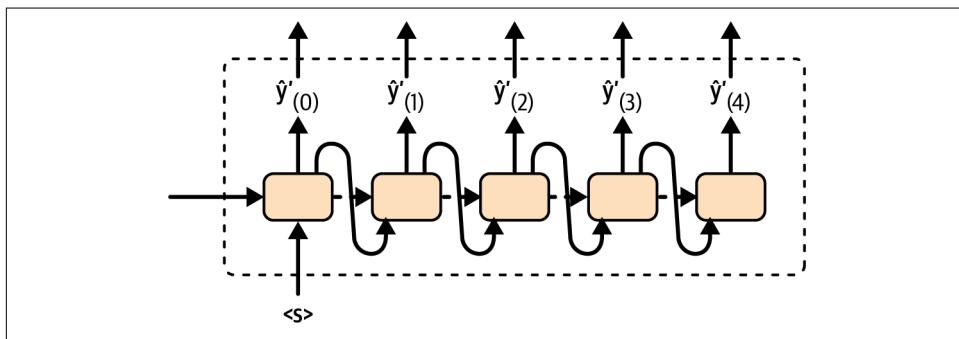
In short, the architecture is as follows: English texts are fed as inputs to the encoder, and the decoder outputs the Spanish translations. Note that the Spanish translations are also used as inputs to the decoder during training, but shifted back by one step. In other words, during training the decoder is given as input the token that it *should* have output at the previous step, regardless of what it actually output. This is called *teacher forcing*—a technique that significantly speeds up training and improves the model's performance. For the very first token, the decoder is given the start-of-sequence (SoS, a.k.a. beginning-of-sequence, BoS) token ("<s>"), and the decoder is expected to end the text with an end-of-sequence (EoS) token ("</s>").

<sup>15</sup> Ilya Sutskever et al., "Sequence to Sequence Learning with Neural Networks", arXiv preprint, arXiv:1409.3215 (2014).

Each token is initially represented by its ID (e.g., 4553 for the token “soccer”). Next, an `nn.Embedding` layer returns the token embedding. These token embeddings are then fed to the encoder and the decoder.

At each step, the decoder’s dense output layer (i.e., an `nn.Linear` layer) outputs a logit score for each token in the output vocabulary (i.e., Spanish). If you pass these logits through the softmax function, you get an estimated probability for each possible token. For example, at the first step the word “Me” may have a probability of 7%, “Yo” may have a probability of 1%, and so on. This is very much like a regular classification task, and indeed we will train the model using the `nn.CrossEntropyLoss`, much like we did in the char-RNN model.

Note that at inference time (after training), you will not have the target text to feed to the decoder. Instead, you need to feed it the word that it has just output at the previous step, as shown in [Figure 14-6](#) (this will require an embedding lookup that is not shown in the diagram).



*Figure 14-6. At inference time, the decoder is fed as input the word it just output at the previous time step*



In a [2015 paper](#),<sup>16</sup> Samy Bengio et al. proposed gradually switching from feeding the decoder the previous *target* token to feeding it the previous *output* token during training.

Let’s build and train this model! First, we need to download a dataset of English/Spanish text pairs. For this, we will use the Datasets library to download English/Spanish pairs from the *Tatoeba Challenge* dataset. The [Tatoeba project](#) is a language-learning initiative started in 2006 by Trang Ho, where contributors have created a

<sup>16</sup> Samy Bengio et al., “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks”, arXiv preprint arXiv:1506.03099 (2015).

huge collection of text pairs from many languages. The Tatoeba Challenge dataset was created by researchers from the University of Helsinki to benchmark machine translation systems, using data extracted from the Tatoeba project. The training set is quite large so we will use the validation set as our training set, setting aside 20% for validation. We will also download the test set:

```
nmt_original_valid_set, nmt_test_set = load_dataset(  
    path="ageron/tatoeba_mt_train", name="eng-spa",  
    split=["validation", "test"])  
split = nmt_original_valid_set.train_test_split(train_size=0.8, seed=42)  
nmt_train_set, nmt_valid_set = split["train"], split["test"]
```

Each sample in the dataset is a dictionary containing an English text along with its Spanish translation. For example:

```
>>> nmt_train_set[0]  
{'source_text': 'Tom tried to break up the fight.',  
 'target_text': 'Tom trató de disolver la pelea.',  
 'source_lang': 'eng',  
 'target_lang': 'spa'}
```

We will need to tokenize this text. We could use a different tokenizer for English and Spanish, but these two languages have many words in common (e.g., animal, color, hotel, hospital, idea, radio, motor), and many similar subwords (e.g., pre, auto, inter, uni), so it makes sense to use a common tokenizer. Let's train a BPE tokenizer on all the training text, both English and Spanish:

```
def train_eng_spa(): # a generator function to iterate over all training text  
    for pair in nmt_train_set:  
        yield pair["source_text"]  
        yield pair["target_text"]  
  
max_length = 256  
vocab_size = 10_000  
nmt_tokenizer_model = tokenizers.models.BPE(unk_token="")  
nmt_tokenizer = tokenizers.Tokenizer(nmt_tokenizer_model)  
nmt_tokenizer.enable_padding(pad_id=0, pad_token="")  
nmt_tokenizer.enable_truncation(max_length=max_length)  
nmt_tokenizer.pre_tokenizer = tokenizers.pre_tokenizers.Whitespace()  
nmt_tokenizer_trainer = tokenizers.trainers.BpeTrainer(  
    vocab_size=vocab_size, special_tokens=[ "<pad>", "<unk>", "<s>", "</s>" ])  
nmt_tokenizer.train_from_iterator(train_eng_spa(), nmt_tokenizer_trainer)
```

Let's test this tokenizer:

```
>>> nmt_tokenizer.encode("I like soccer").ids  
[43, 401, 4381]  
>>> nmt_tokenizer.encode("<s> Me gusta el fútbol").ids  
[2, 396, 582, 219, 3356]
```

Perfect! Now let's create a small utility class that will hold tokenized English texts (i.e., the *source* token ID sequences), along with the corresponding tokenized Spanish

targets (i.e., the *target* token ID sequences), plus the corresponding attention masks. For this, we can create a `namedtuple` base class (i.e., a tuple with named fields), and extend it to add a `to()` method, which will make it easy to move all these tensors to the GPU:

```
from collections import namedtuple

fields = ["src_token_ids", "src_mask", "tgt_token_ids", "tgt_mask"]
class NmtPair(namedtuple("NmtPairBase", fields)):
    def to(self, device):
        return NmtPair(self.src_token_ids.to(device), self.src_mask.to(device),
                      self.tgt_token_ids.to(device), self.tgt_mask.to(device))
```

Next, let's create the data loaders:

```
def nmt_collate_fn(batch):
    src_texts = [pair['source_text'] for pair in batch]
    tgt_texts = [f"<s> {pair['target_text']} </s>" for pair in batch]
    src_encodings = nmt_tokenizer.encode_batch(src_texts)
    tgt_encodings = nmt_tokenizer.encode_batch(tgt_texts)
    src_token_ids = torch.tensor([enc.ids for enc in src_encodings])
    tgt_token_ids = torch.tensor([enc.ids for enc in tgt_encodings])
    src_mask = torch.tensor([enc.attention_mask for enc in src_encodings])
    tgt_mask = torch.tensor([enc.attention_mask for enc in tgt_encodings])
    inputs = NmtPair(src_token_ids, src_mask,
                     tgt_token_ids[:, :-1], tgt_mask[:, :-1])
    labels = tgt_token_ids[:, 1:]
    return inputs, labels

batch_size = 32
nmt_train_loader = DataLoader(nmt_train_set, batch_size=batch_size,
                               collate_fn=nmt_collate_fn, shuffle=True)
nmt_valid_loader = DataLoader(nmt_valid_set, batch_size=batch_size,
                               collate_fn=nmt_collate_fn)
nmt_test_loader = DataLoader(nmt_test_set, batch_size=batch_size,
                            collate_fn=nmt_collate_fn)
```

The `nmt_collate_fn()` function starts by extracting all the English and Spanish texts from the given batch. In the process, it also adds an SoS token at the start of each Spanish text, as well as an EoS token at the end. It then tokenizes both the English and Spanish texts using our BPE tokenizer. Next, the input sequences and the attention masks are converted to tensors and wrapped in an `NmtPair`. Importantly, the function drops the EoS token from the decoder inputs, and drops the SoS token from the decoder targets. For example, the inputs may contain the token IDs for “<s> Me gusta el fútbol”, while the targets may contain the token IDs for “Me gusta el fútbol </s>”. Lastly, the function returns the inputs (i.e., the `NmtPair`) along with the targets. Then we just create the data loaders as usual.

And now we are ready to build our translation model. It's just like [Figure 14-5](#), except the encoder and decoder share the same `nn.Embedding` layer, and the encoder and decoder `nn.GRU` modules contain two layers each:

```
class NmtModel(nn.Module):
    def __init__(self, vocab_size, embed_dim=512, pad_id=0, hidden_dim=512,
                 n_layers=2):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim, padding_idx=pad_id)
        self.encoder = nn.GRU(embed_dim, hidden_dim, num_layers=n_layers,
                             batch_first=True)
        self.decoder = nn.GRU(embed_dim, hidden_dim, num_layers=n_layers,
                             batch_first=True)
        self.output = nn.Linear(hidden_dim, vocab_size)

    def forward(self, pair):
        src_embeddings = self.embed(pair.src_token_ids)
        tgt_embeddings = self.embed(pair.tgt_token_ids)
        src_lengths = pair.src_mask.sum(dim=1)
        src_packed = pack_padded_sequence(
            src_embeddings, lengths=src_lengths.cpu(),
            batch_first=True, enforce_sorted=False)
        _, hidden_states = self.encoder(src_packed)
        outputs, _ = self.decoder(tgt_embeddings, hidden_states)
        return self.output(outputs).permute(0, 2, 1)

torch.manual_seed(42)
vocab_size = nmt_tokenizer.get_vocab_size()
nmt_model = NmtModel(vocab_size).to(device)
```

Almost everything in this model should look familiar: it's very similar to our previous models. We create the modules in the constructor, then the `forward()` method embeds the input sequences (both English and Spanish), it packs the English embeddings and passes them through the encoder, then it passes the Spanish embeddings to the decoder, along with the encoder's last hidden states (across all `nn.GRU` layers). Lastly, the decoder's outputs are passed through the output `nn.Linear` layer, and the final outputs are permuted to ensure that the class dimension (containing the token logits) is the second dimension, since this is expected by the `nn.CrossEntropyLoss` and the `Accuracy` metric, as we saw earlier.



The most common metric used in NMT is the *bilingual evaluation understudy* (BLEU) score, which compares each translation produced by the model with several good translations produced by humans. It counts the number of  $n$ -grams (sequences of  $n$  words) that appear in any of the target translations and adjusts the score to take into account the frequency of the produced  $n$ -grams in the target translations. It is implemented by `TorchMetric`'s `BLEUScore` class.

We could have packed the Spanish embeddings, but then the decoder's outputs would have been packed sequences, which we would have had to pad before we passed them to the output layer. We avoided this complexity because we can just configure the loss to ignore the output tokens when the targets are padding tokens, like this:

```
xentropy = nn.CrossEntropyLoss(ignore_index=0) # ignore <pad> tokens
```

Now you can train this model (e.g., for 10 epochs using a Nadam optimizer with `lr = 0.001`), and it will take quite a while. It's actually not that long when you consider the fact that the model is learning two languages at once!

While it's training, let's write a little helper function to translate some English text to Spanish using our model. It will start by calling the model with the English text for the encoder, and a single SoS token for the decoder. The decoder will just output logits for the first token in the translation. Our function will then pick the most likely token (i.e., with the highest logit) and add it to the decoder inputs, then it will call the model again to get the next token. It will repeat this process, adding one token at a time, until the model outputs an EoS token:

```
def translate(model, src_text, max_length=20, pad_id=0, eos_id=3):
    tgt_text = ""
    token_ids = []
    for index in range(max_length):
        batch, _ = nmt_collate_fn([{"source_text": src_text,
                                    "target_text": tgt_text}])
        with torch.no_grad():
            Y_logits = model(batch.to(device))
            Y_token_ids = Y_logits.argmax(dim=1) # find the best token IDs
            next_token_id = Y_token_ids[0, index] # take the last token ID

            next_token = nmt_tokenizer.id_to_token(next_token_id)
            tgt_text += " " + next_token
            if next_token_id == eos_id:
                break
    return tgt_text
```



This implementation works but it's not optimized at all. We could run the encoder just once on the English text, and we could also run the decoder just once per time step, instead of running it over the whole growing text at each iteration.

Let's try translating some text!

```
>>> nmt_model.eval()
>>> translate(nmt_model, "I like soccer.")
' Me gusta el fútbol . </s>'
```

Hurray, it works! We just built a model from scratch that can translate English to Spanish.

## Model Optimizations

When the output vocabulary is large (e.g., 10,000 tokens or more), computing the `nn.CrossEntropyLoss` can be quite slow, depending on the hardware. To speed things up, one technique is to use *sampled softmax*, introduced in 2015 by Sébastien Jean et al.<sup>17</sup> Instead of computing the softmax over all of the logits, it computes an approximation based on the correct class's logit, as well as a random sample of logits for other classes. This technique requires knowing the target, so it is only useful during training. Moreover, it is not included in PyTorch, so you have to implement it yourself.

Another technique is *adaptive softmax*, introduced in 2016 by Edouard Grave et al.,<sup>18</sup> which speeds up softmax computation by splitting the vocabulary into frequency-based clusters. Frequent tokens are processed normally, while less frequent tokens are placed in progressively larger clusters, reducing computation by only accessing the necessary clusters. This speeds up computations both during training and inference. PyTorch implements this algorithm in the `nn.AdaptiveLogSoftmaxWithLoss` class.

Another thing you can do to speed up training (and also save a lot of memory) is to use the embedding matrix as the weights of the output layer. This is called *tying the weights* of these two layers, and it was first proposed in a 2016 paper by Ofir Press and Lior Wolf.<sup>19</sup> To implement it, just add `self.output.weight = self.embed.weight` to the model's constructor. You can also get rid of the output layer's `bias` parameter, by setting `bias=False` when creating the output layer. Tying the weights significantly reduces the number of model parameters, which speeds up training and may sometimes improve the model's accuracy as well, especially if you don't have a lot of training data. Why does this work? Well, as we saw earlier, the embedding matrix is equivalent to one-hot encoding followed by a linear layer with no bias term and no activation function that maps the one-hot vectors to the embedding space. The output layer does the reverse. So if the model can find an embedding matrix whose transpose is equal to its inverse (such a matrix is called an *orthogonal matrix*), then there's no need to learn a separate set of weights for the output layer.

If you play around with our translation model, you will find that it often works reasonably well on short text, but it really struggles with longer sentences. For example:

<sup>17</sup> Sébastien Jean et al., "On Using Very Large Target Vocabulary for Neural Machine Translation", *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing* 1 (2015): 1–10.

<sup>18</sup> Edouard Grave et al., "Efficient softmax approximation for GPUs", arXiv preprint arXiv:1609.04309 (2016).

<sup>19</sup> Ofir Press, Lior Wolf, "Using the Output Embedding to Improve Language Models", arXiv preprint arXiv:1608.05859 (2016).

```
>>> longer_text = "I like to play soccer with my friends."  
>>> translate(nmt_model, longer_text)  
' Me gusta jugar con mis amigos . </s>'
```

The translation says “I like to play with my friends”. Oops, there’s no mention of soccer. So how can we improve this model? One way is to increase the training set size and add more nn.GRU layers in both the encoder and the decoder. You could also make the encoder bidirectional (but not the decoder, or else it would no longer be causal and it would see the full translation at each time step, instead of just the previous tokens). Another popular technique that can greatly improve the performance of a translation model at inference time is *beam search*.

## Beam Search

To translate an English text to Spanish, we call our model several times, producing one word at a time. Unfortunately, this means that when the model makes one mistake, it is stuck with it for the rest of the translation, which can cause more errors, making the translation worse and worse. For example, suppose we want to translate “I like soccer”, and the model correctly starts with “Me”, but then predicts “gustan” (plural) instead of “gusta” (singular). This mistake is understandable, since “Me gustan” is the correct way to start translating “I like” in many cases. Once the model has made this mistake, it is stuck with “gustan”. It then reasonably adds “los”, which is the plural for “the”. But since the model never saw “los fútbol” in the training data (soccer is singular, not plural), the model tries to find something reasonable to add, and given the context it adds “jugadores”, which means “the players”. So “I like soccer” gets translated to “I like the players”. One error caused a chain of errors.

How can we give the model a chance to go back and fix mistakes it made earlier? One of the most common solutions is *beam search*: it keeps track of a short list of the  $k$  most promising output sequences (say, the top three), and at each decoder step it tries to extend each of them by one word, keeping only the  $k$  most likely sequences. The parameter  $k$  is called the *beam width*.

For example, suppose you use the model to translate the sentence “I like soccer” using beam search with a beam width of three (see Figure 14-7). At the first decoder step, the model will output an estimated probability for each possible first word in the translated sentence. Suppose the top three words are “Me” (75% estimated probability), “a” (3%), and “como” (1%). That’s our short list so far. Next, we use the model to find the next word for each sentence. For the first sentence (“Me”), perhaps the model outputs a probability of 36% for the word “gustan”, 32% for the word “gusta”, 16% for the word “encanta”, and so on. Note that these are actually *conditional* probabilities, given that the sentence starts with “Me”. For the second sentence (“a”), the model might output a conditional probability of 50% for the word

“mi”, and so on. Assuming the vocabulary has 10,000 tokens, we will end up with 10,000 probabilities per sentence.

Next, we compute the probabilities of each of the 30,000 two-token sentences we considered ( $3 \times 10,000$ ). We do this by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes. For example, the estimated probability of the sentence “Me” was 75%, while the estimated conditional probability of the word “gustan” (given that the first word is “Me”) was 36%, so the estimated probability of the sentence “Me gustan” is  $75\% \times 36\% = 27\%$ . After computing the probabilities of all 30,000 two-word sentences, we keep only the top 3. In this example they all start with the word “Me”: “Me gustan” (27%), “Me gusta” (24%), and “Me encanta” (12%). Right now, the sentence “Me gustan” is winning, but “Me gusta” has not been eliminated.

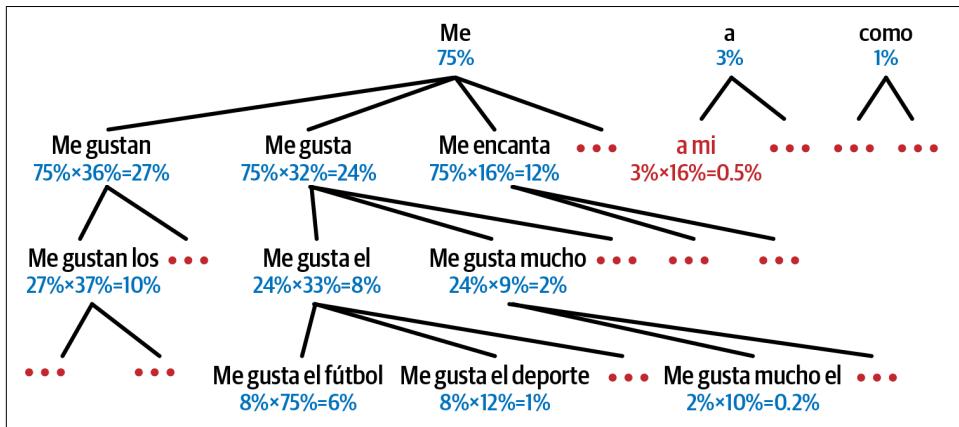


Figure 14-7. Beam search, with a beam width of three

Then we repeat the same process: we use the model to predict the next word in each of these three sentences, and we compute the probabilities of all 30,000 three-word sentences we considered. Perhaps the top 3 are now “Me gustan los” (10%), “Me gusta el” (8%), and “Me gusta mucho” (2%). At the next step we may get “Me gusta el fútbol” (6%), “Me gusta mucho el” (1%), and “Me gusta el deporte” (0.2%). Notice that “Me gustan” was eliminated, and the correct translation is now ahead. We boosted our encoder-decoder model’s performance without any extra training, simply by using it more wisely.

The notebook for this chapter contains a very simple `beam_search()` function, if you’re interested, but in general you will probably want to use the implementation provided by the `GenerationMixin` class in the Transformers library. This is where the text generation models from the Transformers library get their `generate()` method: it accepts a `num_beams` argument which you can set to the desired beam width if you want to use beam search. It also provides a `do_sample` argument that will randomly

sample the next token using the probability distribution output by the model, just like we did earlier with our char-RNN model. Other generation strategies are also supported and can be combined (see <https://hml.info/hfgen> for more details).

With all this, you can get reasonably good translations for fairly short sentences. For example, the following translation is correct:

```
>>> beam_search(nmt_model, longer_text, beam_width=3)
' Me gusta jugar al fútbol con mis amigos . </s>'
```

Unfortunately, this model will still be pretty bad at translating long sentences:

```
>>> longest_text = "I like to play soccer with my friends at the beach."
>>> beam_search(nmt_model, longest_text, beam_width=3)
' Me gusta jugar con jugar con los jug adores de la playa . </s>'
```

This translates to “I like to play with play with the players of the beach”. That’s not quite right. Once again, the problem comes from the limited short-term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem.

## Attention Mechanisms

Consider the path from the word “soccer” to its translation “fútbol” back in [Figure 14-5](#): it is quite long! This means that a representation of this word (along with all the other words) needs to be carried over many steps before it is actually used. Can’t we make this path shorter?

This was the core idea in a landmark [2014 paper<sup>20</sup>](#) by Dzmitry Bahdanau et al., where the authors introduced a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step. For example, at the time step where the decoder needs to output the word “fútbol”, it will focus its attention on the word “soccer”. This means that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact. Attention mechanisms revolutionized neural machine translation (and deep learning in general), allowing a significant improvement in the state of the art, especially for long sentences (e.g., over 30 words).

[Figure 14-8](#) shows our encoder-decoder model with an added attention mechanism:

- On the left, you have the encoder and the decoder (I’ve made the encoder bidirectional in this figure, as it’s generally a good idea).

---

<sup>20</sup> Dzmitry Bahdanau et al., “Neural Machine Translation by Jointly Learning to Align and Translate”, arXiv preprint arXiv:1409.0473 (2014).

- Instead of sending the encoder's final hidden state to the decoder, as well as the previous target word at each time step (which is still done, but it is not shown in the figure), we now send all of the encoder's outputs to the decoder as well.
- Since the decoder cannot deal with all these encoder outputs at once, they need to be aggregated: at each time step, the decoder's memory cell computes a weighted sum of all the encoder outputs. This determines which words the decoder will focus on at this step.
- The weight  $\alpha_{(t,i)}$  is the weight of the  $i^{\text{th}}$  encoder output at the  $t^{\text{th}}$  decoder time step. For example, if the weight  $\alpha_{(3,2)}$  is much larger than the weights  $\alpha_{(3,0)}$  and  $\alpha_{(3,1)}$ , then the decoder will pay much more attention to the encoder's output for word #2 ("soccer") than to the other two outputs, at least at this time step.
- The rest of the decoder works just like earlier: at each time step the memory cell receives the inputs we just discussed, plus the hidden state from the previous time step, and finally (although it is not represented in the diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

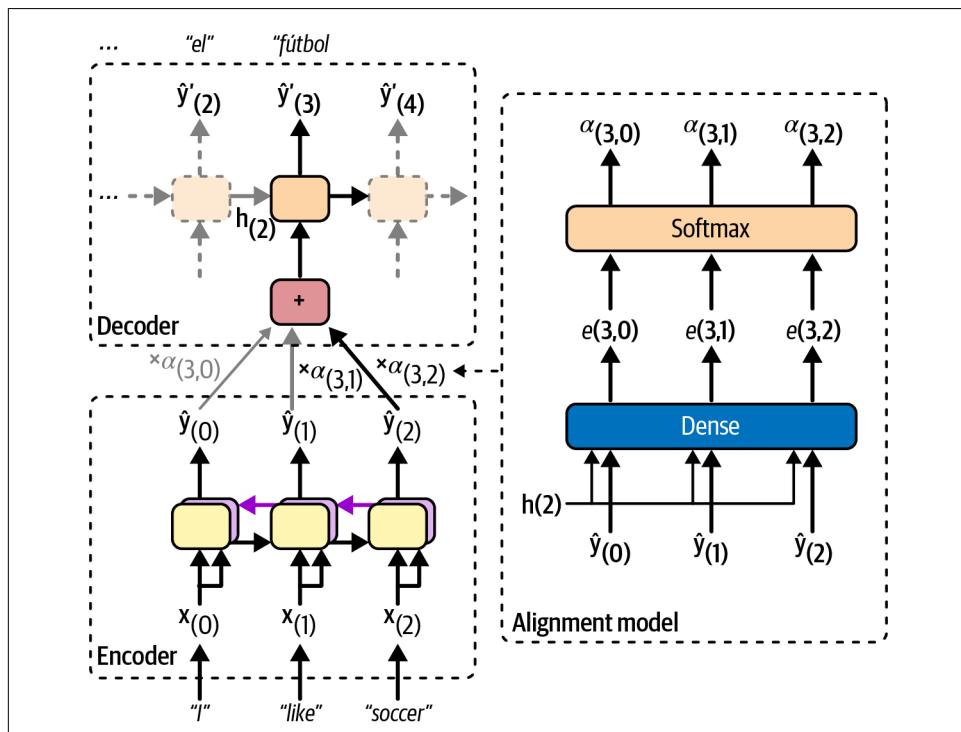


Figure 14-8. Neural machine translation using an encoder-decoder network with an attention model

But where do these  $\alpha_{(t,i)}$  weights come from? Well, they are generated by a small neural network called an *alignment model* (or an *attention layer*), which is trained jointly with the rest of the encoder-decoder model. This alignment model is illustrated on the righthand side of [Figure 14-8](#):

- It starts with a dense layer (i.e., `nn.Linear`) that takes as input each of the encoder’s outputs, along with the decoder’s previous hidden state (e.g.,  $\mathbf{h}_{(2)}$ ), and outputs a score (or energy) for each encoder output (e.g.,  $e_{(3, 2)}$ ). This score measures how well each encoder output is aligned with the decoder’s previous hidden state.

For example, in [Figure 14-8](#), the model has already output “me gusta el” (meaning “I like”), so it’s now expecting a noun. The word “soccer” is the one that best aligns with the current state, so it gets a high score.

- Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g.,  $\alpha_{(3,2)}$ ). All the weights for a given decoder time step add up to 1.

This particular attention mechanism is called *Bahdanau attention* (named after the 2014 paper’s first author). Since it concatenates the encoder output with the decoder’s previous hidden state, it is sometimes called *concatenative attention* (or *additive attention*).

In short, the attention mechanism provides a way to focus the attention of the model on part of the inputs. That said, there’s another way to think of this whole process: it acts as a differentiable memory retrieval mechanism. For example, let’s suppose the encoder analyzed the input sentence “I like soccer”, and it managed to understand that the word “I” is the subject, the word “like” is the verb, and the word “soccer” is the noun, so it encoded this information in its outputs for these words. Now suppose the decoder has already translated “I like”, and it thinks that it should translate the noun next. For this, it needs to fetch the noun from the input sentence. This is analogous to a dictionary lookup: it’s as if the encoder had created a dictionary {"subject": "I", "verb": "like", "noun": "soccer"} and the decoder wanted to look up the value that corresponds to the key “noun”.

However, the model does not have discrete tokens to represent the keys (like “subject”, “verb”, or “noun”); instead, it has vectorized representations of these concepts that it learned during training, so the query it will use for the lookup will not perfectly match any key in the dictionary. One solution is to compute a similarity measure between the query and each key in the dictionary, and then use the softmax function to convert these similarity scores to weights that add up to 1. As we just saw, that’s exactly what the attention layer does. If the key that represents the noun is by far the most similar to the query, then that key’s weight will be close to 1. Next, the attention layer computes a weighted sum of the corresponding values: if the weight

of the “noun” key is close to 1, then the weighted sum will be very close to the representation of the word “soccer”. In short, the decoder queried for a noun and the attention mechanism retrieved it.

In most modern implementations of attention mechanisms, the arguments are named **query**, **key**, and **value**. In our example, the query is the decoder’s hidden states, the key is the encoder’s outputs (this is used to compute the weights), and the value is also the encoder’s outputs (this is used to compute the final weighted sum).



If the input sentence is  $n$  words long, and assuming the output sentence is about as long, then the attention mechanism will need to compute about  $n^2$  weights. This quadratic computational complexity becomes untractable when the sentences are too long.

Another common attention mechanism, known as *Luong attention* or *multiplicative attention*, was proposed shortly after, in 2015,<sup>21</sup> by Minh-Thang Luong et al. Because the goal of the alignment model is to measure the similarity between one of the encoder’s outputs and the decoder’s previous hidden state, the authors proposed to simply compute the dot product (see [Chapter 4](#)) of these two vectors, as this is often a fairly good similarity measure, and modern hardware can compute it very efficiently. For this to be possible, both vectors must have the same dimensionality. The dot product gives a score, and all the scores (at a given decoder time step) go through a softmax layer to give the final weights, just like in Bahdanau attention.

Luong et al. also proposed to use the decoder’s hidden state at the current time step rather than at the previous time step (i.e.,  $\mathbf{h}_{(t)}$  rather than  $\mathbf{h}_{(t-1)}$ ) to compute the attention vector (denoted  $\tilde{\mathbf{h}}_{(t)}$ ). This attention vector is then concatenated with the decoder’s hidden state to form an attentional hidden state, which is then used to predict the next token. This simplifies and speeds up the process by allowing the encoder and decoder to operate independently before attention is applied, rather than interweaving attention into the decoder’s recurrence.

The researchers also proposed a variant of the dot product mechanism where the encoder outputs first go through a fully connected layer (without a bias term) before the dot products are computed. This is called the “general” dot product approach. The researchers compared both dot product approaches with the concatenative attention mechanism (adding a rescaling parameter vector  $\mathbf{v}$ ), and they observed that the dot product variants performed better than concatenative attention. For this reason,

---

<sup>21</sup> Minh-Thang Luong et al., “Effective Approaches to Attention-Based Neural Machine Translation”, *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015): 1412–1421.

concatenative attention is much less used now. The equations for these three attention mechanisms are summarized in [Equation 14-2](#).

*Equation 14-2. Attention mechanisms*

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \hat{\mathbf{y}}_{(i)}$$

with  $\alpha_{(t,i)} = \frac{\exp(e_{(t,i)})}{\sum_{i'} \exp(e_{(t,i')})}$

and  $e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}^\top \hat{\mathbf{y}}_{(i)} & \text{dot} \\ \mathbf{h}_{(t)}^\top \mathbf{W} \hat{\mathbf{y}}_{(i)} & \text{general} \\ \mathbf{v}^\top \tanh(\mathbf{W}[\mathbf{h}_{(t)}, \hat{\mathbf{y}}_{(i)}]) & \text{concat} \end{cases}$

Let's add Luong attention to our encoder-decoder model. Since PyTorch does not include a Luong attention function, we need to write our own. Luckily, it's pretty short:

```
def attention(query, key, value): # note: dq == dk and Lk == Lv
    scores = query @ key.transpose(1, 2) # [B, Lq, dq] @ [B, dk, Lk] = [B, Lq, Lk]
    weights = torch.softmax(scores, dim=-1) # [B, Lq, Lk]
    return weights @ value # [B, Lq, Lk] @ [B, Lv, dv] = [B, Lv, dv]
```

Just like in [Equation 14-2](#), we first compute the attention scores, then we convert them to attention weights using the softmax function, and lastly we compute the attention output by multiplying the attention weights with the value (i.e., the encoder outputs). This implementation efficiently runs all these computations for the whole batch at once. The `query` argument corresponds to  $\mathbf{h}_{(t)}$  in [Equation 14-2](#) (i.e., the decoder's hidden states), and the `key` argument corresponds to  $\hat{\mathbf{y}}_{(i)}$  (i.e., the encoder's outputs), but only for the computation of the attention scores. The `value` argument also corresponds to  $\hat{\mathbf{y}}_{(i)}$ , but only for the final computation of the weighted sum. The `key` and `value` arguments are generally identical, but there are a few scenarios where they can differ (e.g., some models use compressed keys to save memory and speed up the score computation). The shapes are shown in the comments: `B` is the batch size; `Lq` is the length of the longest query in the batch; `Lk` is the length of the longest key in the batch (note that each value must have the same length as its corresponding key); `dq` is the query's embedding size, which must be the same as the key's embedding size `dk`; and `dv` is the value's embedding size.



Since all arguments are 3D tensors, we could replace the @ matrix multiplication operator with the *batch matrix multiplication* function: `torch.bmm()`. This function only works with batches of matrices (i.e., 3D tensors), but it's optimized for this use case so it runs faster. The result is the same: each matrix in the first tensor gets multiplied by the corresponding matrix in the second tensor.

Now let's update our NMT model. The constructor needs just one modification—the output layer's input size must be doubled, since we will concatenate the attention vectors to the decoder outputs:

```
self.output = nn.Linear(2 * hidden_dim, vocab_size)
```

Next, let's add attention to the `forward()` method:

```
def forward(self, pair):
    src_embeddings = self.embed(pair.src_token_ids) # same as earlier
    tgt_embeddings = self.embed(pair.tgt_token_ids) # same
    src_lengths = pair.src_mask.sum(dim=1) # same
    src_packed = pack_padded_sequence(src_embeddings, [...]) # same
    encoder_outputs_packed, hidden_states = self.encoder(src_packed)
    decoder_outputs, _ = self.decoder(tgt_embeddings, hidden_states) # same
    encoder_outputs, _ = pad_packed_sequence(encoder_outputs_packed,
                                             batch_first=True)
    attn_output = attention(query=decoder_outputs,
                           key=encoder_outputs, value=encoder_outputs)
    combined_output = torch.cat((attn_output, decoder_outputs), dim=-1)
    return self.output(combined_output).permute(0, 2, 1)
```

Let's go through this code:

- We compute the English and Spanish embeddings, the English sequence lengths, and we pack the English embeddings, just like earlier.
- We then run the encoder like earlier, but we no longer ignore its outputs since we will need them for the attention function.
- Next, we run the decoder, just like earlier.
- Since the encoder's inputs are represented as a packed sequence, its outputs are also represented as a packed sequence. Not many operations support packed sequences, so we must convert the encoder's outputs to a padded tensor using the `pad_packed_sequence()` function.
- And now we can call our `attention()` function. Note that we pass the decoder outputs instead of the hidden states because the decoder only returns the last hidden states. That's OK because the `nn.GRU` layer's outputs are equal to its top-layer hidden states.

- Lastly, we concatenate the attention output and the decoder outputs along the last dimension, and we pass the result through the output layer. As earlier, we also permute the last two dimensions of the result.



Our attention mechanism doesn't ignore padding tokens. The model learns to ignore them during training, but it's preferable to mask them entirely. We will see how in [Chapter 15](#).

And that's it! If you train this model, you will find that it now handles much longer sentences. For example:

```
>>> translate(nmt_attn_model, longest_text)
' Me gusta jugar fu tbol con mis amigos en la playa . </s>'
```

Perfect! We didn't even have to use beam search. In fact, attention mechanisms turned out to be so powerful that some Google researchers tried getting rid of recurrent layers altogether, only using feedforward layers and attention. Surprisingly, it worked like a charm. This led the researchers to name their paper "Attention is all you need", introducing the Transformer architecture to the world. This was the start of a huge revolution in NLP and beyond. In the next chapter, we will explore the Transformer architecture and see how it revolutionized deep learning.

## Exercises

1. What are the pros and cons of using a stateful RNN versus a stateless RNN?
2. Why do people use encoder-decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?
3. How can you deal with variable-length input sequences? What about variable-length output sequences?
4. What is beam search, and why would you use it? What tool can you use to implement it?
5. What is an attention mechanism? How does it help?
6. When would you need to use sampled softmax?
7. *Embedded Reber grammars* were used by Hochreiter and Schmidhuber in [their paper](#) about LSTMs. They are artificial grammars that produce strings such as "BPBTSXXVPSEPE". Check out Jenny Orr's [nice introduction](#) to this topic, then choose a particular embedded Reber grammar (such as the one represented on Orr's page), and train an RNN to identify whether a string respects that grammar

or not. You will first need to write a function capable of generating a training batch containing about 50% strings that respect the grammar, and 50% that don't.

8. Train an encoder-decoder model that can convert a date string from one format to another (e.g., from “April 22, 2019” to “2019-04-22”).

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.