

cially 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.int32)
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)
0.9288888888888889
```

We got a tiny little accuracy boost. Better than nothing, but not astounding. The problem is that we propagated each representative instance's label to all the instances in the same cluster, including the instances located close to the cluster boundaries, which are more likely to be mislabeled. Let's see what happens if we only propagate the labels to the 20% of the instances that are closest to the centroids:

```
percentile_closest = 20

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:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.9422222222222222
```

Nice! With just 50 labeled instances (only 5 examples per class on average!), we got 94.2% performance, which is pretty close to the performance of logistic regression on the fully labeled digits dataset (which was 96.7%). This is because the propagated labels are actually pretty good, their accuracy is very close to 99%:

```
>>> np.mean(y_train_partially_propagated == y_train[partially_propagated])
0.9896907216494846
```

Active Learning

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

- The model is trained on the labeled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
- The instances for which the model is most uncertain (i.e., when its estimated probability is lowest) must be labeled by the expert.
- Then you just iterate this process again and again, until the performance improvement stops being worth the labeling effort.

Other 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, a Random Forest, and so on).

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

This algorithm defines clusters as continuous regions of high density. It is actually quite simple:

- For each instance, the algorithm counts how many instances are located within a small distance ϵ (epsilon) from it. This region is called the instance's ϵ -neighborhood.
- If an instance has at least `min_samples` instances in its ϵ -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 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 dense enough, and they 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)
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, ...,  3,  2,  3,  3,  4,  2,  6,  3])
```

Notice that some instances have a cluster index equal to -1: this 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:

```
>>> len(dbSCAN.core_sample_indices_)
808
>>> dbSCAN.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
>>> dbSCAN.components_
array([[ -0.02137124,   0.40618608],
       [-0.84192557,   0.53058695],
       ...
       [ -0.94355873,   0.3278936 ],
       [  0.79419406,   0.60777171]])
```

This clustering is represented in the left plot of Figure 9-14. As you can see, it identified quite a lot of anomalies, plus 7 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.

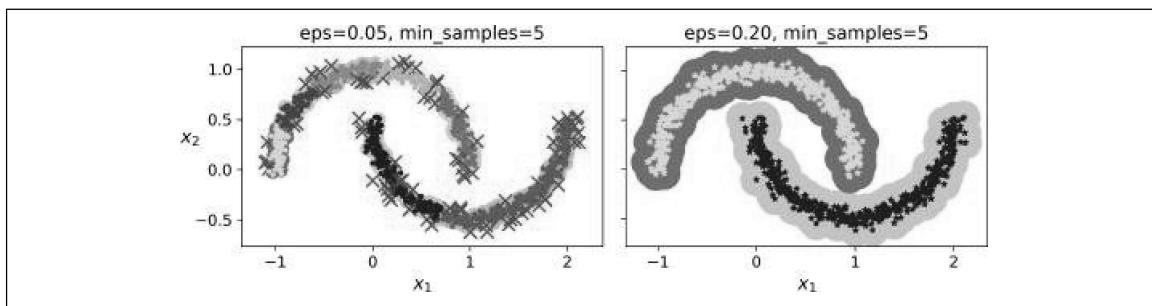


Figure 9-14. DBSCAN clustering using two different neighborhood radiiuses

Somewhat 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. The rationale for this decision is that several classification algorithms could make sense here, and it is easy enough to train one, for example 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 cluster they most likely belong to, and even estimate a probability for each cluster. Note that we only trained them on the core instances, but we could also have chosen to train them on all the instances, or all but the anomalies: this choice depends on the final task.

```
>>> 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. ]])
```

The decision boundary is represented on Figure 9-15 (the crosses represent the 4 instances in `X_new`). Notice that since there is no anomaly in the KNN's training set, the classifier always chooses a cluster, even when that cluster is far away. However, 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, we can 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])
```

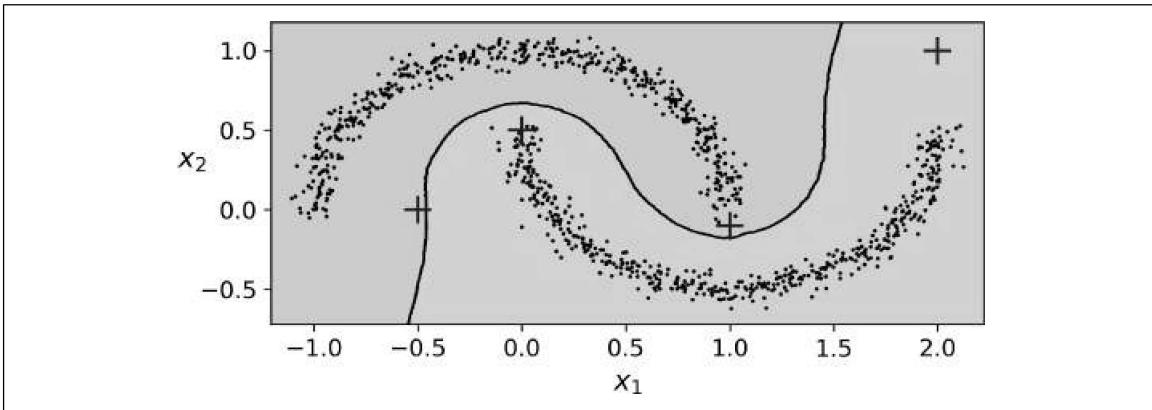


Figure 9-15. *cluster_classification_diagram*

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`). However, if the density varies significantly across the clusters, it can be impossible for it to capture all the clusters properly. Moreover, its computational complexity is roughly $O(m \log m)$, making it pretty close to linear with regards to the number of instances. However, Scikit-Learn's implementation can require up to $O(m^2)$ memory if `eps` is large.

Other Clustering Algorithms

Scikit-Learn implements several more clustering algorithms that you should take a look at. We 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 just one big group of bubbles. Similarly, at each iteration agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If you draw a tree with a branch for every pair of clusters that merged, you get a binary tree of clusters, where the leaves are the individual instances. This approach scales very well to large numbers of instances or clusters, it can capture clusters of various shapes, it 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. This is a sparse m by 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*: this 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). It builds a tree structure during training containing

just enough information to quickly assign each new instance to a cluster, without having to store all the instances in the tree: this allows it to use limited memory, while handle 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-shift step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains). This algorithm 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. This has some of the same features as DBSCAN, in particular it can find any number of clusters of any shape, it has just one hyperparameter (the radius of the circles, called the bandwidth) and it relies on local density estimation. However, it tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is $O(m^2)$, so it is not suited for large datasets.
- *Affinity propagation*: this algorithm uses a voting system, where instances vote for similar instances to be their representatives, and once the algorithm converges, each representative and its voters form a cluster. This algorithm can detect any number of clusters of different sizes. Unfortunately, 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 its 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), however it does not scale well to large number 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 9-11. When you observe an instance, you know it was generated from one of the Gaussian distri-

butions, 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:

- For each instance, a cluster is picked randomly among k clusters. The probability of choosing the j^{th} cluster is defined by the cluster's weight $\phi^{(j)}$.⁷ The index of the cluster chosen for the i^{th} instance is noted $z^{(i)}$.
- If $z^{(i)}=j$, meaning the i^{th} instance has been assigned to the j^{th} cluster, 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 noted $\mathbf{x}^{(i)} \sim \mathcal{N}(\mu^{(j)}, \Sigma^{(j)})$.

This generative process can be represented as a *graphical model* (see Figure 9-16). This is a graph which represents the structure of the conditional dependencies between random variables.

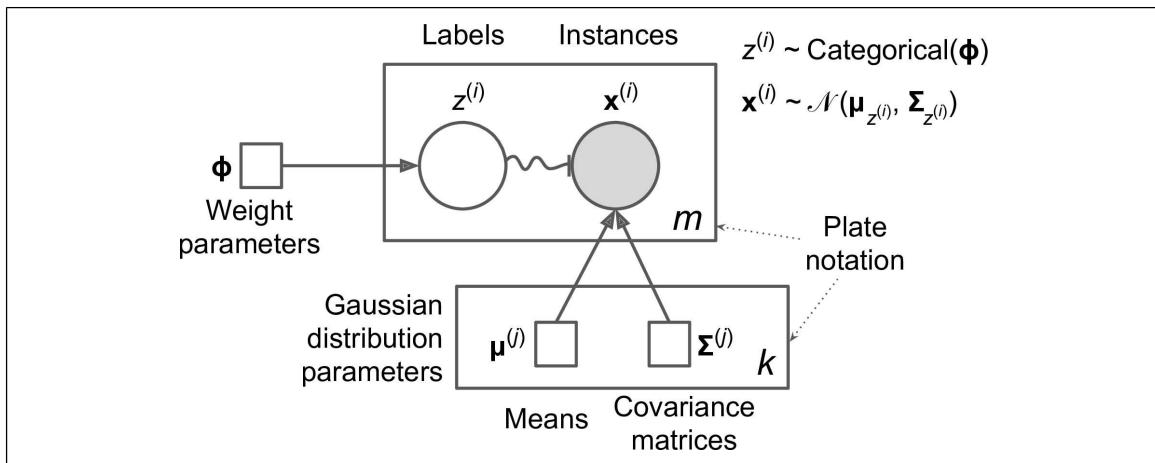


Figure 9-16. Gaussian mixture model

Here is how to interpret it:⁸

- The circles represent random variables.
- The squares represent fixed values (i.e., parameters of the model).

⁷ Phi (ϕ or φ) is the 21st letter of the Greek alphabet.

⁸ Most of these notations are standard, but a few additional notations were taken from the Wikipedia article on plate notation.

- The large rectangles are called *plates*: they indicate that their content is repeated several times.
- The number indicated at the bottom right hand side of each plate indicates how many times its content is repeated, so there are m random variables $z^{(i)}$ (from $z^{(1)}$ to $z^{(m)}$) and m random variables $\mathbf{x}^{(i)}$, and k means $\mu^{(j)}$ and k covariance matrices $\Sigma^{(j)}$, but just one weight vector ϕ (containing all the weights $\phi^{(1)}$ to $\phi^{(k)}$).
- Each variable $z^{(i)}$ is drawn from the *categorical distribution* with weights ϕ . Each variable $\mathbf{x}^{(i)}$ is drawn from the normal distribution with the mean and covariance matrix defined by its cluster $z^{(i)}$.
- The solid arrows represent conditional dependencies. For example, the probability distribution for each random variable $z^{(i)}$ depends on the weight vector ϕ . Note that when an arrow crosses a plate boundary, it means that it applies to all the repetitions of that plate, so for example the weight vector ϕ conditions the probability distributions of all the random variables $\mathbf{x}^{(1)}$ to $\mathbf{x}^{(m)}$.
- The squiggly arrow from $z^{(i)}$ to $\mathbf{x}^{(i)}$ represents a switch: depending on the value of $z^{(i)}$, the instance $\mathbf{x}^{(i)}$ will be sampled from a different Gaussian distribution. For example, if $z^{(i)}=j$, then $\mathbf{x}^{(i)} \sim \mathcal{N}(\mu^{(j)}, \Sigma^{(j)})$.
- Shaded nodes indicate that the value is known, so in this case only the random variables $\mathbf{x}^{(i)}$ have known values: they are called *observed variables*. The unknown random variables $z^{(i)}$ are called *latent variables*.

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

```
from sklearn.mixture import GaussianMixture

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

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

```
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717,  1.05933727],
       [-1.40763984,  1.42710194],
       [ 0.05135313,  0.07524095]])
>>> gm.covariances_
array([[[[ 1.14807234, -0.03270354],
         [-0.03270354,  0.95496237]],
        [[ 0.63478101,  0.72969804],
         [ 0.72969804,  1.1609872 ]],
```

```
[[ 0.68809572,  0.79608475],
 [ 0.79608475,  1.21234145]])
```

Great, it worked fine! Indeed, the weights that were used to generate the data were 0.2, 0.4 and 0.4, and similarly, the means and covariance matrices were very 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*) then updating the clusters (this is called the *maximization step*). Sounds familiar? Indeed, in the context of clustering you can think of EM as a generalization of K-Means which not only finds the cluster centers ($\mu^{(1)}$ to $\mu^{(k)}$), but also their size, shape and orientation ($\Sigma^{(1)}$ to $\Sigma^{(k)}$), as well as their relative weights ($\phi^{(1)}$ to $\phi^{(k)}$). Unlike K-Means, EM uses soft cluster assignments rather than 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 only set to 1.

You can check whether or not the algorithm converged and how many iterations it took:

```
>>> gm.converged_
True
>>> gm.n_iter_
3
```

Okay, now that you have an estimate of the location, size, shape, 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). For this, just use the `predict()` method for hard clustering, or the `predict_proba()` method for soft clustering:

```
>>> gm.predict(X)
array([2, 2, 1, ..., 0, 0, 0])
>>> gm.predict_proba(X)
array([[2.32389467e-02, 6.77397850e-07, 9.76760376e-01],
       [1.64685609e-02, 6.75361303e-04, 9.82856078e-01],
```

```
[2.01535333e-06, 9.99923053e-01, 7.49319577e-05],
...,
[9.99999571e-01, 2.13946075e-26, 4.28788333e-07],
[1.00000000e+00, 1.46454409e-41, 5.12459171e-16],
[1.00000000e+00, 8.02006365e-41, 2.27626238e-15]])
```

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

```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[ 2.95400315,  2.63680992],
       [-1.16654575,  1.62792705],
       [-1.39477712, -1.48511338],
       [ 0.27221525,  0.690366 ],
       [ 0.54095936,  0.48591934],
       [ 0.38064009, -0.56240465]])
```



```
>>> y_new
array([0, 1, 2, 2, 2, 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)
array([-2.60782346, -3.57106041, -3.33003479, ...,
       -3.51352783,
       -4.39802535, -3.80743859])
```

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 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 9-17 shows the cluster means, the decision boundaries (dashed lines), and the density contours of this model:

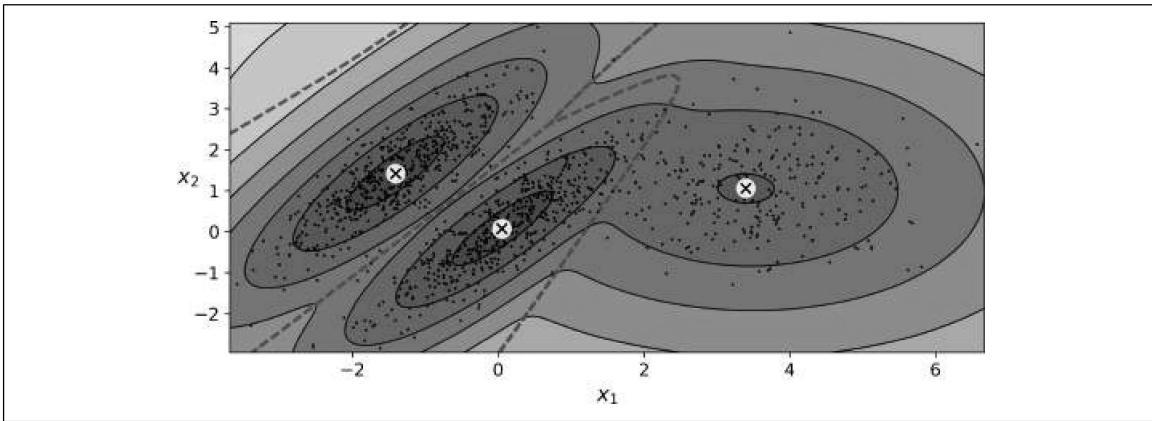


Figure 9-17. Cluster means, decision boundaries and density contours of a trained Gaussian mixture model

Nice! The algorithm clearly found an excellent solution. Of course, we made its task easy by actually generating the data using a set of 2D Gaussian distributions (unfortunately, real life data is not always so Gaussian and low-dimensional), and 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, just set the `covariance_type` hyper-parameter 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 9-18 plots the solutions found by the EM algorithm when `covariance_type` is set to "tied" or "spherical".

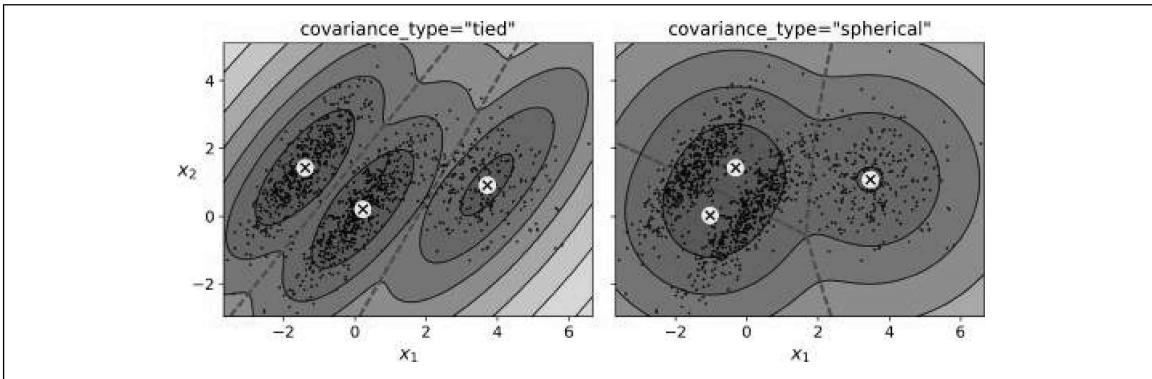


Figure 9-18. `covariance_type`_diagram



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. Let's see how.

Anomaly Detection using Gaussian Mixtures

Anomaly detection (also called *outlier detection*) is the task of detecting instances that deviate strongly from the norm. These instances are of course called *anomalies* or *outliers*, while the normal instances are called *inliers*. Anomaly detection is very useful in a wide variety of applications, for example in fraud detection, or for detecting defective products in manufacturing, or to remove outliers from a dataset before training another model, which can significantly improve the performance of the resulting model.

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 4%, then you can set the density threshold to be the value that results in having 4% 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 tradeoff (see Chapter 3). Here is how you would identify the outliers using the 4th percentile low-

est density as the threshold (i.e., approximately 4% of the instances will be flagged as anomalies):

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

These anomalies are represented as stars on Figure 9-19:

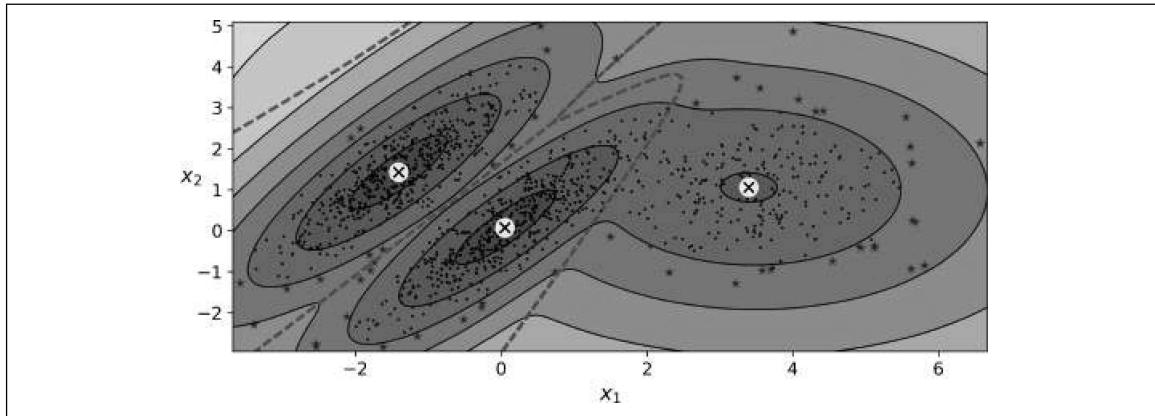


Figure 9-19. Anomaly detection using a Gaussian mixture model

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 precisely used to clean up a dataset.



Gaussian mixture models try to fit all the data, including the outliers, so if you have too many of them, this will bias the model’s view of “normality”: 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 it?

Selecting the Number of Clusters

With K-Means, you could 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*.

mation criterion such as the *Bayesian information criterion* (BIC) or the *Akaike information criterion* (AIC), defined in Equation 9-1.

Equation 9-1. Bayesian information criterion (BIC) and Akaike information criterion (AIC)

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

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

- m is the number of instances, as always.
- p is the number of parameters learned by the model.
- \hat{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, but when they differ, the model selected by the BIC tends to be simpler (fewer parameters) than the one selected by the AIC, but it does 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 the English language, but they have very different meanings in statistics: given a statistical model with some parameters θ , the word “probability” is used to describe how plausible a future outcome x is (knowing the parameter values θ), while the word “likelihood” is used to describe how plausible a particular set of parameter values θ are, after the outcome x is known.

Consider a one-dimensional mixture model of two Gaussian distributions centered at -4 and +1. For simplicity, this toy model has a single parameter θ that controls the standard deviations of both distributions. The top left contour plot in Figure 9-20 shows the entire model $f(x; \theta)$ as a function of both x and θ . To estimate the probability distribution of a future outcome x , you need to set the model parameter θ . For example, if you set it to $\theta=1.3$ (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). On the other hand, if you have observed a single instance $x=2.5$ (the vertical line in the upper left plot), you get the likelihood function noted $\mathcal{L}(\theta|x=2.5)=f(x=2.5; \theta)$ represented in the upper right plot.

In short, the PDF is a function of x (with θ fixed) while the likelihood function is a function of θ (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 θ , the result can be any positive value.

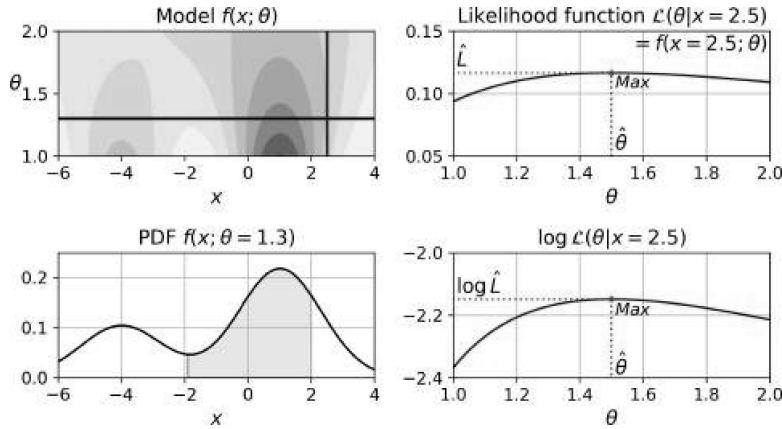


Figure 9-20. A model's parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right) and a log likelihood function (lower right)

Given a dataset \mathbf{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 \mathbf{X} . In this example, if you have observed a single instance $x=2.5$, the *maximum likelihood estimate* (MLE) of θ is $\hat{\theta}=1.5$. If a prior probability distribution g over θ 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 it is equivalent to maximize the likelihood function or to maximize its logarithm (represented in the lower right hand side of Figure 9-20): indeed, the logarithm is a strictly increasing function, so if θ 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 θ 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 θ that maximizes the likelihood function, then you are ready to compute $\hat{L} = \mathcal{L}(\hat{\theta}, \mathbf{X})$. This is the value which is 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, just call the `bic()` or `aic()` methods:

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```

Figure 9-21 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. Note that we could also search for the best value for the `covariance_type` hyperparameter. For example, if it is "spherical" rather than "full", then the model has much fewer parameters to learn, but it does not fit the data as well.

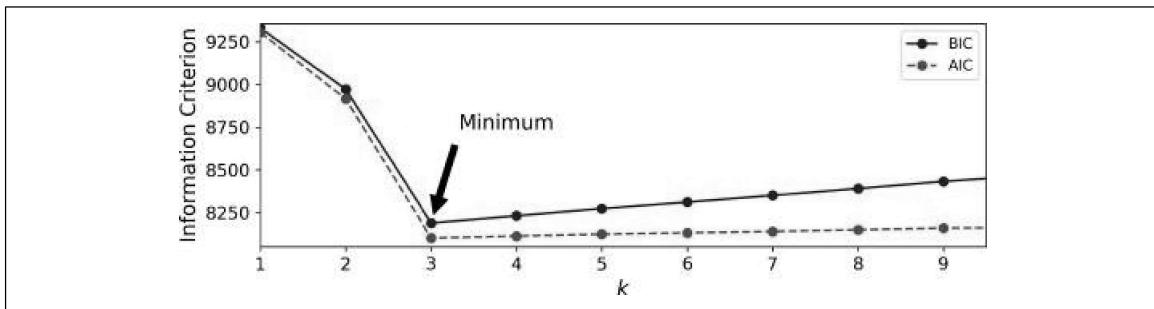


Figure 9-21. AIC and BIC for different numbers of clusters k

Bayesian Gaussian Mixture Models

Rather than manually searching for the optimal number of clusters, it is possible to use instead the `BayesianGaussianMixture` class which is capable of giving weights equal (or close) to zero to unnecessary clusters. Just 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, random_state=42)
>>> bgm.fit(X)
>>> np.round(bgm.weights_, 2)
array([0.4 , 0.21, 0.4 , 0. , 0. , 0. , 0. , 0. , 0. , 0. ])
```

Perfect: the algorithm automatically detected that only 3 clusters are needed, and the resulting clusters are almost identical to the ones in Figure 9-17.

In this model, the cluster parameters (including the weights, means and covariance matrices) are not treated as fixed model parameters anymore, but as latent random variables, like the cluster assignments (see Figure 9-22). So `z` now includes both the cluster parameters and the cluster assignments.

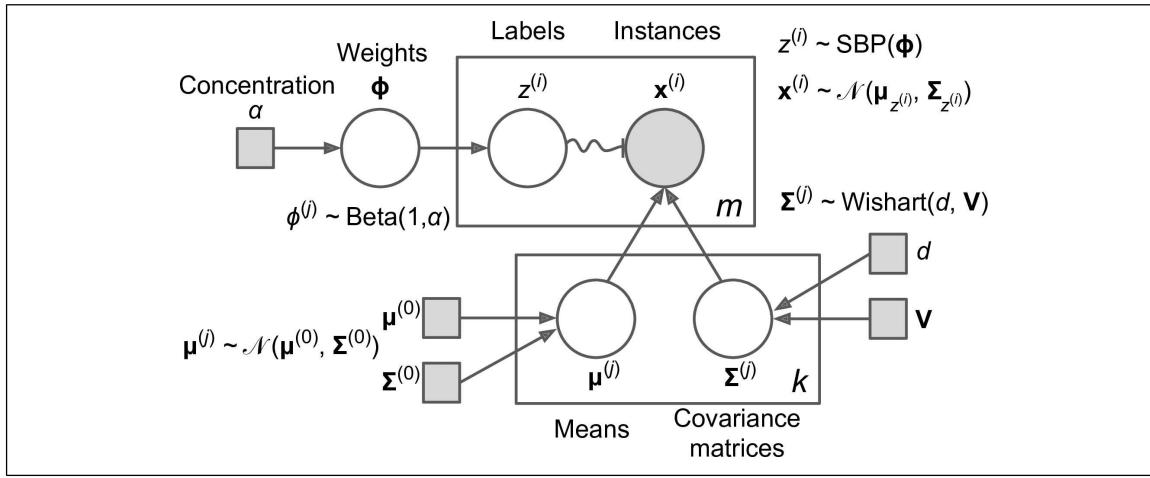


Figure 9-22. Bayesian Gaussian mixture model

Prior knowledge about the latent variables \mathbf{z} can be encoded in a probability distribution $p(\mathbf{z})$ called the *prior*. For example, we may have a prior belief that the clusters are likely to be few (low concentration), or conversely, that they are more likely to be plentiful (high concentration). This can be adjusted using the `weight_concentration_prior` hyperparameter. Setting it to 0.01 or 1000 gives very different clusterings (see Figure 9-23). However, the more data we have, the less the priors matter. In fact, to plot diagrams with such large differences, you must use very strong priors and little data.

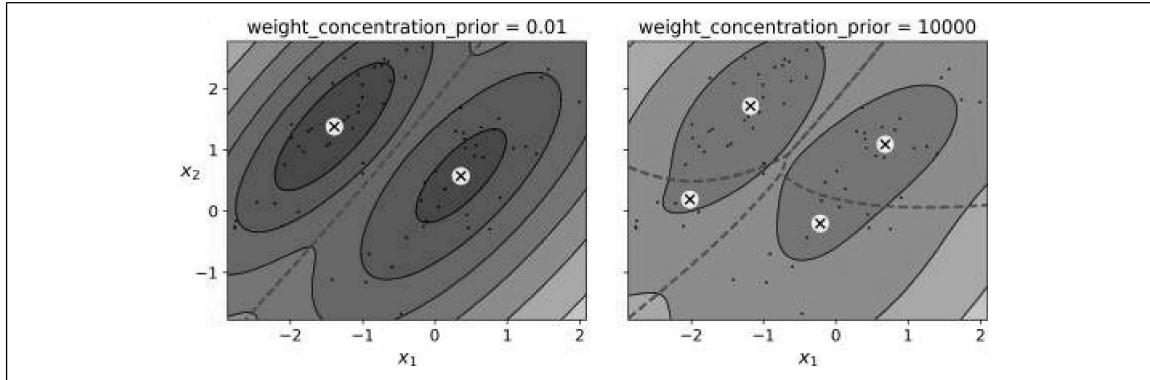


Figure 9-23. Using different concentration priors



The fact that you see only 3 regions in the right plot although there are 4 centroids is not a bug: the weight of the top-right cluster is much larger than the weight of the lower-right cluster, so the probability that any given point in this region belongs to the top-right cluster is greater than the probability that it belongs to the lower-right cluster, even near the lower-right cluster.

Bayes' theorem (Equation 9-2) tells us how to update the probability distribution over the latent variables after we observe some data \mathbf{X} . It computes the *posterior* distribution $p(\mathbf{z}|\mathbf{X})$, which is the conditional probability of \mathbf{z} given \mathbf{X} .

Equation 9-2. Bayes' theorem

$$p(\mathbf{z}|\mathbf{X}) = \text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}} = \frac{p(\mathbf{X}|\mathbf{z}) p(\mathbf{z})}{p(\mathbf{X})}$$

Unfortunately, in a Gaussian mixture model (and many other problems), the denominator $p(\mathbf{x})$ is intractable, as it requires integrating over all the possible values of \mathbf{z} (Equation 9-3). This means considering all possible combinations of cluster parameters and cluster assignments.

Equation 9-3. The evidence $p(\mathbf{X})$ is often intractable

$$p(\mathbf{X}) = \int p(\mathbf{X}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z}$$

This is one of the central problems in Bayesian statistics, and there are several approaches to solving it. One of them is *variational inference*, which picks a family of distributions $q(\mathbf{z}; \lambda)$ with its own *variational parameters* λ (lambda), then it optimizes these parameters to make $q(\mathbf{z})$ a good approximation of $p(\mathbf{z}|\mathbf{X})$. This is achieved by finding the value of λ that minimizes the KL divergence from $q(\mathbf{z})$ to $p(\mathbf{z}|\mathbf{X})$, noted $D_{KL}(q||p)$. The KL divergence equation is shown in (see Equation 9-4), and it can be rewritten as the log of the evidence ($\log p(\mathbf{X})$) minus the *evidence lower bound* (ELBO). Since the log of the evidence does not depend on q , it is a constant term, so minimizing the KL divergence just requires maximizing the ELBO.

Equation 9-4. KL divergence from $q(\mathbf{z})$ to $p(\mathbf{z}|\mathbf{X})$

$$\begin{aligned} D_{KL}(q \parallel p) &= \mathbb{E}_q \left[\log \frac{q(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{X})} \right] \\ &= \mathbb{E}_q [\log q(\mathbf{z}) - \log p(\mathbf{z} \mid \mathbf{X})] \\ &= \mathbb{E}_q \left[\log q(\mathbf{z}) - \log \frac{p(\mathbf{z}, \mathbf{X})}{p(\mathbf{X})} \right] \\ &= \mathbb{E}_q [\log q(\mathbf{z}) - \log p(\mathbf{z}, \mathbf{X}) + \log p(\mathbf{X})] \\ &= \mathbb{E}_q [\log q(\mathbf{z})] - \mathbb{E}_q [\log p(\mathbf{z}, \mathbf{X})] + \mathbb{E}_q [\log p(\mathbf{X})] \\ &= \mathbb{E}_q [\log p(\mathbf{X})] - (\mathbb{E}_q [\log p(\mathbf{z}, \mathbf{X})] - \mathbb{E}_q [\log q(\mathbf{z})]) \\ &= \log p(\mathbf{X}) - \text{ELBO} \end{aligned}$$

where $\text{ELBO} = \mathbb{E}_q [\log p(\mathbf{z}, \mathbf{X})] - \mathbb{E}_q [\log q(\mathbf{z})]$

In practice, there are different techniques to maximize the ELBO. In *mean field variational inference*, it is necessary to pick the family of distributions $q(\mathbf{z}; \boldsymbol{\lambda})$ and the prior $p(z)$ very carefully to ensure that the equation for the ELBO simplifies to a form that can actually be computed. Unfortunately, there is no general way to do this, it depends on the task and requires some mathematical skills. For example, the distributions and lower bound equations used in Scikit-Learn's `BayesianGaussianMixture` class are presented in the documentation. From these equations it is possible to derive update equations for the cluster parameters and assignment variables: these are then used very much like in the Expectation-Maximization algorithm. In fact, the computational complexity of the `BayesianGaussianMixture` class is similar to that of the `GaussianMixture` class (but generally significantly slower). A simpler approach to maximizing the ELBO is called *black box stochastic variational inference* (BBSVI): at each iteration, a few samples are drawn from q and they are used to estimate the gradients of the ELBO with regards to the variational parameters $\boldsymbol{\lambda}$, which are then used in a gradient ascent step. This approach makes it possible to use Bayesian inference with any kind of model (provided it is differentiable), even deep neural networks: this is called Bayesian deep learning.



If you want to dive deeper into Bayesian statistics, check out the *Bayesian Data Analysis* book by Andrew Gelman, John Carlin, Hal Stern, David Dunson, Aki Vehtari, and Donald Rubin.

Gaussian mixture models work great on clusters with ellipsoidal shapes, but if you try to fit a dataset with different shapes, you may have bad surprises. For example, let's see what happens if we use a Bayesian Gaussian mixture model to cluster the moons dataset (see Figure 9-24):

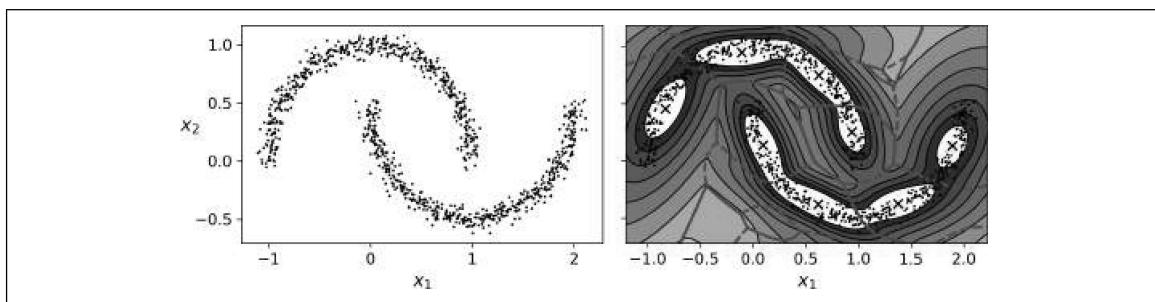


Figure 9-24. `moons_vs_bgm_diagram`

Oops, the algorithm desperately searched for ellipsoids, so it found 8 different clusters instead of 2. 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. Let's now look at a few clustering algorithms capable of dealing with arbitrarily shaped clusters.

Other Anomaly Detection and Novelty Detection Algorithms

Scikit-Learn also implements a few 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 cleanup a dataset. It assumes that the normal instances (inliers) are generated from a single Gaussian distribution (not a mixture), but it also assumes that the dataset is contaminated with outliers that were not generated from this Gaussian distribution. When it 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 gives a better estimation of the elliptic envelope, and thus makes it 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 value) 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. An anomaly is usually far from other instances, so on average (across all the Decision Trees) it tends to get isolated in less 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 Chapter 5). 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 just like all SVMs, it does not scale to large datasets.

PART II

Neural Networks and Deep Learning

Introduction to Artificial Neural Networks with Keras



With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 10 in the final release of the book.

Birds inspired us to fly, burdock plants inspired velcro, and countless more inventions were inspired by nature. It seems only logical, then, to look at the brain’s architecture for inspiration on how to build an intelligent machine. This is the key idea that sparked *artificial neural networks* (ANNs). However, although planes were inspired by birds, they don’t have to flap their wings. 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.¹

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), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of *Go* by playing millions of games against itself (DeepMind’s Alpha-Zero).

¹ 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.

In the first part of this chapter, we will introduce artificial neural networks, starting with a quick tour of the very first ANN architectures, leading up to *Multi-Layer Perceptrons* (MLPs) which are heavily used today (other architectures will be explored in the next chapters). In the second part, we will look at how to implement neural networks using the popular Keras API. This is a beautifully designed and simple high-level API for building, training, evaluating and running neural networks. But don't be fooled by its simplicity: it is expressive and flexible enough to let you build a wide variety of neural network architectures. In fact, it will probably be sufficient for most of your use cases. Moreover, should you ever need extra flexibility, you can always write custom Keras components using its lower-level API, as we will see in Chapter 12.

But first, let's go back in time to see how artificial neural networks came to be!

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,² “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 we will see.

The early successes of ANNs until the 1960s led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear 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 there was a revival of interest in *connectionism* (the study of neural networks), as new architectures were invented and better training techniques were developed. But progress was slow, and by the 1990s other powerful Machine Learning techniques were invented, such as Support Vector Machines (see Chapter 5). These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks entered a long winter.

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

² “A Logical Calculus of Ideas Immanent in Nervous Activity,” W. McCulloch and W. Pitts (1943).

- 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, but also thanks to the gaming industry, which has produced powerful GPU cards by the millions.
- 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 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 rather rare in practice (or when it is the case, they are usually fairly close to the global optimum).
- 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.

Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron (represented in Figure 10-1). It is an unusual-looking cell mostly found in animal cerebral cortices (e.g., your brain), composed of a *cell body* containing the nucleus and most of the cell's complex components, and 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 directly to the cell body) of other neurons. Biological neurons receive short electrical impulses called *signals* from other neurons via these synapses. When a neuron receives a sufficient number of signals from other neurons within a few milliseconds, it fires its own signals.

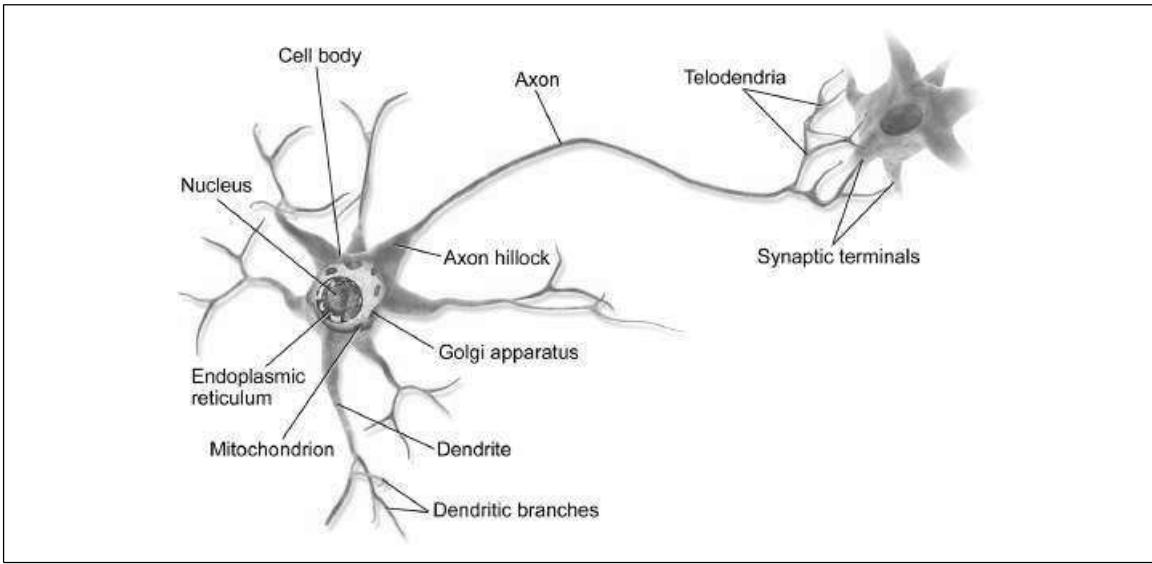


Figure 10-1. Biological neuron³

Thus, individual biological neurons seem to behave in a rather simple way, but they are organized in a vast network of billions of neurons, each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a vast 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 (BNN)⁴ is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, as shown in Figure 10-2.

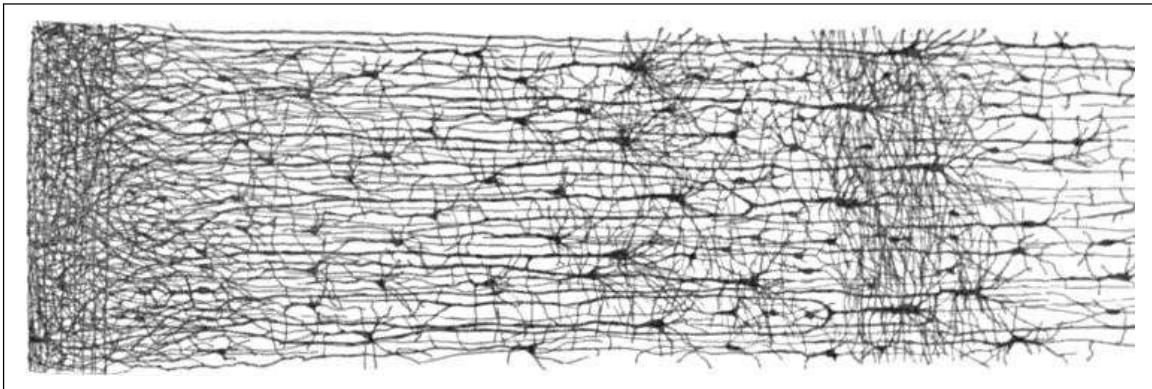


Figure 10-2. Multiple layers in a biological neural network (human cortex)⁵

³ Image by Bruce Blaus (Creative Commons 3.0). Reproduced from <https://en.wikipedia.org/wiki/Neuron>.

⁴ In the context of Machine Learning, the phrase “neural networks” generally refers to ANNs, not BNNs.

⁵ Drawing of a cortical lamination by S. Ramon y Cajal (public domain). Reproduced from https://en.wikipedia.org/wiki/Cerebral_cortex.

Logical Computations with Neurons

Warren McCulloch and Walter 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 simply activates its output when more than a certain number of its inputs are active. McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want. For example, let's build a few ANNs that perform various logical computations (see Figure 10-3), assuming that a neuron is activated when at least two of its inputs are active.

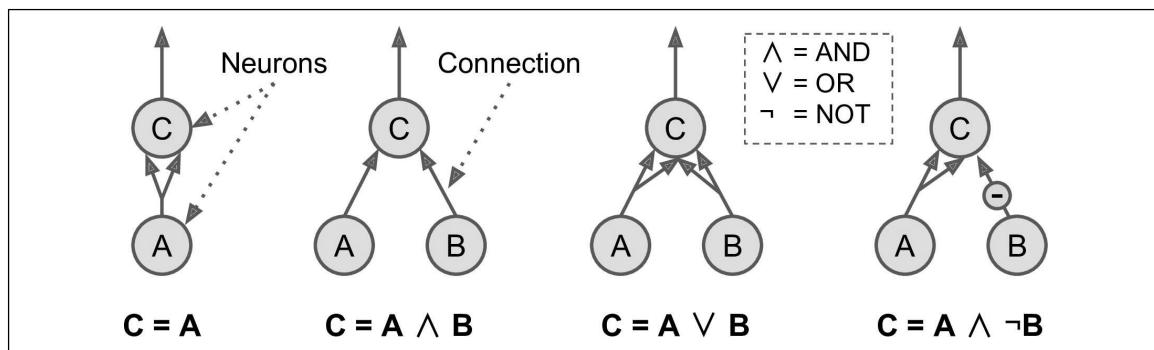


Figure 10-3. ANNs performing simple logical computations

- The first network on the left is simply 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.
- 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 if 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 easily imagine how these networks can be combined to compute complex logical expressions (see the exercises at the end of the chapter).

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 10-4) called

a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU): the inputs and output are now numbers (instead of binary on/off values) and each input connection is associated with a weight. The TLU computes a weighted sum of its inputs ($z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n = \mathbf{x}^T \mathbf{w}$), then applies a *step function* to that sum and outputs the result: $h_w(\mathbf{x}) = \text{step}(z)$, where $z = \mathbf{x}^T \mathbf{w}$.

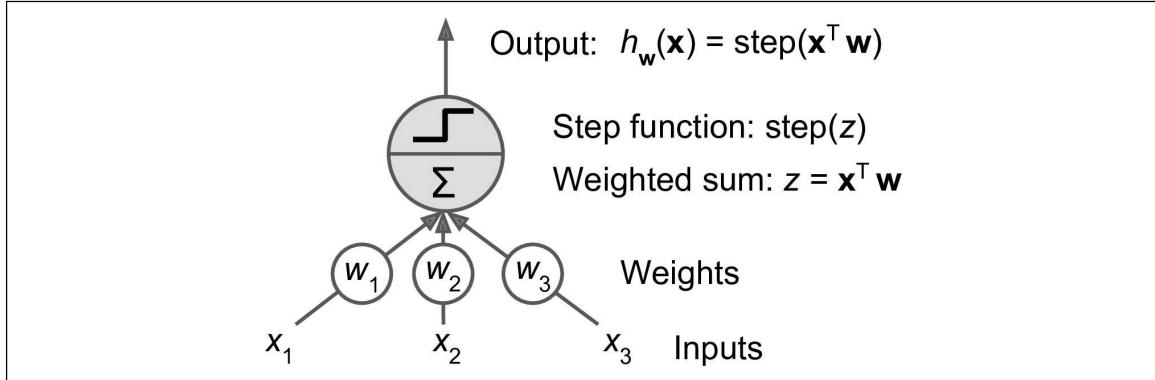


Figure 10-4. Threshold logic unit

The most common step function used in Perceptrons is the *Heaviside step function* (see Equation 10-1). Sometimes the sign function is used instead.

Equation 10-1. Common step functions used in Perceptrons

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

A single TLU can be used for simple linear binary classification. It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a Logistic Regression classifier or a linear SVM). For example, you could use a single TLU to classify iris flowers based on the petal length and width (also adding an extra bias feature $x_0 = 1$, just like we did in previous chapters). Training a TLU in this case means finding the right values for w_0 , w_1 , and w_2 (the training algorithm is discussed shortly).

A Perceptron is simply composed of a single layer of TLUs,⁶ with each TLU connected to all the inputs. When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), it is called a *fully connected layer* or a *dense layer*. To represent the fact that each input is sent to every TLU, it is common to draw special passthrough neurons called *input neurons*: they just output whatever input they are fed. All the input neurons form the *input layer*. Moreover, an extra bias fea-

⁶ The name *Perceptron* is sometimes used to mean a tiny network with a single TLU.

ture is generally added ($x_0 = 1$): it is typically represented using a special type of neuron called a *bias neuron*, which just outputs 1 all the time. A Perceptron with two inputs and three outputs is represented in Figure 10-5. This Perceptron can classify instances simultaneously into three different binary classes, which makes it a multi-output classifier.

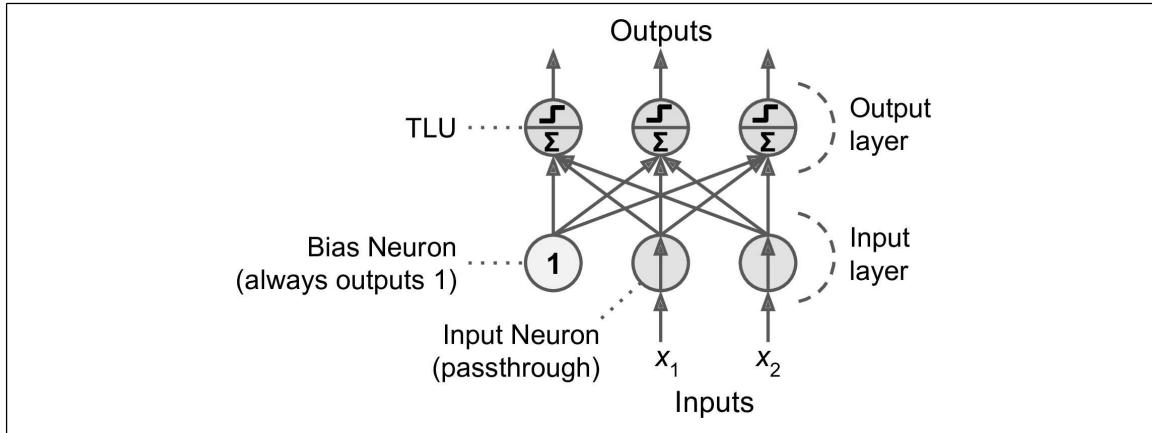


Figure 10-5. Perceptron diagram

Thanks to the magic of linear algebra, it is possible to efficiently compute the outputs of a layer of artificial neurons for several instances at once, by using Equation 10-2:

Equation 10-2. Computing the outputs of a fully connected layer

$$h_{\mathbf{W}, \mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

- As always, \mathbf{X} represents the matrix of input features. It has one row per instance, one column per feature.
- The weight matrix \mathbf{W} contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector \mathbf{b} contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function ϕ is called the *activation function*: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).

So how is a Perceptron trained? The Perceptron training algorithm proposed by Frank Rosenblatt was largely inspired by *Hebb's rule*. In his book *The Organization of Behavior*, published in 1949, Donald Hebb suggested that when a biological neuron often triggers another neuron, the connection between these two neurons grows stronger. This idea was later summarized by Siegrid Löwel in this catchy phrase: “Cells that fire together, wire together.” This rule later became known as Hebb’s rule

(or *Hebbian learning*); that is, the connection weight between two neurons is increased whenever they have the same output. Perceptrons are trained using a variant of this rule that takes into account the error made by the network; it reinforces connections that help reduce the error. More specifically, the 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 10-3.

Equation 10-3. Perceptron learning rule (weight update)

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

- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- \hat{y}_j is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

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 would converge to a solution.⁷ This is called the *Perceptron convergence theorem*.

Scikit-Learn provides a `Perceptron` class that implements a single TLU network. It 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()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype(np.int) # Iris Setosa?

per_clf = Perceptron()
per_clf.fit(X, y)
```

⁷ Note that this solution is generally not unique: in general when the data are linearly separable, there is an infinity of hyperplanes that can separate them.

```
y_pred = per_clf.predict([[2, 0.5]])
```

You may have noticed the fact that the Perceptron learning algorithm strongly resembles Stochastic Gradient Descent. In fact, Scikit-Learn's `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters: `loss="perceptron"`, `learning_rate="constant"`, `eta0=1` (the learning rate), and `penalty=None` (no regularization).

Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they just make predictions based on a hard threshold. This is one of the good reasons to prefer Logistic Regression over Perceptrons.

In their 1969 monograph titled *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 10-6). Of course this is true of any other linear classification model as well (such as Logistic Regression classifiers), but researchers had expected much more from Perceptrons, and their disappointment was great, and many researchers dropped neural networks altogether in favor of higher-level problems such as logic, problem solving, and search.

However, it turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons. The resulting ANN is called a *Multi-Layer Perceptron* (MLP). In particular, an MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the right of Figure 10-6: with inputs (0, 0) or (1, 1) the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1. All connections have a weight equal to 1, except the four connections where the weight is shown. Try verifying that this network indeed solves the XOR problem!

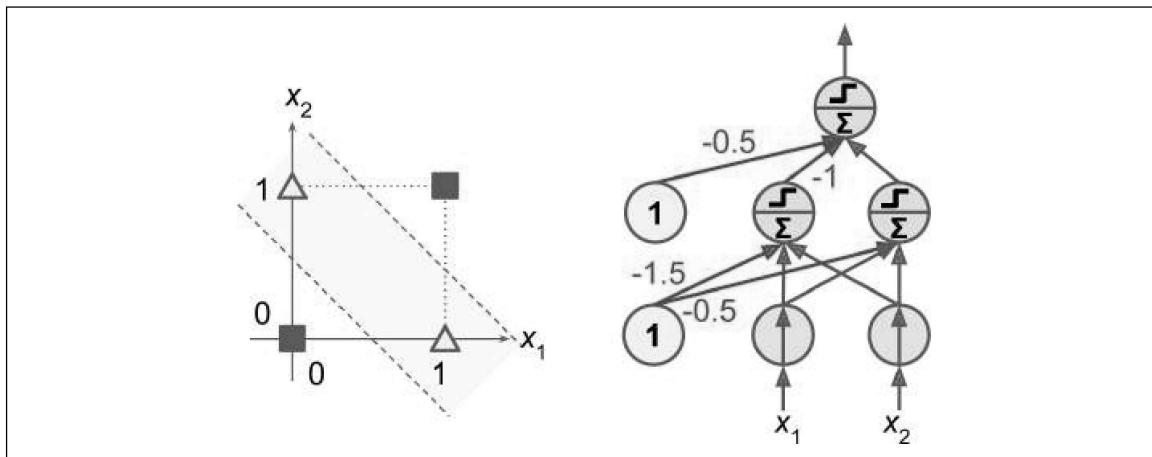


Figure 10-6. XOR classification problem and an MLP that solves it

Multi-Layer Perceptron and Backpropagation

An MLP is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer* (see Figure 10-7). 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. Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

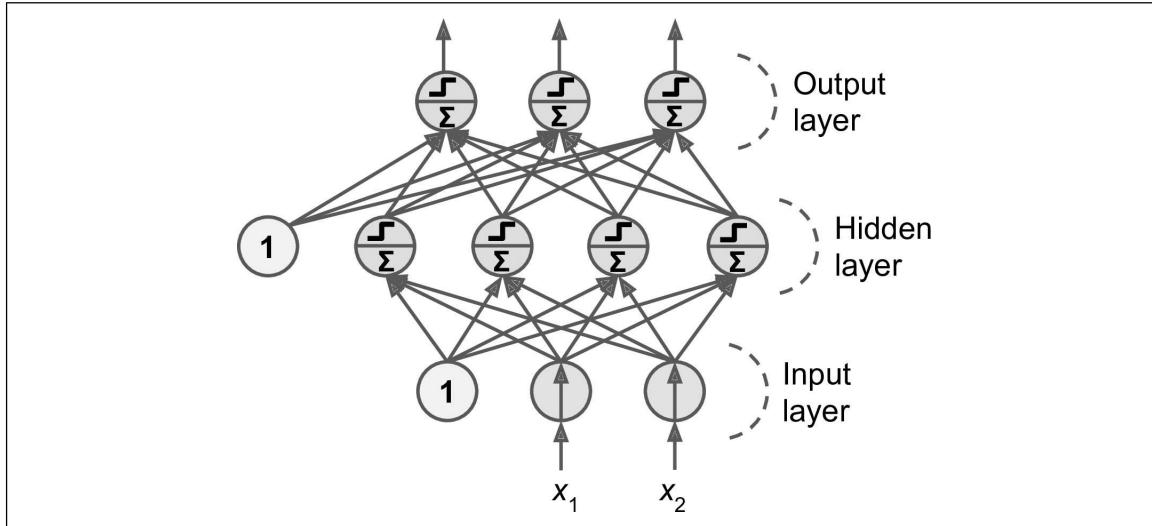


Figure 10-7. Multi-Layer Perceptron



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 layers⁸, it is called a *deep neural network* (DNN). The field of Deep Learning studies DNNs, and more generally models containing deep stacks of computations. However, 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. But in 1986, David Rumelhart, Geoffrey Hinton and Ronald Williams published a groundbreaking paper⁹ introducing the *backpropagation* training algorithm, which is still used today. In short, it is simply Gradient Descent (introduced in Chapter 4)

⁸ 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.

⁹ “Learning Internal Representations by Error Propagation,” D. Rumelhart, G. Hinton, R. Williams (1986).

using an efficient technique for computing the gradients automatically¹⁰: in just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network’s error with regards to every single model parameter. In other words, it can find out how each connection weight and each bias term should be tweaked in order to reduce the error. Once it has these gradients, it just performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.



Automatically computing gradients is called *automatic differentiation*, or *autodiff*. There are various autodiff techniques, with different pros and cons. The one used by backpropagation is called *reverse-mode autodiff*. It is fast and precise, and is well suited when the function to differentiate has many variables (e.g., connection weights) and few outputs (e.g., one loss). If you want to learn more about autodiff, check out ???.

Let’s run through this algorithm in a bit more detail:

- It handles one mini-batch at a time (for example containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an *epoch*, as we saw in Chapter 4.
- Each mini-batch is passed to the network’s input layer, which just sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch). The result is 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 connection contributed to the error. This is done analytically by simply applying the *chain rule* (perhaps the most fundamental rule in calculus), which makes this step fast and precise.
- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule—and so on until the algorithm reaches the input layer. As we explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights in the

¹⁰ This technique was actually independently invented several times by various researchers in different fields, starting with P. Werbos in 1974.

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 in the network, using the error gradients it just computed.

This algorithm is so important, it's worth summarizing it again: for each training instance the backpropagation algorithm first makes a prediction (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally slightly tweaks the connection weights to reduce the error (Gradient Descent step).



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 order for this algorithm to work properly, the authors made a key change to the MLP's architecture: they replaced the step function with the logistic function, $\sigma(z) = 1 / (1 + \exp(-z))$. 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 logistic 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 logistic function. Two other popular activation functions are:

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

Just like the logistic function it 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 logistic function), which tends to make each layer's output more or less centered around 0 at the beginning of training. This often helps speed up convergence.

The Rectified Linear Unit function: $\text{ReLU}(z) = \max(0, z)$

It 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 $z < 0$. However, in practice it works very well and has the advantage of being

fast to compute¹¹. Most importantly, the fact that it does not have a maximum output value also 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 10-8. 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, say $f(x) = 2x + 3$ 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 non-linearity between layers, then even a deep stack of layers is equivalent to a single layer: you cannot solve very complex problems with that.

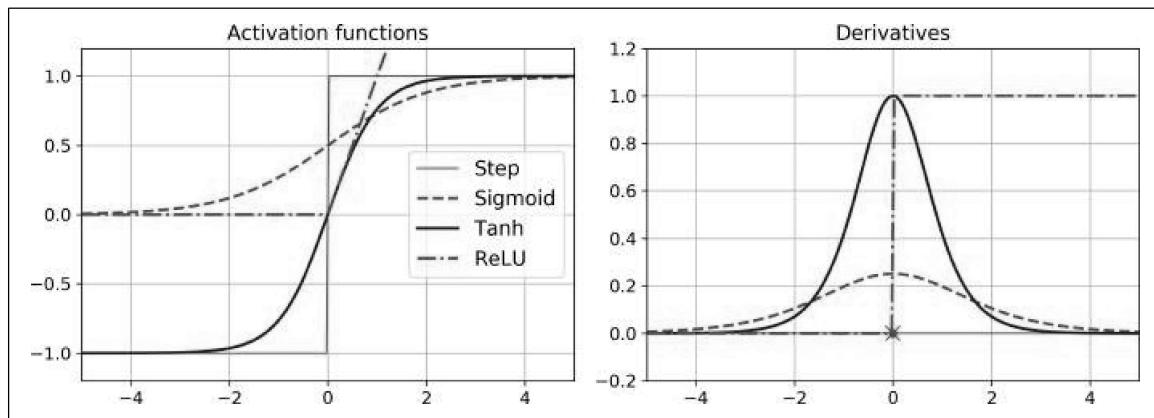


Figure 10-8. Activation functions and their derivatives

Okay! So now you know where neural nets came from, what their architecture is and how to compute their outputs, and you also learned about the backpropagation algorithm. But what exactly can you do with them?

Regression MLPs

First, MLPs can be used for regression tasks. 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 on 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 4 output neurons.

¹¹ 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 misleading.

In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values. However, if you want to guarantee that the output will always be positive, then you can use the ReLU activation function, or the *softplus* activation function in the output layer. Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and scale the labels to the appropriate range: 0 to 1 for the logistic function, or -1 to 1 for the hyperbolic tangent.

The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the Huber loss, which is a combination of both.



The Huber loss is quadratic when the error is smaller than a threshold δ (typically 1), but linear when the error is larger than δ . This makes it less sensitive to outliers than the mean squared error, and it is often more precise and converges faster than the mean absolute error.

Table 10-1 summarizes the typical architecture of a regression MLP.

Table 10-1. Typical Regression MLP Architecture

Hyperparameter	Typical Value
# input neurons	One per input feature (e.g., $28 \times 28 = 784$ for MNIST)
# hidden layers	Depends on the problem. Typically 1 to 5.
# neurons per hidden layer	Depends on the problem. Typically 10 to 100.
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLPs

MLPs can also be used for classification tasks. For a binary classification problem, you just need a single output neuron using the logistic activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. Obviously, 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 non-urgent email. In this case, you would need two output neurons, both using the logistic 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 one. This lets the model output any combination of labels: you can have non-urgent ham, urgent ham, non-urgent 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 3 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 10-9). 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 one (which is required if the classes are exclusive). This is called multiclass classification.

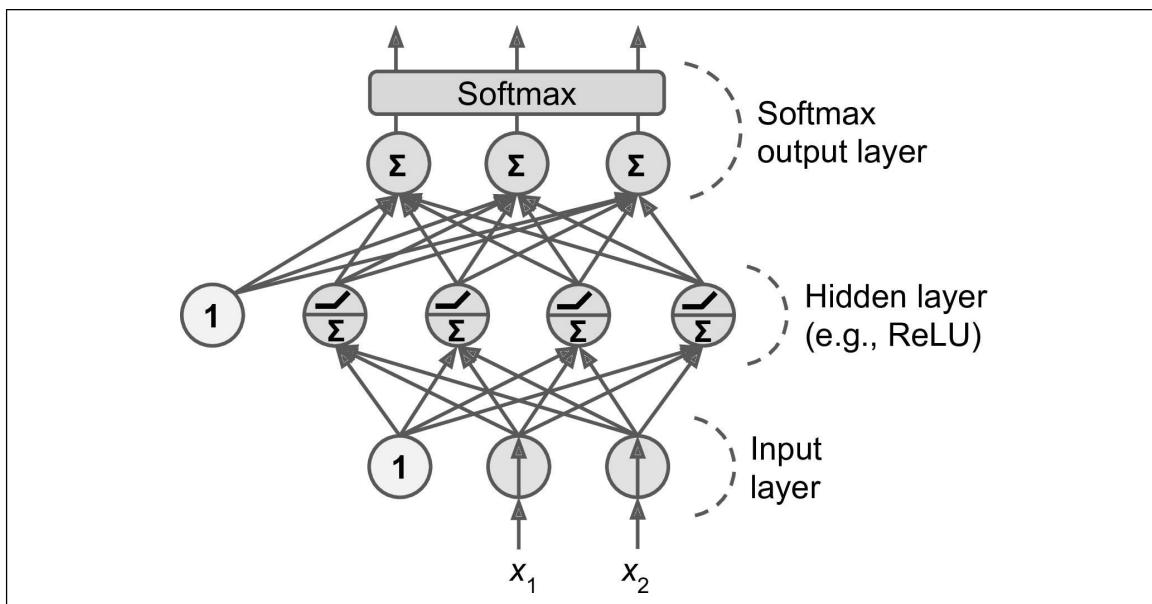


Figure 10-9. A modern MLP (including ReLU and softmax) for classification

Regarding the loss function, since we are predicting probability distributions, the cross-entropy (also called the log loss, see Chapter 4) is generally a good choice.

Table 10-2 summarizes the typical architecture of a classification MLP.

Table 10-2. Typical Classification MLP Architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Loss function	Cross-Entropy	Cross-Entropy	Cross-Entropy



Before we go on, I recommend you go through exercise 1, at the end of this chapter. You will play with various neural network architectures and visualize their outputs using the *TensorFlow Playground*. This will be very useful to better understand MLPs, for example the effects of all the hyperparameters (number of layers and neurons, activation functions, and more).

Now you have all the concepts you need to start implementing MLPs with Keras!

Implementing MLPs with Keras

Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate and execute all sorts of neural networks. Its documentation (or specification) is available at <https://keras.io>. The reference implementation is simply called Keras as well, so to avoid any confusion we will call it keras-team (since it is available at <https://github.com/keras-team/keras>). It was developed by François Chollet as part of a research project¹² and released as an open source project in March 2015. It quickly gained popularity owing to its ease-of-use, flexibility and beautiful design. To perform the heavy computations required by neural networks, keras-team relies on a computation backend. At the present, you can choose from three popular open source deep learning libraries: TensorFlow, Microsoft Cognitive Toolkit (CNTK) or Theano.

Moreover, since late 2016, other implementations have been released. You can now run Keras on Apache MXNet, Apple's Core ML, Javascript or Typescript (to run Keras code in a web browser), or PlaidML (which can run on all sorts of GPU devices, not just Nvidia). Moreover, TensorFlow itself now comes bundled with its own Keras implementation called tf.keras. It only supports TensorFlow as the backend, but it has the advantage of offering some very useful extra features (see Figure 10-10): for example, it supports TensorFlow's Data API which makes it quite easy to load and preprocess data efficiently. For this reason, we will use tf.keras in this book. However, in this chapter we will not use any of the TensorFlow-specific features, so the code should run fine on other Keras implementations as well (at least in Python), with only minor modifications, such as changing the imports.

¹² Project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System).

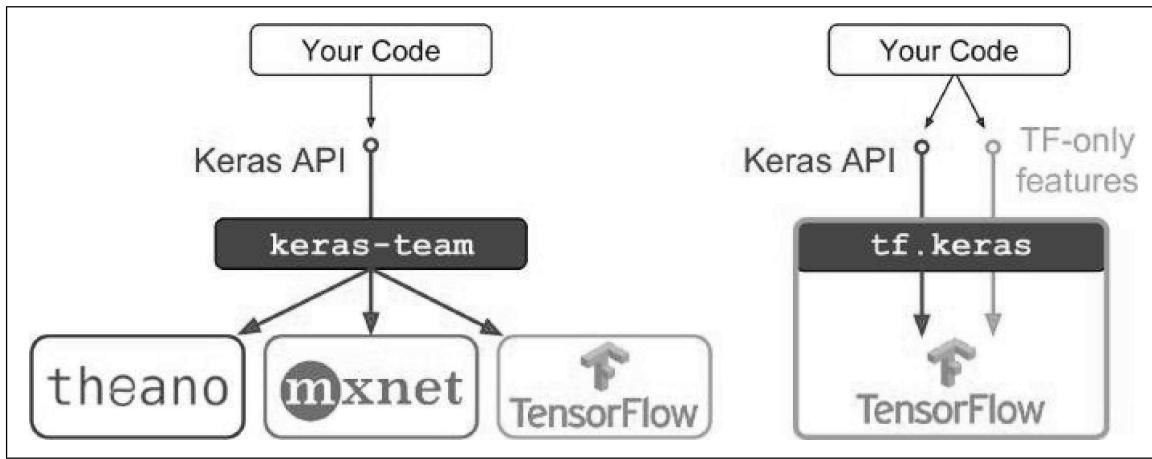


Figure 10-10. Two Keras implementations: keras-team (left) and tf.keras (right)

As tf.keras is bundled with TensorFlow, let's install TensorFlow!

Installing TensorFlow 2

Assuming you installed Jupyter and Scikit-Learn by following the installation instructions in Chapter 2, you can simply use pip to install TensorFlow. If you created an isolated environment using virtualenv, you first need to activate it:

```
$ cd $ML_PATH           # Your ML working directory (e.g., $HOME/ml)
$ source env/bin/activate # on Linux or Mac OSX
$ .\env\Scripts\activate # on Windows
```

Next, install TensorFlow 2 (if you are not using a virtualenv, you will need administrator rights, or to add the --user option):

```
$ python3 -m pip install --upgrade tensorflow
```



For GPU support, you need to install tensorflow-gpu instead of tensorflow, and there are other libraries to install. See <https://tensorflow.org/install/gpu> for more details.

To test your installation, open a Python shell or a Jupyter notebook, then import TensorFlow and tf.keras, and print their versions:

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

The second version is the version of the Keras API implemented by `tf.keras`. Note that it ends with `-tf`, highlighting the fact that `tf.keras` implements the Keras API, plus some extra TensorFlow-specific features.

Now let's use `tf.keras`! Let's start by building a simple image classifier.

Building an Image Classifier Using the Sequential API

First, we need to load a dataset. We will tackle *Fashion MNIST*, which is a drop-in replacement of MNIST (introduced in Chapter 3). It has the exact same format as MNIST (70,000 grayscale images of 28×28 pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits, so each class is 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.

Using Keras to Load the Dataset

Keras provides some utility functions to fetch and load common datasets, including MNIST, Fashion MNIST, the original California housing dataset, and more. Let's load Fashion MNIST:

```
fashion_mnist = keras.datasets.fashion_mnist  
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

When loading MNIST or Fashion MNIST using Keras rather than Scikit-Learn, one important difference is that every image is represented as a 28×28 array rather than a 1D array of size 784. Moreover, the pixel intensities are represented as integers (from 0 to 255) rather than floats (from 0.0 to 255.0). Here is the shape and data type of the training set:

```
>>> X_train_full.shape  
(60000, 28, 28)  
>>> X_train_full.dtype  
dtype('uint8')
```

Note that the dataset is already split into a training set and a test set, but there is no validation set, so let's create one. Moreover, since we are going to train the neural network using Gradient Descent, we must scale the input features. For simplicity, we just scale the pixel intensities down to the 0-1 range by dividing them by 255.0 (this also converts them to floats):

```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0  
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

With MNIST, when the label is equal to 5, it means that the image represents the handwritten digit 5. Easy. However, for Fashion MNIST, we need the list of class names to know what we are dealing with:

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

For example, the first image in the training set represents a coat:

```
>>> class_names[y_train[0]]
'Coat'
```

Figure 10-11 shows a few samples from the Fashion MNIST dataset:



Figure 10-11. Samples from Fashion MNIST

Creating the Model Using the Sequential API

Now let's build the neural network! Here is a classification MLP with two hidden layers:

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

Let's go through this code line by line:

- The first line creates a **Sequential** model. This is the simplest kind of Keras model, for neural networks that are just composed of a single stack of layers, connected sequentially. This is called the sequential API.
- Next, we build the first layer and add it to the model. It is a **Flatten** layer whose role is simply to convert each input image into a 1D array: if it receives input data X , it computes $X.\text{reshape}(-1, 1)$. This layer does not have any parameters, it is just there to do some simple preprocessing. Since it is the first layer in the model, you should specify the **input_shape**: this does not include the batch size, only the shape of the instances. Alternatively, you could add a **keras.layers.InputLayer** as the first layer, setting **shape=[28, 28]**.
- Next we add a **Dense** hidden layer with 300 neurons. It will use the ReLU activation function. Each **Dense** layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vec-

tor of bias terms (one per neuron). When it receives some input data, it computes Equation 10-2.

- Next we add a second `Dense` hidden layer with 100 neurons, also using the ReLU activation function.
- Finally, we add a `Dense` output layer with 10 neurons (one per class), using the softmax activation function (because the classes are exclusive).



Specifying `activation="relu"` is equivalent to `activation=keras.activations.relu`. Other activation functions are available in the `keras.activations` package, we will use many of them in this book. See <https://keras.io/activations/> for the full list.

Instead of adding the layers one by one as we just did, you can pass a list of layers when creating the `Sequential` model:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Using Code Examples From keras.io

Code examples documented on keras.io will work fine with `tf.keras`, but you need to change the imports. For example, consider this keras.io code:

```
from keras.layers import Dense
output_layer = Dense(10)
```

You must change the imports like this:

```
from tensorflow.keras.layers import Dense
output_layer = Dense(10)
```

Or simply use full paths, if you prefer:

```
from tensorflow import keras
output_layer = keras.layers.Dense(10)
```

This is more verbose, but I use this approach in this book so you can easily see which packages to use, and to avoid confusion between standard classes and custom classes. In production code, I use the previous approach, as do most people.

The model's `summary()` method displays all the model's layers¹³, including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parameters. The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters (we will see examples of non-trainable parameters in Chapter 11):

```
>>> model.summary()

Layer (type)          Output Shape         Param #
=====
flatten_1 (Flatten)    (None, 784)           0
=====
dense_3 (Dense)        (None, 300)          235500
=====
dense_4 (Dense)        (None, 100)          30100
=====
dense_5 (Dense)        (None, 10)           1010
=====

Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
```

Note that Dense layers often have a *lot* of parameters. For example, the first hidden layer has 784×300 connection weights, plus 300 bias terms, which adds up to 235,500 parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data. We will come back to this later.

You can easily get a model's list of layers, to fetch a layer by its index, or you can fetch it by name:

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,
 <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,
 <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
>>> model.layers[1].name
'dense_3'
>>> model.get_layer('dense_3').name
'dense_3'
```

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` method. For a Dense layer, this includes both the connection weights and the bias terms:

¹³ You can also generate an image of your model using `keras.utils.plot_model()`.

```

>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.03854964, -0.04054524,  0.00599282, ...,  0.02566582,
       0.01032123,  0.06914985],
       ...,
       [ 0.02632413, -0.05105981, -0.00332005, ...,  0.04175945,
       0.0443138 , -0.05558084]], dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., ..., 0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)

```

Notice that the Dense layer initialized the connection weights randomly (which is needed to break symmetry, as we discussed earlier), and the biases were just initialized to zeros, which is fine. If you ever want to use a different initialization method, you can set `kernel_initializer` (`kernel` is another name for the matrix of connection weights) or `bias_initializer` when creating the layer. We will discuss initializers further in Chapter 11, but if you want the full list, see <https://keras.io/initializers/>.



The shape of the weight matrix depends on the number of inputs. This is why it is recommended to specify the `input_shape` when creating the first layer in a Sequential model. However, if you do not specify the input shape, it's okay: Keras will simply wait until it knows the input shape before it actually builds the model. This will happen either when you feed it actual data (e.g., during training), or when you call its `build()` method. Until the model is really built, the layers will not have any weights, and you will not be able to do certain things (such as print the model summary or save the model), so if you know the input shape when creating the model, it is best to specify it.

Compiling the Model

After a model is created, you must call its `compile()` method to specify the loss function and the optimizer to use. Optionally, you can also specify a list of extra metrics to compute during training and evaluation:

```

model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])

```



Using `loss="sparse_categorical_crossentropy"` is equivalent to `loss=keras.losses.sparse_categorical_crossentropy`. Similarly, `optimizer="sgd"` is equivalent to `optimizer=keras.optimizers.SGD()` and `metrics=["accuracy"]` is equivalent to `metrics=[keras.metrics.sparse_categorical_accuracy]` (when using this loss). We will use many other losses, optimizers and metrics in this book, but for the full lists see <https://keras.io/losses/>, <https://keras.io/optimizers/> and <https://keras.io/metrics/>.

This requires some explanation. First, we use the "`sparse_categorical_crossentropy`" loss because we have sparse labels (i.e., for each instance there is just a target class index, from 0 to 9 in this case), and the classes are exclusive. If instead we had one target probability per class for each instance (such as one-hot vectors, e.g. [0., 0., 0., 1., 0., 0., 0., 0., 0.] to represent class 3), then we would need to use the "`categorical_crossentropy`" loss instead. If we were doing binary classification (with one or more binary labels), then we would use the "`sigmoid`" (i.e., logistic) activation function in the output layer instead of the "`softmax`" activation function, and we would use the "`binary_crossentropy`" loss.



If you want to convert sparse labels (i.e., class indices) to one-hot vector labels, you can use the `keras.utils.to_categorical()` function. To go the other way round, you can just use the `np.argmax()` function with `axis=1`.

Secondly, regarding the optimizer, "`sgd`" simply means that we will train the model using simple Stochastic Gradient Descent. In other words, Keras will perform the backpropagation algorithm described earlier (i.e., reverse-mode autodiff + Gradient Descent). We will discuss more efficient optimizers in Chapter 11 (they improve the Gradient Descent part, not the autodiff).

Finally, since this is a classifier, it's useful to measure its "`accuracy`" during training and evaluation.

Training and Evaluating the Model

Now the model is ready to be trained. For this we simply need to call its `fit()` method. We pass it the input features (`X_train`) and the target classes (`y_train`), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution). We also pass a validation set (this is optional): Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs: 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):

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                      validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [=====] - 3s 55us/sample - loss: 1.4948 - acc: 0.5757
                                         - val_loss: 1.0042 - val_acc: 0.7166
Epoch 2/30
55000/55000 [=====] - 3s 55us/sample - loss: 0.8690 - acc: 0.7318
                                         - val_loss: 0.7549 - val_acc: 0.7616
[...]
Epoch 50/50
55000/55000 [=====] - 4s 72us/sample - loss: 0.3607 - acc: 0.8752
                                         - val_loss: 0.3706 - val_acc: 0.8728
```

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of instances processed so far (along with a progress bar), the mean training time per sample, the loss and accuracy (or any other extra metrics you asked for), both on the training set and the validation set. You can see that the training loss went down, which is a good sign, and the validation accuracy reached 87.28% after 50 epochs, not too far from the training accuracy, so there does not seem to be much overfitting going on.



Instead of passing a validation set using the `validation_data` argument, you could instead set `validation_split` to the ratio of the training set that you want Keras to use for validation (e.g., 0.1).

If the training set was very skewed, with some classes being overrepresented and others underrepresented, it would be useful to set the `class_weight` argument when calling the `fit()` method, giving a larger weight to underrepresented classes, and a lower weight to overrepresented classes. These weights would be used by Keras when computing the loss. If you need per-instance weights instead, you can set the `sample_weight` argument (it supersedes `class_weight`). This could be useful for example if some instances were labeled by experts while others were labeled using a crowdsourcing platform: you might want to give more weight to the former. You can also provide sample weights (but not class weights) for the validation set by adding them as a third item in the `validation_data` tuple.

The `fit()` method returns a `History` object containing the training parameters (`history.params`), the list of epochs it went through (`history.epoch`), and most importantly a dictionary (`history.history`) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if

any). If you create a Pandas DataFrame using this dictionary and call its `plot()` method, you get the learning curves shown in Figure 10-12:

```
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

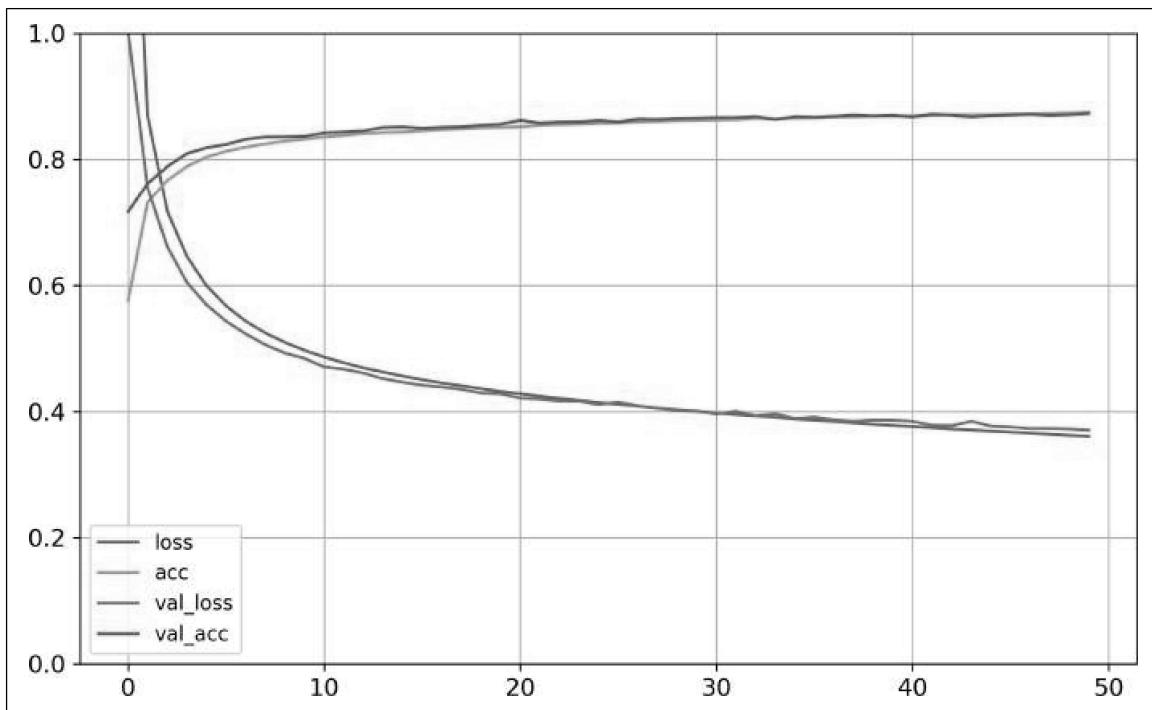


Figure 10-12. Learning Curves

You can see that both the training and validation accuracy steadily increase during training, while the training and validation loss decrease. Good! Moreover, the validation curves are quite close to the training curves, which means that there is not too much overfitting. In this particular case, the model performed better on the validation set than on the training set at the beginning of training: this sometimes happens by chance (especially when the validation set is fairly small). However, the training set performance ends up beating the validation performance, as is generally the case when you train for long enough. You can tell that the model has not quite converged yet, as the validation loss is still going down, so you should probably continue training. It's as simple as calling the `fit()` method again, since Keras just continues training where it left off (you should be able to reach close to 89% validation accuracy).

If you are not satisfied with the performance of your model, you should go back and tune the model's hyperparameters, for example the number of layers, the number of neurons per layer, the types of activation functions we use for each hidden layer, the

number of training epochs, the batch size (it can be set in the `fit()` method using the `batch_size` argument, which defaults to 32). We will get back to hyperparameter tuning at the end of this chapter. Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production. You can easily do this using the `evaluate()` method (it also supports several other arguments, such as `batch_size` or `sample_weight`, please check the documentation for more details):

```
>>> model.evaluate(X_test, y_test)
8832/10000 [=====] - ETA: 0s - loss: 0.4074 - acc: 0.8540
[0.40738476498126985, 0.854]
```

As we saw in Chapter 2, it is common to get slightly lower performance on the test set than on the validation set, because the hyperparameters are tuned on the validation set, not the test set (however, in this example, we did not do any hyperparameter tuning, so the lower accuracy is just bad luck). Remember to resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using the Model to Make Predictions

Next, we can use the model's `predict()` method to make predictions on new instances. Since we don't have actual new instances, we will just use the first 3 instances of the test set:

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.   , 0.   , 0.   , 0.   , 0.   , 0.09, 0.   , 0.12, 0.   , 0.79],
       [0.   , 0.   , 0.94, 0.   , 0.02, 0.   , 0.04, 0.   , 0.   , 0.   ],
       [0.   , 1.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ]],
      dtype=float32)
```

As you can see, for each instance the model estimates one probability per class, from class 0 to class 9. For example, for the first image it estimates that the probability of class 9 (ankle boot) is 79%, the probability of class 7 (sneaker) is 12%, the probability of class 5 (sandal) is 9%, and the other classes are negligible. In other words, it "believes" it's footwear, probably ankle boots, but it's not entirely sure, it might be sneakers or sandals instead. If you only care about the class with the highest estimated probability (even if that probability is quite low) then you can use the `predict_classes()` method instead:

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

And the classifier actually classified all three images correctly:

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

Now you know how to build, train, evaluate and use a classification MLP using the Sequential API. But what about regression?

Building a Regression MLP Using the Sequential API

Let's switch to the California housing problem and tackle it using a regression neural network. 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 is no missing value. After loading the data, we split it into a training set, a validation set and a test set, and we scale all the features:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)
X_test_scaled = scaler.transform(X_test)
```

Building, training, evaluating and using a regression MLP using the Sequential API to make predictions is quite similar to what we did for classification. The main differences are the fact that the output layer has a single neuron (since we only want to predict a single value) and uses no activation function, and the loss function is the mean squared error. Since the dataset is quite noisy, we just use a single hidden layer with fewer neurons than before, to avoid overfitting:

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                     validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3] # pretend these are new instances
y_pred = model.predict(X_new)
```

As you can see, the Sequential API is quite easy to use. However, although sequential models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the Functional API.

Building Complex Models Using the Functional API

One example of a non-sequential neural network is a *Wide & Deep* neural network. This neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al.¹⁴. It connects all or part of the inputs directly to the output layer, as shown in Figure 10-13. 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). 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.

¹⁴ “Wide & Deep Learning for Recommender Systems,” Heng-Tze Cheng et al. (2016).

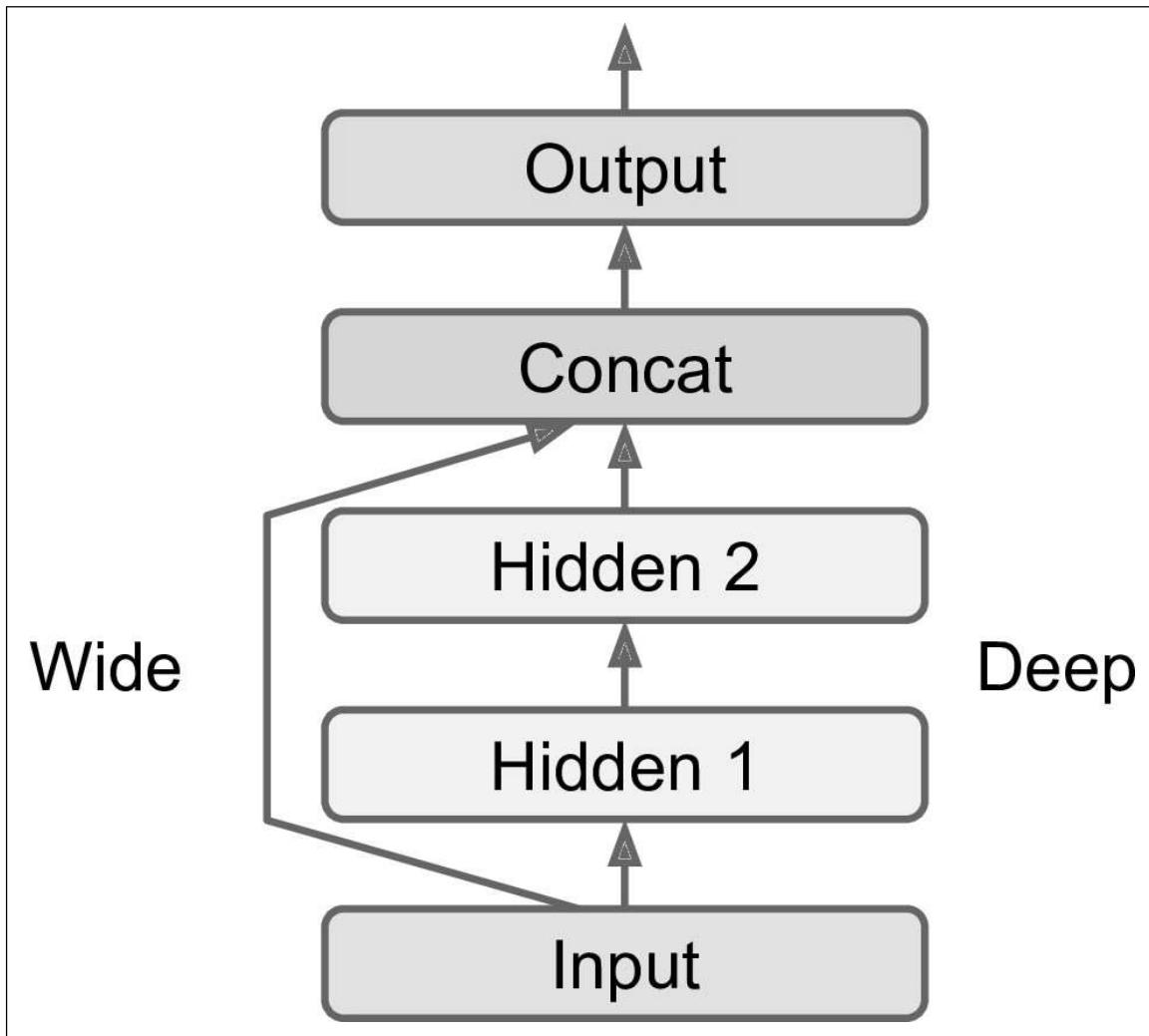


Figure 10-13. Wide and Deep Neural Network

Let's build such a neural network to tackle the California housing problem:

```

input = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input], outputs=[output])

```

Let's go through each line of this code:

- First, we need to create an `Input` object. This is needed because we may have multiple inputs, as we will see later.
- Next, we create a `Dense` layer with 30 neurons and using the ReLU activation function. As soon as it is created, notice that we call it like a function, passing it the input. This is why this is called the Functional API. Note that we are just tell-

ing Keras how it should connect the layers together, no actual data is being processed yet.

- We then create a second hidden layer, and again we use it as a function. Note however that we pass it the output of the first hidden layer.
- Next, we create a `Concatenate()` layer, and once again we immediately use it like a function, to concatenate the input and the output of the second hidden layer (you may prefer the `keras.layers.concatenate()` function, which creates a `Concatenate` layer and immediately calls it with the given inputs).
- Then we create the output layer, with a single neuron and no activation function, and we call it like a function, passing it the result of the concatenation.
- Lastly, we create a Keras `Model`, specifying which inputs and outputs to use.

Once you have built the Keras model, everything is exactly like earlier, so no need to repeat it here: you must compile the model, train it, evaluate it and use it to make predictions.

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 (see Figure 10-14)? In this case, one solution is to use multiple inputs. For example, suppose we want to send 5 features through the deep path (features 0 to 4), and 6 features through the wide path (features 2 to 7):

```
input_A = keras.layers.Input(shape=[5])
input_B = keras.layers.Input(shape=[6])
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input_A, input_B], outputs=[output])
```

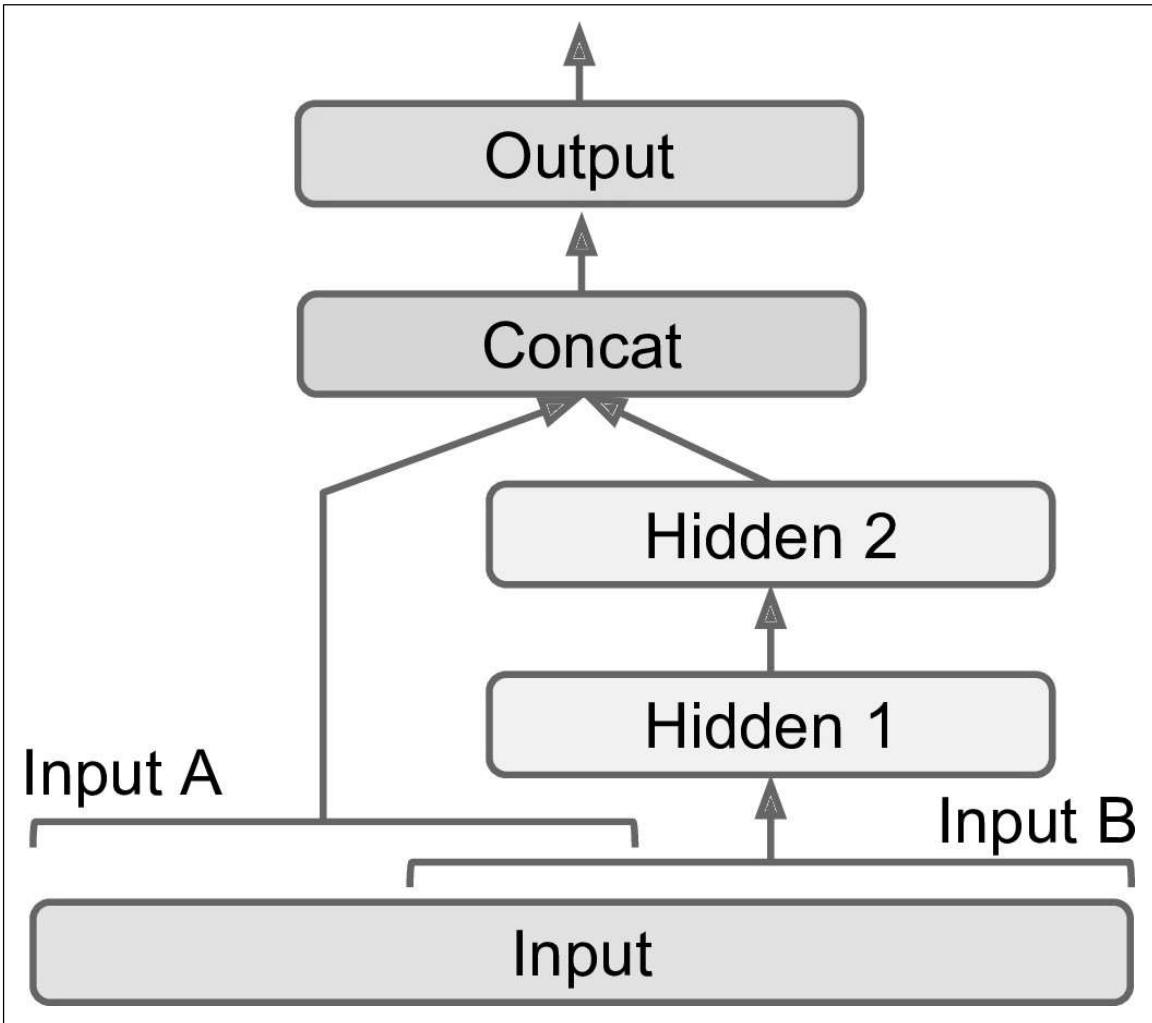


Figure 10-14. Handling Multiple Inputs

The code is self-explanatory. Note that we specified `inputs=[input_A, input_B]` when creating the model. Now we can compile the model as usual, but when we call the `fit()` method, instead of passing a single input matrix `X_train`, we must pass a pair of matrices (`X_train_A, X_train_B`): one per input. The same is true for `X_valid`, and also for `X_test` and `X_new` when you call `evaluate()` or `predict()`:

```

model.compile(loss="mse", optimizer="sgd")

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                     validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))

```

There are also many use cases in which you may want to have multiple outputs:

- The task may demand it, for example you may want to locate and classify the main object in a picture. This is both a regression task (finding the coordinates of the object's center, as well as its width and height) and a classification task.
- Similarly, you may have multiple independent tasks to perform 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.
- Another use case is as a regularization technique (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 some auxiliary outputs in a neural network architecture (see Figure 10-15) to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.

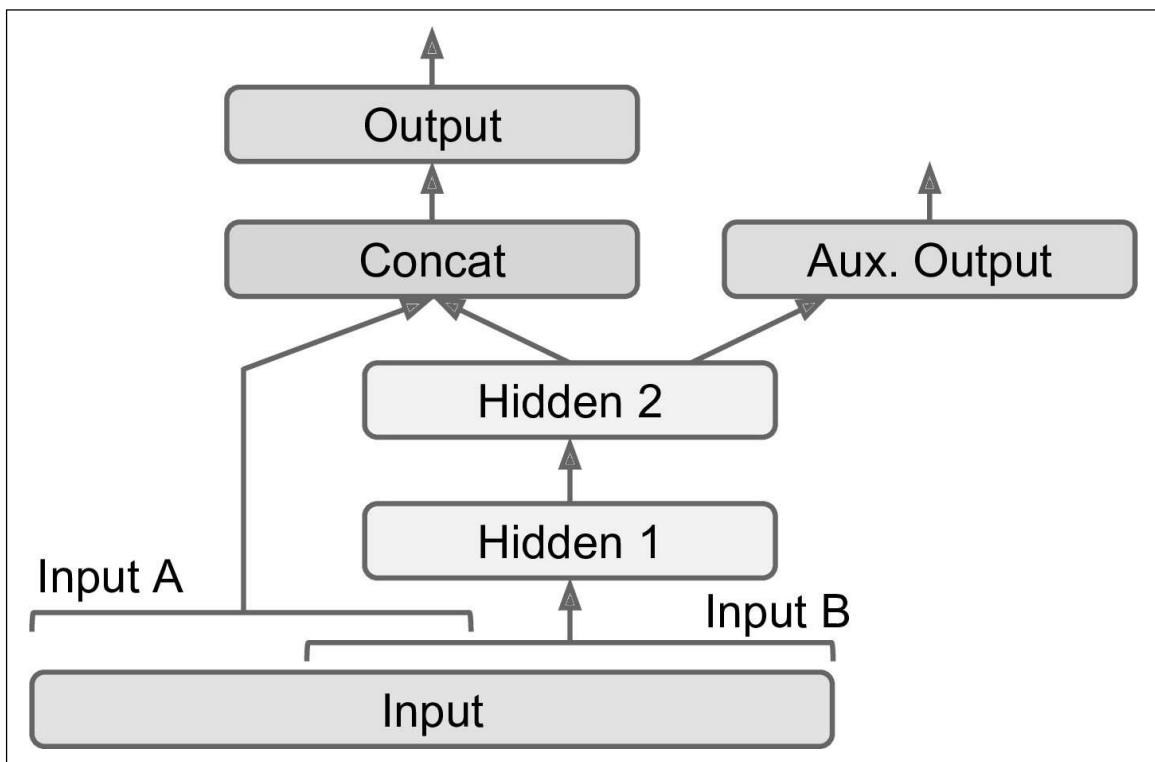


Figure 10-15. Handling Multiple Outputs – Auxiliary Output for Regularization

Adding extra outputs is quite easy: just connect them to the appropriate layers and add them to your model's list of outputs. For example, the following code builds the network represented in Figure 10-15:

```
[...] # Same as above, up to the main output layer
output = keras.layers.Dense(1)(concat)
aux_output = keras.layers.Dense(1)(hidden2)
model = keras.models.Model(inputs=[input_A, input_B],
                            outputs=[output, aux_output])
```

Each output will need its own loss function, so when we compile the model we should pass a list of losses (if we pass a single loss, Keras will assume that the same loss must be used for all outputs). By default, Keras will compute all these losses and simply add them up to get the final loss used for training. However, we care much more about the main output than about the auxiliary output (as it is just used for regularization), so we want to give the main output's loss a much greater weight. Fortunately, it is possible to set all the loss weights when compiling the model:

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

Now when we train the model, we need to provide some labels for each output. In this example, the main output and the auxiliary output should try to predict the same thing, so they should use the same labels. So instead of passing `y_train`, we just need to pass `(y_train, y_train)` (and the same goes for `y_valid` and `y_test`):

```
history = model.fit(
    [X_train_A, X_train_B], [y_train, y_train], epochs=20,
    validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

When we evaluate the model, Keras will return the total loss, as well as all the individual losses:

```
total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])
```

Similarly, the `predict()` method will return predictions for each output:

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

As you can see, you can build any sort of architecture you want quite easily with the Functional API. Let's look at one last way you can build Keras models.

Building Dynamic Models Using the Subclassing API

Both the Sequential API and the Functional API are declarative: you start by declaring which layers you want to use and how they should be connected, and only then can you start feeding the model some data for training or inference. This has many advantages: the model can easily be saved, cloned, shared, its structure can be displayed and analyzed, the framework can infer shapes and check types, so errors can be caught early (i.e., before any data ever goes through the model). It's also fairly easy to debug, since the whole model is just a static graph of layers. But the flip side is just that: it's static. Some models involve loops, varying shapes, conditional branching, and other dynamic behaviors. For such cases, or simply if you prefer a more imperative programming style, the Subclassing API is for you.

Simply subclass the `Model` class, create the layers you need in the constructor, and use them to perform the computations you want in the `call()` method. For example, creating an instance of the following `WideAndDeepModel` class gives us an equivalent model to the one we just built with the Functional API. You can then compile it, evaluate it and use it to make predictions, exactly like we just did.

```
class WideAndDeepModel(keras.models.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # handles standard args (e.g., name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```

This example looks very much like the Functional API, except we do not need to create the inputs, we just use the `input` argument to the `call()` method, and we separate the creation of the layers¹⁵ in the constructor from their usage in the `call()` method. However, the big difference is that you can do pretty much anything you want in the `call()` method: `for` loops, `if` statements, low-level TensorFlow operations, your imagination is the limit (see Chapter 12)! This makes it a great API for researchers experimenting with new ideas.

However, this extra flexibility comes at a cost: your model's architecture is hidden within the `call()` method, so Keras cannot easily inspect it, it cannot save or clone it, and when you call the `summary()` method, you only get a list of layers, without any information on how they are connected to each other. Moreover, Keras cannot check types and shapes ahead of time, and it is easier to make mistakes. So unless you really need that extra flexibility, you should probably stick to the Sequential API or the Functional API.

¹⁵ Keras models have an `output` attribute, so we cannot use that name for the main output layer, which is why we renamed it to `main_output`.



Keras models can be used just like regular layers, so you can easily compose them to build complex architectures.

Now that you know how to build and train neural nets using Keras, you will want to save them!

Saving and Restoring a Model

Saving a trained Keras model is as simple as it gets:

```
model.save("my_keras_model.h5")
```

Keras will save both the model's architecture (including every layer's hyperparameters) and the value of all the model parameters for every layer (e.g., connection weights and biases), using the HDF5 format. It also saves the optimizer (including its hyperparameters and any state it may have).

You will typically have a script that trains a model and saves it, and one or more scripts (or web services) that load the model and use it to make predictions. Loading the model is just as easy:

```
model = keras.models.load_model("my_keras_model.h5")
```



This will work when using the Sequential API or the Functional API, but unfortunately not when using Model subclassing. However, you can use `save_weights()` and `load_weights()` to at least save and restore the model parameters (but you will need to save and restore everything else yourself).

But what if training lasts several hours? This is quite common, especially when training on large datasets. In this case, you should not only save your model at the end of training, but also save checkpoints at regular intervals during training. But how can you tell the `fit()` method to save checkpoints? The answer is: using callbacks.

Using Callbacks

The `fit()` method accepts a `callbacks` argument that lets you specify a list of objects that Keras will call during training at the start and end of training, at the start and end of each epoch and even before and after processing each batch. For example, the `ModelCheckpoint` callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:

```
[...] # build and compile the model  
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")  
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

Moreover, if you use a validation set during training, you can set `save_best_only=True` when creating the `ModelCheckpoint`. In this case, it will only save your model when its performance on the validation set is the best so far. This way, you do not need to worry about training for too long and overfitting the training set: simply restore the last model saved after training, and this will be the best model on the validation set. This is a simple way to implement early stopping (introduced in Chapter 4):

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",  
                                               save_best_only=True)  
history = model.fit(X_train, y_train, epochs=10,  
                     validation_data=(X_valid, y_valid),  
                     callbacks=[checkpoint_cb])  
model = keras.models.load_model("my_keras_model.h5") # rollback to best model
```

Another way to implement early stopping is to simply use the `EarlyStopping` callback. It will interrupt training when it measures no progress on the validation set for a number of epochs (defined by the `patience` argument), and it will optionally roll back to the best model. You can combine both callbacks to both save checkpoints of your model (in case your computer crashes), and actually interrupt training early when there is no more progress (to avoid wasting time and resources):

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,  
                                                 restore_best_weights=True)  
history = model.fit(X_train, y_train, epochs=100,  
                     validation_data=(X_valid, y_valid),  
                     callbacks=[checkpoint_cb, early_stopping_cb])
```

The number of epochs can be set to a large value since training will stop automatically when there is no more progress. Moreover, there is no need to restore the best model saved in this case since the `EarlyStopping` callback will keep track of the best weights and restore them for us at the end of training.



There are many other callbacks available in the `keras.callbacks` package. See <https://keras.io/callbacks/>.

If you need extra control, you can easily write your own custom callbacks. For example, the following custom callback will display the ratio between the validation loss and the training loss during training (e.g., to detect overfitting):

```

class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nval/train: {:.2f}".format(logs["val_loss"] / logs["loss"]))

```

As you might expect, you can implement `on_train_begin()`, `on_train_end()`, `on_epoch_begin()`, `on_epoch_end()`, `on_batch_end()` and `on_batch_end()`. Moreover, callbacks can also be used during evaluation and predictions, should you ever need them (e.g., for debugging). In this case, you should implement `on_test_begin()`, `on_test_end()`, `on_test_batch_begin()`, or `on_test_batch_end()` (called by `evaluate()`), or `on_predict_begin()`, `on_predict_end()`, `on_predict_batch_begin()`, or `on_predict_batch_end()` (called by `predict()`).

Now let's take a look at one more tool you should definitely have in your toolbox when using `tf.keras`: TensorBoard.

Visualization Using TensorBoard

TensorBoard is a great interactive visualization tool that you can use to view the learning curves during training, compare learning curves between multiple runs, visualize the computation graph, analyze training statistics, view images generated by your model, visualize complex multidimensional data projected down to 3D and automatically clustered for you, and more! This tool is installed automatically when you install TensorFlow, so you already have it!

To use it, you must modify your program so that it outputs the data you want to visualize to special binary log files called *event files*. Each binary data record is called a *summary*. The TensorBoard server will monitor the log directory, and it will automatically pick up the changes and update the visualizations: this allows you to visualize live data (with a short delay), such as the learning curves during training. In general, you want to point the TensorBoard server to a root log directory, and configure your program so that it writes to a different subdirectory every time it runs. This way, the same TensorBoard server instance will allow you to visualize and compare data from multiple runs of your program, without getting everything mixed up.

So let's start by defining the root log directory we will use for our TensorBoard logs, plus a small function that will generate a subdirectory path based on the current date and time, so that it is different at every run. You may want to include extra information in the log directory name, such as hyperparameter values that you are testing, to make it easier to know what you are looking at in TensorBoard:

```

root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

```

```
run_logdir = get_run_logdir() # e.g., './my_logs/run_2019_01_16-11_28_43'
```

Next, the good news is that Keras provides a nice `TensorBoard` callback:

```
[...] # Build and compile your model
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                     validation_data=(X_valid, y_valid),
                     callbacks=[tensorboard_cb])
```

And that's all there is to it! It could hardly be easier to use. If you run this code, the `TensorBoard` callback will take care of creating the log directory for you (along with its parent directories if needed), and during training it will create event files and write summaries to them. After running the program a second time (perhaps changing some hyperparameter value), you will end up with a directory structure similar to this one:

```
my_logs
└── run_2019_01_16-16_51_02
    └── events.out.tfevents.1547628669.mycomputer.local.v2
└── run_2019_01_16-16_56_50
    └── events.out.tfevents.1547629020.mycomputer.local.v2
```

Next you need to start the `TensorBoard` server. If you installed TensorFlow within a `virtualenv`, you should activate it. Next, run the following command at the root of the project (or from anywhere else as long as you point to the appropriate log directory). If your shell cannot find the `tensorboard` script, then you must update your `PATH` environment variable so that it contains the directory in which the script was installed (alternatively, you can just replace `tensorboard` with `python3 -m tensorboard.main`).

```
$ tensorboard --logdir=./my_logs --port=6006
TensorBoard 2.0.0 at http://mycomputer.local:6006 (Press CTRL+C to quit)
```

Finally, open up a web browser to `http://localhost:6006`. You should see `TensorBoard`'s web interface. Click on the `SCALARS` tab to view the learning curves (see Figure 10-16). Notice that the training loss went down nicely during both runs, but the second run went down much faster. Indeed, we used a larger learning rate by setting `optimizer=keras.optimizers.SGD(lr=0.05)` instead of `optimizer="sgd"`, which defaults to a learning rate of 0.001.

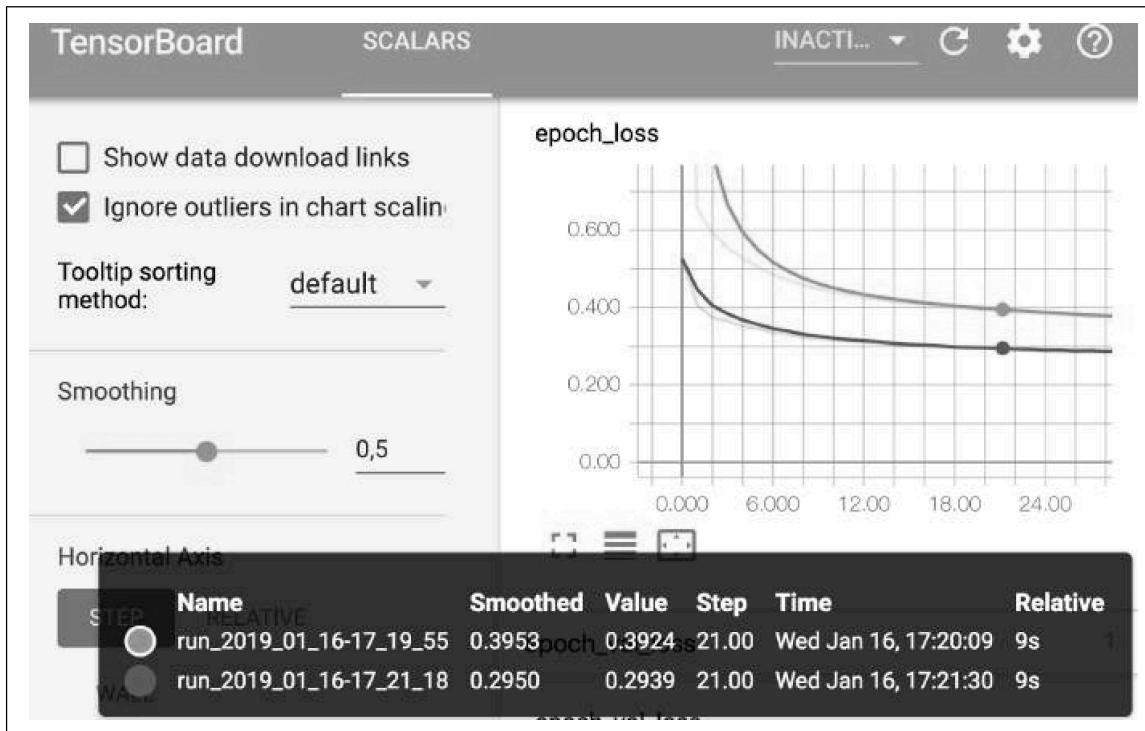


Figure 10-16. Visualizing Learning Curves with TensorBoard

Unfortunately, at the time of writing, no other data is exported by the `TensorBoard` callback, but this issue will probably be fixed by the time you read these lines. In TensorFlow 1, this callback exported the computation graph and many useful statistics: type `help(keras.callbacks.TensorBoard)` to see all the options.

Let's summarize what you learned so far in this chapter: we saw where neural nets came from, what an MLP is and how you can use it for classification and regression, how to build MLPs using `tf.keras`'s Sequential API, or more complex architectures using the Functional API or Model Subclassing, you learned how to save and restore a model, use callbacks for checkpointing, early stopping, and more, and finally how to use TensorBoard for visualization. You can already go ahead and use neural networks to tackle many problems! However, you may wonder how to choose the number of hidden layers, the number of neurons in the network, and all the other hyperparameters. Let's look at this now.

Fine-Tuning Neural Network Hyperparameters

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 simple MLP you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initia-

alization logic, and much more. How do you know what combination of hyperparameters is the best for your task?

One option is to simply try many combinations of hyperparameters and see which one works best on the validation set (or using K-fold cross-validation). For this, one approach is simply use `GridSearchCV` or `RandomizedSearchCV` to explore the hyperparameter space, as we did in Chapter 2. For this, we need to wrap our Keras models in objects that mimic regular Scikit-Learn regressors. The first step is to create a function that will build and compile a Keras model, given a set of hyperparameters:

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):  
    model = keras.models.Sequential()  
    options = {"input_shape": input_shape}  
    for layer in range(n_hidden):  
        model.add(keras.layers.Dense(n_neurons, activation="relu", **options))  
        options = {}  
    model.add(keras.layers.Dense(1, **options))  
    optimizer = keras.optimizers.SGD(learning_rate)  
    model.compile(loss="mse", optimizer=optimizer)  
    return model
```

This function creates a simple `Sequential` model for univariate regression (only one output neuron), with the given input shape and the given number of hidden layers and neurons, and it compiles it using an `SGD` optimizer configured with the given learning rate. The `options` dict is used to ensure that the first layer is properly given the input shape (note that if `n_hidden=0`, the first layer will be the output layer). It is good practice to provide reasonable defaults to as many hyperparameters as you can, as Scikit-Learn does.

Next, let's create a `KerasRegressor` based on this `build_model()` function:

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

The `KerasRegressor` object is a thin wrapper around the Keras model built using `build_model()`. Since we did not specify any hyperparameter when creating it, it will just use the default hyperparameters we defined in `build_model()`. Now we can use this object like a regular Scikit-Learn regressor: we can train it using its `fit()` method, then evaluate it using its `score()` method, and use it to make predictions using its `predict()` method. Note that any extra parameter you pass to the `fit()` method will simply get passed to the underlying Keras model. Also note that the score will be the opposite of the MSE because Scikit-Learn wants scores, not losses (i.e., higher should be better).

```
keras_reg.fit(X_train, y_train, epochs=100,  
              validation_data=(X_valid, y_valid),  
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])  
mse_test = keras_reg.score(X_test, y_test)  
y_pred = keras_reg.predict(X_new)
```

However, we do not actually want to train and evaluate a single model like this, we want to train hundreds of variants and see which one performs best on the validation set. Since there are many hyperparameters, it is preferable to use a randomized search rather than grid search (as we discussed in Chapter 2). Let's try to explore the number of hidden layers, the number of neurons and the learning rate:

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distributions = {
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distributions, n_iter=10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

As you can see, this is identical to what we did in Chapter 2, with the exception that we pass extra parameters to the `fit()` method: they simply get relayed to the underlying Keras models. Note that `RandomizedSearchCV` uses K-fold cross-validation, so it does not use `X_valid` and `y_valid`. These are just used for early stopping.

The exploration may last many hours depending on the hardware, the size of the dataset, the complexity of the model and the value of `n_iter` and `cv`. When it is over, you can access the best parameters found, the best score, and the trained Keras model like this:

```
>>> rnd_search_cv.best_params_
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 42}
>>> rnd_search_cv.best_score_
-0.3189529188278931
>>> model = rnd_search_cv.best_estimator_.model
```

You can now save this model, evaluate it on the test set, and if you are satisfied with its performance, deploy it to production. Using randomized search is not too hard, and it works well for many fairly simple problems. However, when training is slow (e.g., for more complex problems with larger datasets), this approach will only explore a tiny portion of the hyperparameter space. You can partially alleviate this problem by assisting the search process manually: first run a quick random search using wide ranges of hyperparameter values, then run another search using smaller ranges of values centered on the best ones found during the first run, and so on. This will hopefully zoom in to a good set of hyperparameters. However, this is very time consuming, and probably not the best use of your time.

Fortunately, there are many techniques to explore a search space much more efficiently than randomly. Their core idea is simple: when a region of the space turns out

to be good, it should be explored more. This takes care of the “zooming” process for you and leads to much better solutions in much less time. Here are a few Python libraries you can use to optimize hyperparameters:

- Hyperopt: a popular Python library for optimizing over all sorts of complex search spaces (including real values such as the learning rate, or discrete values such as the number of layers).
- Hyperas, kopt or Talos: optimizing hyperparameters for Keras model (the first two are based on Hyperopt).
- Scikit-Optimize (skopt): a general-purpose optimization library. The `Bayes SearchCV` class performs Bayesian optimization using an interface similar to `Grid SearchCV`.
- Spearmint: a Bayesian optimization library.
- Sklearn-Deep: a hyperparameter optimization library based on evolutionary algorithms, also with a `GridSearchCV`-like interface.
- And many more!

Moreover, many companies offer services for hyperparameter optimization. For example Google Cloud ML Engine has a hyperparameter tuning service. Other companies provide APIs for hyperparameter optimization, such as Arimo, SigOpt, Oscar and many more.

Hyperparameter tuning is still an active area of research. Evolutionary algorithms are making a comeback lately. For example, check out DeepMind’s excellent 2017 paper¹⁶, where they jointly optimize a population of models and their hyperparameters. Google also used an evolutionary approach, not just to search for hyperparameters, but also to look for the best neural network architecture for the problem. They call this *AutoML*, and it is already available as a cloud service. Perhaps the days of building neural networks manually will soon be over? Check out Google’s post on this topic. In fact, evolutionary algorithms have also been used successfully to train individual neural networks, replacing the ubiquitous Gradient Descent! See this 2017 post by Uber where they introduce their *Deep Neuroevolution* technique.

Despite all this exciting progress, and all these tools and services, it still helps to have an idea of what values are reasonable for each hyperparameter, so you can build a quick prototype, and restrict the search space. Here are a few guidelines for choosing the number of hidden layers and neurons in an MLP, and selecting good values for some of the main hyperparameters.

¹⁶ “Population Based Training of Neural Networks,” Max Jaderberg et al. (2017).

Number of Hidden Layers

For many problems, you can just begin with a single hidden layer and you will get reasonable results. It has actually been shown that an MLP with just one hidden layer can model even the most complex functions provided it has enough neurons. For a long time, these facts convinced researchers that there was no need to investigate any deeper neural networks. But they overlooked the fact that 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.

To understand why, suppose you are asked to draw a forest using some drawing software, but you are forbidden to use copy/paste. You would have to draw each tree individually, branch per branch, leaf per leaf. If you could instead draw one leaf, copy/paste it to draw a branch, then copy/paste that branch to create a tree, and finally copy/paste this tree to make a forest, you would be finished in no time. Real-world data is often structured in such a hierarchical way and Deep Neural Networks automatically take advantage of this fact: lower hidden layers model low-level structures (e.g., line segments of various shapes and orientations), intermediate hidden layers combine these low-level structures to model intermediate-level structures (e.g., squares, circles), and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g., faces).

Not only does this hierarchical architecture help DNNs converge faster to a good solution, 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, then you can kickstart 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 value 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 it will work just fine (e.g., 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 amount of neurons, in roughly the same amount of training time). For more complex problems, you can gradually 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 we will see in Chapter 14), and they need a huge amount of training data. However, 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 be a lot faster and require much less data (we will discuss this in Chapter 11).

Number of Neurons per Hidden Layer

Obviously 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 $28 \times 28 = 784$ input neurons and 10 output neurons.

As for the hidden layers, it used to be a common practice 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. For example, a typical neural network for MNIST may have three hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned now, as it seems that simply using the same number of neurons in all hidden layers performs just as well in most cases, or even better, and there is just one hyperparameter to tune instead of one per layer—for example, all hidden layers could simply have 150 neurons. However, depending on the dataset, it can sometimes help to make the first hidden layer bigger than the others.

Just like for the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. In general you will get more bang for the buck by increasing the number of layers than the number of neurons per layer. Unfortunately, as you can see, finding the perfect amount of neurons is still somewhat of a dark art.

A simpler approach is to pick a model with more layers and neurons than you actually need, then use early stopping to prevent it from overfitting (and other regularization techniques, such as *dropout*, as we will see in Chapter 11). This has been dubbed the “stretch pants” approach:¹⁷ 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.

Learning Rate, Batch Size and Other Hyperparameters

The number of hidden layers and neurons are not the only hyperparameters you can tweak in an MLP. Here are some of the most important ones, and some tips on how to set them:

- The learning rate is arguably the most important hyperparameter. In general, the optimal learning rate is about half of the maximum learning rate (i.e., the learn-

¹⁷ By Vincent Vanhoucke in his Deep Learning class on Udacity.com.

ing rate above which the training algorithm diverges, as we saw in Chapter 4). So a simple approach for tuning the learning rate is to start with a large value that makes the training algorithm diverge, then divide this value by 3 and try again, and repeat until the training algorithm stops diverging. At that point, you generally won't be too far from the optimal learning rate. That said, it is sometimes useful to reduce the learning rate during training: we will discuss this in Chapter 11.

- Choosing a better optimizer than plain old Mini-batch Gradient Descent (and tuning its hyperparameters) is also quite important. We will discuss this in Chapter 11.
- The batch size can also have a significant impact on your model's performance and the training time. In general the optimal batch size will be lower than 32 (in April 2018, Yann Lecun even tweeted "*Friends don't let friends use mini-batches larger than 32*"). A small batch size ensures that each training iteration is very fast, and although a large batch size will give a more precise estimate of the gradients, in practice this does not matter much since the optimization landscape is quite complex and the direction of the true gradients do not point precisely in the direction of the optimum. However, having a batch size greater than 10 helps take advantage of hardware and software optimizations, in particular for matrix multiplications, so it will speed up training. Moreover, if you use *Batch Normalization* (see Chapter 11), the batch size should not be too small (in general no less than 20).
- We discussed the choice of the activation function earlier in this chapter: in general, the ReLU activation function will be a good default for all hidden layers. For the output layer, it really depends on your task.
- In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

For more best practices, make sure to read Yoshua Bengio's great 2012 paper¹⁸, which presents many practical recommendations for deep networks.

This concludes this introduction to artificial neural networks and their implementation with Keras. In the next few chapters, we will discuss techniques to train very deep nets, we will see how to customize your models using TensorFlow's lower-level API and how to load and preprocess data efficiently using the Data API, and we will dive into other popular neural network architectures: convolutional neural networks for image processing, recurrent neural networks for sequential data, autoencoders for

¹⁸ "Practical recommendations for gradient-based training of deep architectures," Yoshua Bengio (2012).

representation learning, and generative adversarial networks to model and generate data.¹⁹

Exercises

1. Visit the TensorFlow Playground at <https://playground.tensorflow.org/>
 - Layers and patterns: 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. Notice that 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, the more complex the patterns can be.
 - Activation function: try replacing the Tanh activation function with the 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.
 - 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.
 - Too small: now remove one neuron to keep just 2. 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 it systematically underfits the training set.
 - Large enough: next, set the number of neurons to 8 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 almost never get stuck in local minima, and even when they do these local optima are almost as good as the global optimum. However, they can still get stuck on long plateaus for a long time.
 - Deep net and vanishing gradients: now change the dataset to be the spiral (bottom right dataset under “DATA”). Change the network architecture to have 4 hidden layers with 8 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 (i.e. on the right) tend to evolve faster than the neurons in the lowest layers (i.e. on the left). This problem, called the “vanishing gradients” problem, can be alleviated using better weight initialization and

¹⁹ A few extra ANN architectures are presented in ???.

other techniques, better optimizers (such as AdaGrad or Adam), or using Batch Normalization.

- More: go ahead and play with the other parameters to get a feel of what they do. In fact, you should definitely play with this UI for at least one hour, it will grow your intuitions about neural networks significantly.
2. Draw an ANN using the original artificial neurons (like the ones in Figure 10-3) 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 classical 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 logistic 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.
 - What is the shape of the input matrix \mathbf{X} ?
 - What about the shape of the hidden layer's weight vector \mathbf{W}_h , and the shape of its bias vector \mathbf{b}_h ?
 - What is the shape of the output layer's weight vector \mathbf{W}_o , and its bias vector \mathbf{b}_o ?
 - What is the shape of the network's output matrix \mathbf{Y} ?
 - 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, using what activation function? Answer the same questions 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 an 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 MNIST dataset and see if you can get over 98% precision. Try adding all the bells and whistles (i.e., save checkpoints, use early stopping, plot learning curves using TensorBoard, and so on).

Solutions to these exercises are available in ???.

Training Deep Neural Networks



With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 11 in the final release of the book.

In Chapter 10 we introduced artificial neural networks and trained our first deep neural networks. But they were very shallow nets, with just a few hidden layers. What if you need to tackle a very complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper DNN, perhaps with 10 layers or much more, each containing hundreds of neurons, connected by hundreds of thousands of connections. This would not be a walk in the park:

- First, you would be faced with the tricky *vanishing gradients* problem (or the related *exploding gradients* problem) that affects deep neural networks and makes lower layers very hard to train.
- Second, you might not have enough training data for such a large network, or it might be too costly to label.
- Third, training may be extremely slow.
- Fourth, a model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances, or they are too noisy.

In this chapter, we will go through each of these problems in turn and present techniques to solve them. We will start by explaining the vanishing gradients problem and exploring some of the most popular solutions to this problem. 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 various optimizers that can speed up training large models tremendously compared to plain Gradient Descent. Finally, we will go through a few popular regularization techniques for large neural networks.

With these tools, you will be able to train very deep nets: welcome to Deep Learning!

Vanishing/Exploding Gradients Problems

As we discussed in Chapter 10, the backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regards 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 layer 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, so many layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which is mostly encountered in recurrent neural networks (see ???). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

Although this unfortunate behavior has been empirically observed for quite a while (it was one of the reasons why deep neural networks were mostly abandoned for a long time), it is only around 2010 that significant progress was made in understanding it. A paper titled “Understanding the Difficulty of Training Deep Feedforward Neural Networks” by Xavier Glorot and Yoshua Bengio¹ found a few suspects, including the combination of the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time, namely random initialization using 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 is actually made worse by the fact that the logistic function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the logistic function in deep networks).

¹ “Understanding the Difficulty of Training Deep Feedforward Neural Networks,” X. Glorot, Y Bengio (2010).

Looking at the logistic 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. 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.

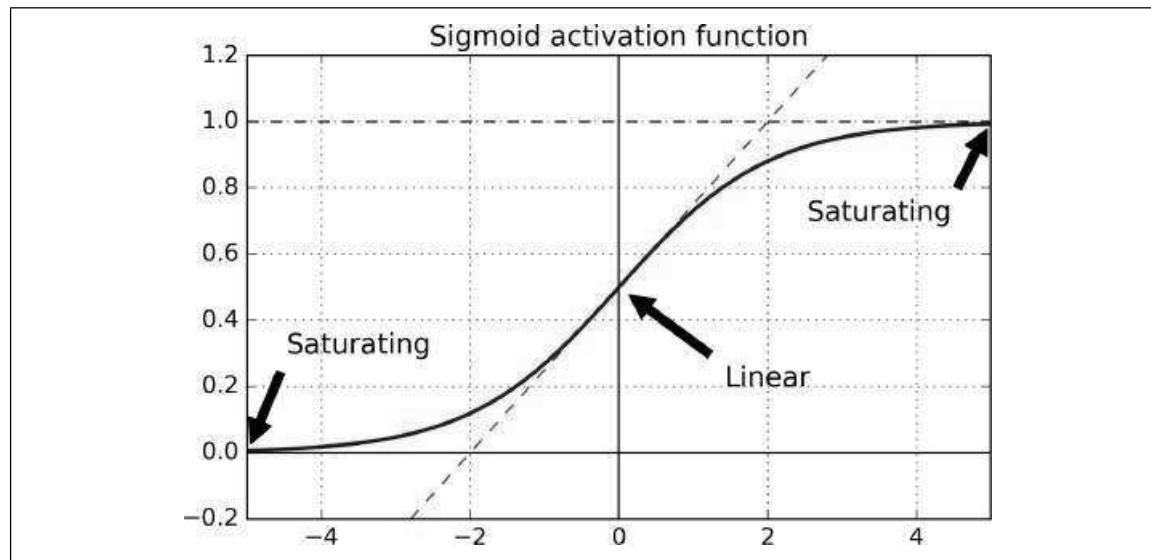


Figure 11-1. Logistic activation function saturation

Glorot and He Initialization

In their paper, Glorot and Bengio propose a way to significantly alleviate this problem. 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 argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs,² and we also 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 neurons (these numbers are called the *fan-in* and *fan-out* of the layer), but they proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly as

² Here's an analogy: if you set a microphone amplifier's 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.

described in Equation 11-1, where $fan_{avg} = (fan_{in} + fan_{out})/2$. This initialization strategy is called *Xavier initialization* (after the author's first name) or *Glorot initialization* (after his last name).

Equation 11-1. Glorot initialization (when using the logistic 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 just replace fan_{avg} with fan_{in} in Equation 11-1, you get an initialization strategy that was actually already proposed by Yann LeCun in the 1990s, called *LeCun initialization*, which was even recommended in the 1998 book *Neural Networks: Tricks of the Trade* by Genevieve Orr and Klaus-Robert Müller (Springer). It is equivalent to Glorot initialization when $fan_{in} = fan_{out}$. It took over a decade for researchers to realize just how important this trick really is. Using Glorot initialization can speed up training considerably, and it is one of the tricks that led to the current success of Deep Learning.

Some papers³ have provided similar strategies for different activation functions. 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 compute $r = \sqrt{3\sigma^2}$). The initialization strategy for the ReLU activation function (and its variants, including the ELU activation described shortly) is sometimes called *He initialization* (after the last name of its author). The SELU activation function will be explained later in this chapter. It should be used with LeCun initialization (preferably with a normal distribution, as we will see).

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

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1/fan_{avg}$
He	ReLU & variants	$2/fan_{in}$
LeCun	SELU	$1/fan_{in}$

By default, Keras uses Glorot initialization with a uniform distribution. You can change this to He initialization by setting `kernel_initializer="he_uniform"` or `kernel_initializer="he_normal"` when creating a layer, like this:

³ Such as “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” K. He et al. (2015).

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

If you want He initialization with a uniform distribution, but based on fan_{avg} rather than fan_{in} , you can use the `VarianceScaling` initializer like this:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                                 distribution='uniform')
keras.layers.Dense(10, activation="sigmoid", kernel_initializer=he_avg_init)
```

Nonsaturating Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/exploding gradients problems 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 roughly 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 quite 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 weighted sum of its inputs are negative for all instances in the training set. When this happens, it just keeps outputting 0s, and gradient descent does not affect it anymore since the gradient of the ReLU function is 0 when its input is negative.⁴

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*. This function is defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (see Figure 11-2). The hyperparameter α defines how much the function “leaks”: it is the slope of the function for $z < 0$, and is typically set to 0.01. This small slope ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up. A 2015 paper⁵ 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 $\alpha = 0.2$ (huge leak) seemed to result in better performance than $\alpha = 0.01$ (small leak). They also evaluated the *randomized leaky ReLU* (RReLU), where α is picked randomly in a given range during training, and it is fixed to an average value during testing. It also performed fairly well and seemed to act as a regularizer (reducing the risk of overfitting the training set).

⁴ Unless it is part of the first hidden layer, a dead neuron may sometimes come back to life: gradient descent may indeed tweak neurons in the layers below in such a way that the weighted sum of the dead neuron's inputs is positive again.

⁵ “Empirical Evaluation of Rectified Activations in Convolution Network,” B. Xu et al. (2015).

Finally, they also evaluated the *parametric leaky ReLU* (PReLU), where α 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). This was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

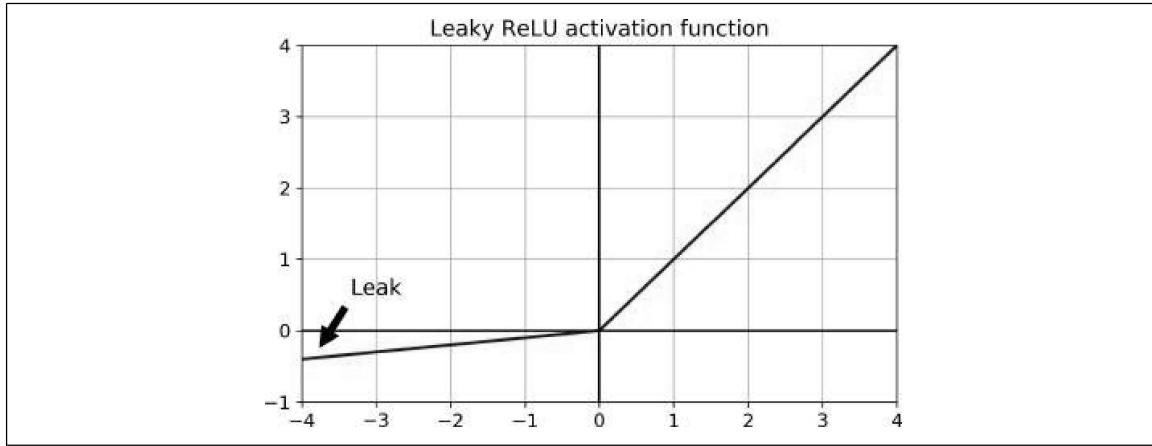


Figure 11-2. Leaky ReLU

Last but not least, a 2015 paper by Djork-Arné Clevert et al.⁶ proposed a new activation function called the *exponential linear unit* (ELU) that outperformed all the ReLU variants in their experiments: training time was reduced and the neural network performed better on the test set. It is represented in Figure 11-3, and Equation 11-2 shows its definition.

Equation 11-2. ELU activation function

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

⁶ “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” D. Clevert, T. Unterthiner, S. Hochreiter (2015).

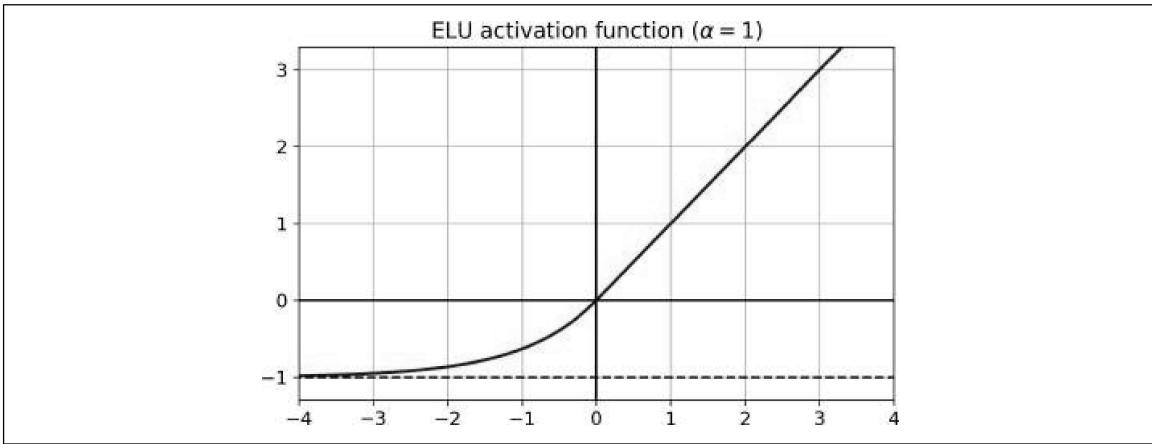


Figure 11-3. ELU activation function

It looks a lot like the ReLU function, with a few major differences:

- First it takes on negative values when $z < 0$, which allows the unit to have an average output closer to 0. This helps alleviate the vanishing gradients problem, as discussed earlier. The hyperparameter α defines 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 if you want.
- Second, it has a nonzero gradient for $z < 0$, which avoids the dead neurons problem.
- Third, if α 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 left and right of $z = 0$.

The main drawback of the ELU activation function is that it is slower to compute than the ReLU and its variants (due to the use of the exponential function), but during training this is compensated by the faster convergence rate. However, at test time an ELU network will be slower than a ReLU network.

Moreover, in a 2017 paper⁷ by Günter Klambauer et al., called “Self-Normalizing Neural Networks”, the authors showed that if you build a neural network composed exclusively of a stack of dense layers, and if all hidden layers use the SELU activation function (which is just a scaled version of the ELU activation function, as its name suggests), then the network will *self-normalize*: the output of each layer will tend to preserve mean 0 and standard deviation 1 during training, which solves the vanishing/exploding gradients problem. As a result, this activation function often outper-

⁷ “Self-Normalizing Neural Networks,” G. Klambauer, T. Unterthiner and A. Mayr (2017).

forms other activation functions very significantly for such neural nets (especially deep ones). However, there are a few conditions for self-normalization to happen:

- The input features must be standardized (mean 0 and standard deviation 1).
- Every hidden layer's weights must also be initialized using LeCun normal initialization. In Keras, this means setting `kernel_initializer="lecun_normal"`.
- The network's architecture must be sequential. Unfortunately, if you try to use SELU in non-sequential architectures, such as recurrent networks (see ???) or networks with *skip connections* (i.e., connections that skip layers, such as in wide & deep nets), self-normalization will not be guaranteed, so SELU will not necessarily outperform other activation functions.
- The paper only guarantees self-normalization if all layers are dense. However, in practice the SELU activation function seems to work great with convolutional neural nets as well (see Chapter 14).



So which activation function should you use for the hidden layers of your deep neural networks? Although your mileage will vary, in general SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic. If the network's architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at $z = 0$). If you care a lot about runtime latency, then you may prefer leaky ReLU. If you don't want to tweak yet another hyperparameter, you may just use the default α values used by Keras (e.g., 0.3 for the leaky ReLU). If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, in particular RReLU if your network is overfitting, or PReLU if you have a huge training set.

To use the leaky ReLU activation function, you must create a `LeakyReLU` instance like this:

```
leaky_relu = keras.layers.LeakyReLU(alpha=0.2)
layer = keras.layers.Dense(10, activation=leaky_relu,
                           kernel_initializer="he_normal")
```

For PReLU, just replace `LeakyReLU(alpha=0.2)` with `PReLU()`. There is currently no official implementation of RReLU in Keras, but you can fairly easily implement your own (see the exercises at the end of Chapter 12).

For SELU activation, just set `activation="selu"` and `kernel_initializer="lecun_normal"` when creating a layer:

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

Batch Normalization

Although using He initialization along with ELU (or any variant of ReLU) can significantly reduce 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,⁸ Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) to address the vanishing/exploding gradients problems. The technique consists of adding an operation in the model just before or after the activation function of each hidden layer, simply zero-centering and normalizing each input, then scaling and shifting the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, this 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 (e.g., using a `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 each input over the current mini-batch (hence the name "Batch Normalization"). The whole operation is summarized in Equation 11-3.

Equation 11-3. Batch Normalization algorithm

1. $\boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_B)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\sigma_B^2 + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$

- $\boldsymbol{\mu}_B$ is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).

⁸ "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," S. Ioffe and C. Szegedy (2015).

- σ_B is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- m_B is the number of instances in the mini-batch.
- $\hat{\mathbf{x}}^{(i)}$ is the vector of zero-centered and normalized inputs for instance i .
- γ 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).
- β is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- ϵ is a tiny number to avoid division by zero (typically 10^{-5}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a rescaled and shifted version of the inputs.

So during training, BN just standardizes its inputs then rescales and offsets them. Good! What about at test time? Well it is 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 mean and 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 (IID), so computing statistics over the batch instances would be unreliable (during training, the batches should not be too small, if possible more than 30 instances, and all instances should be IID, as we saw in Chapter 4). One solution could be 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, it is often preferred to estimate these final statistics during training using a moving average of the layer's input means and standard deviations. To sum up, four parameter vectors are learned in each batch-normalized layer: γ (the output scale vector) and β (the output offset vector) are learned through regular backpropagation, and μ (the final input mean vector), and σ (the final input standard deviation vector) are estimated using an exponential moving average. Note that μ and σ are estimated during training, but they are not used at all during training, only after training (to replace the batch input means and standard deviations in Equation 11-3).

The authors demonstrated that this technique 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 and commonly used to evaluate computer vision systems). The vanish-

ing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as the tanh and even the logistic activation function. The networks were also much less sensitive to the weight initialization. They 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 Normalization 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 Normalization also 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 we discussed earlier). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. So if you need predictions to be lightning-fast, you may want to check how well plain ELU + He initialization perform before playing with Batch Normalization.



You may find that training is rather slow, because each epoch takes much more time when you use batch normalization. However, 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 smaller (this is the time measured by the clock on your wall).

Implementing Batch Normalization with Keras

As with most things with Keras, implementing Batch Normalization is quite simple. Just add a `BatchNormalization` layer before or after each hidden layer’s activation function, and optionally add a BN layer as well as the first layer in your model. For example, this model applies BN after every hidden layer and as the first layer in the model (after flattening the input images):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

That's all! In this tiny example with just two hidden layers, it's unlikely that Batch Normalization will have a very positive impact, but for deeper networks it can make a tremendous difference.

Let's zoom in a bit. If you display the model summary, you can see that each BN layer adds 4 parameters per input: γ , β , μ and σ (for example, the first BN layer adds 3136 parameters, which is 4 times 784). The last two parameters, μ and σ , are the moving averages, they are not affected by backpropagation, so Keras calls them "Non-trainable"⁹ (if you count the total number of BN parameters, $3136 + 1200 + 400$, and divide by two, you get 2,368, which is the total number of non-trainable params in this model).

```
>>> model.summary()
Model: "sequential_3"

Layer (type)          Output Shape       Param #
=====
flatten_3 (Flatten)    (None, 784)        0
=====
batch_normalization_v2 (Batch Normalization) (None, 784)    3136
dense_50 (Dense)      (None, 300)        235500
batch_normalization_v2_1 (Batch Normalization) (None, 300)    1200
dense_51 (Dense)      (None, 100)        30100
batch_normalization_v2_2 (Batch Normalization) (None, 100)    400
=====
dense_52 (Dense)      (None, 10)         1010
=====

Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

Let's look at the parameters of the first BN layer. Two are trainable (by backprop), and two are not:

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

Now when you create a BN layer in Keras, it also creates two operations that will be called by Keras at each iteration during training. These operations will update the

⁹ However, they are estimated during training, based on the training data, so arguably they *are* trainable. In Keras, "Non-trainable" really means "untouched by backpropagation".

moving averages. Since we are using the TensorFlow backend, these operations are TensorFlow operations (we will discuss TF operations in Chapter 12).

```
>>> model.layers[1].updates
[<tf.Operation 'cond_2/Identity' type=Identity>,
 <tf.Operation 'cond_3/Identity' type=Identity>]
```

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, as it seems to depend on the task. So that's one more thing you can experiment with to see which option works best on your dataset. To add the BN layers before the activation functions, we must remove the activation function from the hidden layers, and add them as separate layers after the BN layers. Moreover, since a Batch Normalization layer includes one offset parameter per input, you can remove the bias term from the previous layer (just pass `use_bias=False` when creating it):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    keras.layers.Activation("elu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

The `BatchNormalization` class has quite 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 when updating the exponential moving averages: given a new value v (i.e., a new vector of input means or standard deviations computed over the current batch), the running average \hat{v} is updated using the following equation:

$$\hat{v} \leftarrow \hat{v} \times \text{momentum} + v \times (1 - \text{momentum})$$

A good momentum value is typically close to 1—for example, 0.9, 0.99, or 0.999 (you want more 9s for larger datasets and smaller mini-batches).

Another important hyperparameter is `axis`: it determines which axis should be normalized. It defaults to `-1`, meaning that by default it will normalize the last axis (using the means and standard deviations computed across the *other* axes). For example, when the input batch is 2D (i.e., the batch shape is [batch size, features]), this means that each input feature will be normalized based on the mean and standard deviation computed across all the instances in the batch. For example, the first BN layer in the previous code example will independently normalize (and rescale and shift) each of the 784 input features. However, if we move the first BN layer before the `Flatten`

layer, then the input batches will be 3D, with shape [batch size, height, width], therefore the BN layer will compute 28 means and 28 standard deviations (one per column of pixels, computed across all instances in the batch, and all rows in the column), and it will normalize all pixels in a given column using the same mean and standard deviation. There will also be just 28 scale parameters and 28 shift parameters. If instead you still want to treat each of the 784 pixels independently, then you should set `axis=[1, 2]`.

Notice that the BN layer does not perform the same computation during training and after training: it uses batch statistics during training, and the “final” statistics after training (i.e., the final value of the moving averages). Let’s take a peek at the source code of this class to see how this is handled:

```
class BatchNormalization(Layer):
    [...]
    def call(self, inputs, training=None):
        if training is None:
            training = keras.backend.learning_phase()
        [...]
```

The `call()` method is the one that actually performs the computations, and as you can see it has an extra `training` argument: if it is `None` it falls back to `keras.backend.learning_phase()`, which returns `1` during training (the `fit()` method ensures that). Otherwise, it returns `0`. If you ever need to write a custom layer, and it needs to behave differently during training and testing, simply use the same pattern (we will discuss custom layers in Chapter 12).

Batch Normalization has become one of the most used layers in deep neural networks, to the point that it is often omitted in the diagrams, as it is assumed that BN is added after every layer. However, a very recent paper¹⁰ by Hongyi Zhang et al. may well change this: the authors show that by using a novel fixed-update (fixup) weight initialization technique, they manage to train a very deep neural network (10,000 layers!) without BN, achieving state-of-the-art performance on complex image classification tasks.

Gradient Clipping

Another popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold. This is called *Gradient Clipping*.¹¹ This technique is most often used in recurrent neu-

¹⁰ “Fixup Initialization: Residual Learning Without Normalization,” Hongyi Zhang, Yann N. Dauphin, Tengyu Ma (2019).

¹¹ “On the difficulty of training recurrent neural networks,” R. Pascanu et al. (2013).

ral networks, as Batch Normalization is tricky to use in RNNs, as we will see in ???.

For other types of networks, BN is usually sufficient.

In Keras, implementing Gradient Clipping is just a matter of setting the `clipvalue` or `clipnorm` argument when creating an optimizer. For example:

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

This will clip every component of the gradient vector to a value between -1.0 and 1.0 . This means that all the partial derivatives of the loss (with regards to each and every trainable parameter) will be clipped between -1.0 and 1.0 . The threshold is a hyper-parameter you can tune. Note that it may change the orientation of the gradient vector: for example, if the original gradient vector is $[0.9, 100.0]$, it points mostly in the direction of the second axis, but once you clip it by value, you get $[0.9, 1.0]$, which points roughly in the diagonal between the two axes. In practice however, this approach works well. If you want to ensure that Gradient Clipping does not change the direction of the gradient vector, you should clip by norm by setting `clipnorm` instead of `clipvalue`. This will clip the whole gradient if its ℓ_2 norm is greater than the threshold you picked. For example, if you set `clipnorm=1.0`, then the vector $[0.9, 100.0]$ will be clipped to $[0.00899964, 0.9999595]$, preserving its orientation, but almost eliminating the first component. If you observe that the gradients explode during training (you can track the size of the gradients using TensorBoard), you may want to try both clipping by value and clipping by norm, with different threshold, and see which option performs best on the validation set.

Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle (we will discuss how to find them in Chapter 14), then just reuse the lower layers of this network: this is called *transfer learning*. It will not only speed up training considerably, but will also require much less training data.

For example, suppose that you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects. 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 11-4).

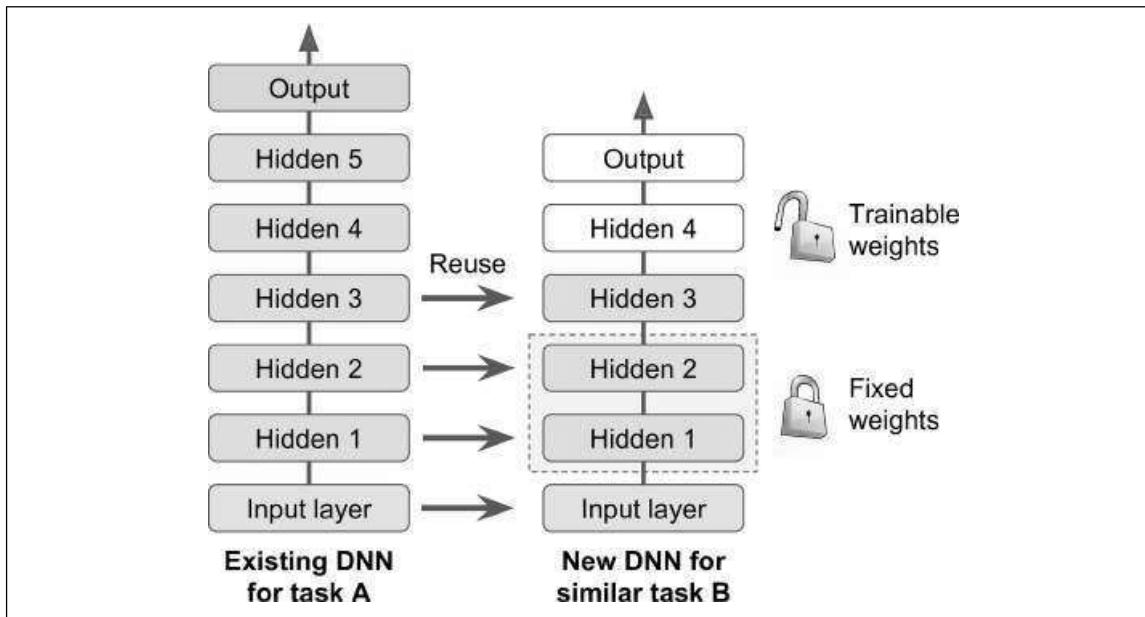


Figure 11-4. Reusing pretrained layers



If the input pictures of 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.

The output layer of the original model should usually be replaced since it is most likely not useful at all for the new task, and it may not even have the right number of outputs for the new task.

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 want to reuse (starting with the lower layers). For very similar tasks, you can try keeping all the hidden layers and just replace the output layer.

Try freezing all the reused layers first (i.e., make their weights non-trainable, so gradient descent won't modify them), 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 freeze all 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 add more hidden layers.

Transfer Learning With Keras

Let's look at an example. Suppose the fashion MNIST dataset only contained 8 classes, for example all classes except for sandals and shirts. Someone built and trained a Keras model on that set and got reasonably good performance (>90% accuracy). Let's call this model A. You now want to tackle a different task: you have images of sandals and shirts, and you want to train a binary classifier (positive=shirts, negative=sandals). However, your dataset is quite small, you only have 200 labeled images. When you train a new model for this task (let's call it model B), with the same architecture as model A, it performs reasonably well (97.2% accuracy), but since it's a much easier task (there are just 2 classes), you were hoping for more. 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, you need to load model A, and create a new model based on the model A's layers. Let's reuse all layers except for the output layer:

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Note that `model_A` and `model_B_on_A` now share some layers. When you train `model_B_on_A`, it will also affect `model_A`. If you want to avoid that, you need to clone `model_A` before you reuse its layers. To do this, you must clone model A's architecture, then copy its weights (since `clone_model()` does not clone the weights):

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

Now we could just train `model_B_on_A` for task B, 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. To do this, simply set every layer's `trainable` attribute to `False` and compile the model:

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False
```

```
model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                      metrics=["accuracy"])
```



You must always compile your model after you freeze or unfreeze layers.

Next, we can train the model for a few epochs, then unfreeze the reused layers (which requires compiling the model again) and continue training to fine-tune the reused layers for task B. After unfreezing the reused layers, it is usually a good idea to reduce the learning rate, once again to avoid damaging the reused weights:

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                            validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) # the default lr is 1e-3
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                      metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                            validation_data=(X_valid_B, y_valid_B))
```

So, what's the final verdict? Well this model's test accuracy is 99.25%, which means that transfer learning reduced the error rate from 2.8% down to almost 0.7%! That's a factor of 4!

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]
```

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 just looks too positive, you should be suspicious: perhaps the flashy new technique does not 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 on the way. Most of the time, this is not malicious at all, but it is part of the reason why so many results in Science can never be reproduced.

So why did I cheat? Well it turns out that transfer learning does not work very well with small dense networks: it works best with deep convolutional neural networks, so we will revisit transfer learning in Chapter 14, using the same techniques (and this time there will be no 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 all hope! First, you should of course try to gather more labeled training data, but if this is too hard or too expensive, you may still be able to perform *unsupervised pretraining* (see Figure 11-5). It is often rather cheap to gather unlabeled training examples, but quite expensive to label them. If you can gather plenty of unlabeled training data, you can try to train the layers one by one, starting with the lowest layer and then going up, using an unsupervised feature detector algorithm such as *Restricted Boltzmann Machines* (RBMs; see ???) or autoencoders (see ???). Each layer is trained on the output of the previously trained layers (all layers except the one being trained are frozen). Once all layers have been trained this way, you can add the output layer for your task, and fine-tune the final network using supervised learning (i.e., with the labeled training examples). At this point, you can unfreeze all the pretrained layers, or just some of the upper ones.

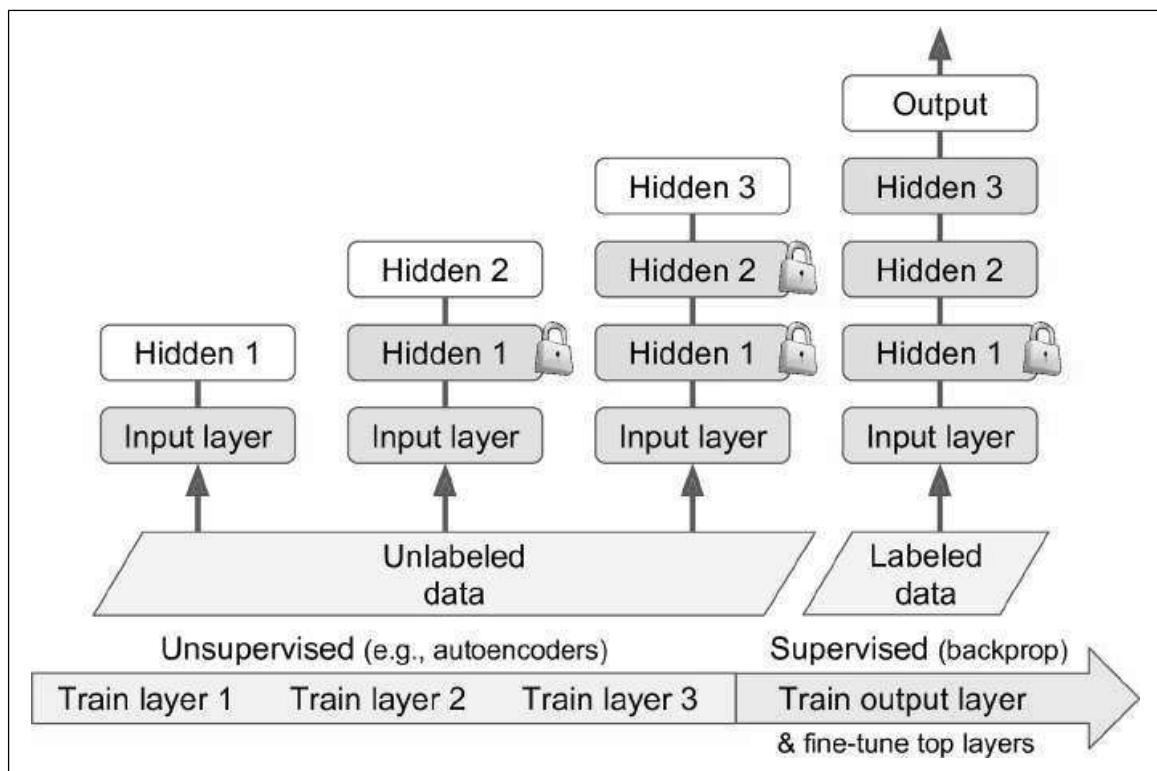


Figure 11-5. Unsupervised pretraining

This is a rather long and tedious process, but it often works well; in fact, 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 using RBMs) was the norm for deep nets, and it was only after the vanishing gradients problem was alleviated that it became much more com-

mon to train DNNs purely using supervised learning. However, unsupervised pre-training (today typically using autoencoders 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.

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. However, you could gather a lot of pictures of random people on the web and train a first neural network to detect whether or not 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 using little training data.

For *natural language processing* (NLP) applications, you can easily download 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 (we will discuss more pretraining tasks in ???).



Self-supervised learning is when you automatically generate the labels from the data itself, then you train a model on the resulting “labeled” dataset using supervised learning techniques. Since this approach requires no human labeling whatsoever, it is best classified as a form of unsupervised learning.

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 Normalization, and reusing parts of a pretrained network (possibly built on 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