c. Use grid search with cross-validation (with the help of the `GridSearchCV` class) to find good hyperparameter values for a `DecisionTreeClassifier`. Hint: try various values for `max_leaf_nodes`.

d. Train it on the full training set using these hyperparameters, and measure your model's performance on the test set. You should get roughly 85% to 87% accuracy.

8. Grow a forest.

a. Continuing the previous exercise, generate 1,000 subsets of the training set, each containing 100 instances selected randomly. Hint: you can use Scikit-Learn's `ShuffleSplit` class for this.

b. Train one Decision Tree on each subset, using the best hyperparameter values found above. Evaluate these 1,000 Decision Trees on the test set. Since they were trained on smaller sets, these Decision Trees will likely perform worse than the first Decision Tree, achieving only about 80% accuracy.

c. Now comes the magic. For each test set instance, generate the predictions of the 1,000 Decision Trees, and keep only the most frequent prediction (you can use SciPy's `mode()` function for this). This gives you *majority-vote predictions* over the test set.

d. Evaluate these predictions on the test set: you should obtain a slightly higher accuracy than your first model (about 0.5 to 1.5% higher). Congratulations, you have trained a Random Forest classifier!

Solutions to these exercises are available in Appendix A.

CHAPTER 7

Ensemble Learning and Random Forests

Suppose you ask a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer. This is called the *wisdom of the crowd*. Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*; thus, this technique is called *Ensemble Learning*, and an Ensemble Learning algorithm is called an *Ensemble method*.

For example, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you just obtain the predictions of all individual trees, then predict the class that gets the most votes (see the last exercise in Chapter 6). Such an ensemble of Decision Trees is called a *Random Forest*, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

Moreover, as we discussed in Chapter 2, you will often use Ensemble methods near the end of a project, once you have already built a few good predictors, to combine them into an even better predictor. In fact, the winning solutions in Machine Learning competitions often involve several Ensemble methods (most famously in the Netflix Prize competition).

In this chapter we will discuss the most popular Ensemble methods, including *bagging*, *boosting*, *stacking*, and a few others. We will also explore Random Forests.

Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and perhaps a few more (see Figure 7-1).

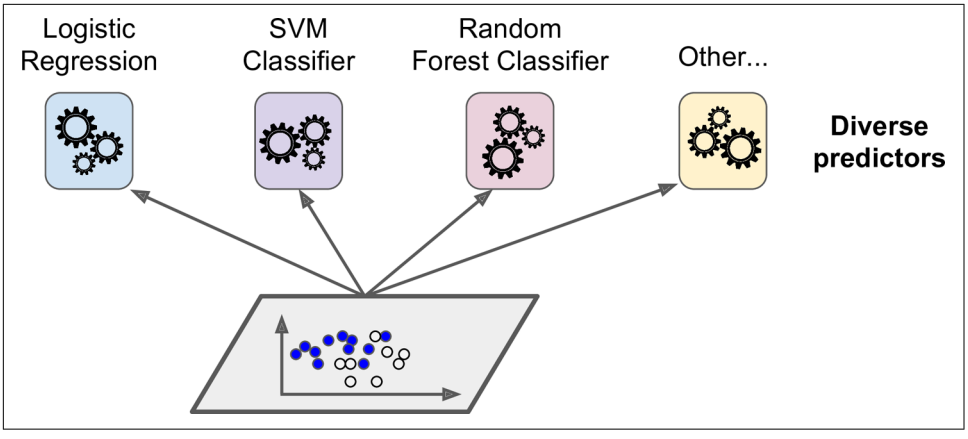

Figure 7-1. Training diverse classifiers

A very simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes. This majority-vote classifier is called a *hard voting* classifier (see Figure 7-2).

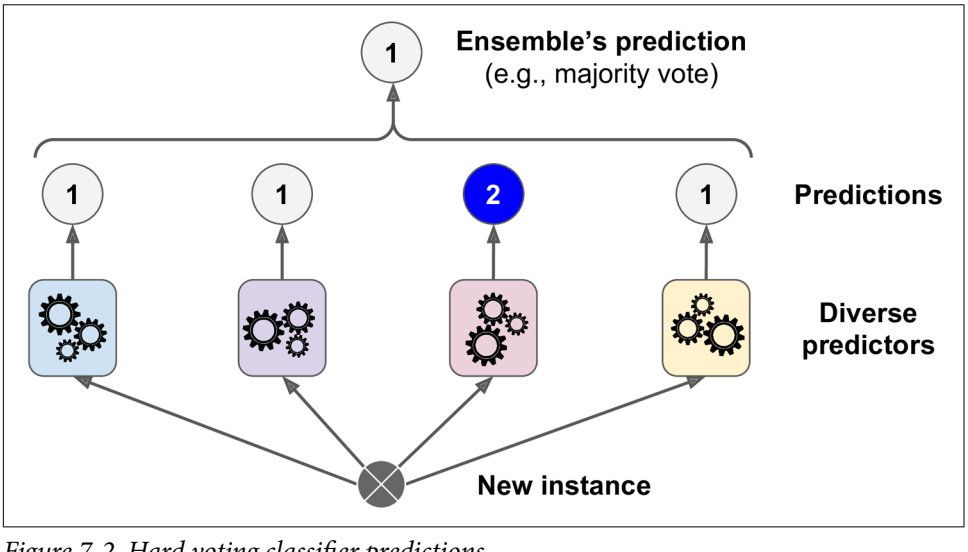

Figure 7-2. Hard voting classifier predictions

Somewhat surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a *weak learner* (meaning it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy), provided there are a sufficient number of weak learners and they are sufficiently diverse.

How is this possible? The following analogy can help shed some light on this mystery. Suppose you have a slightly biased coin that has a 51% chance of coming up heads, and 49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, and hence a majority of heads. If you do the math, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the *law of large numbers*: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%). Figure 7-3 shows 10 series of biased coin tosses. You can see that as the number of tosses increases, the ratio of heads approaches 51%. Eventually all 10 series end up so close to 51% that they are consistently above 50%.

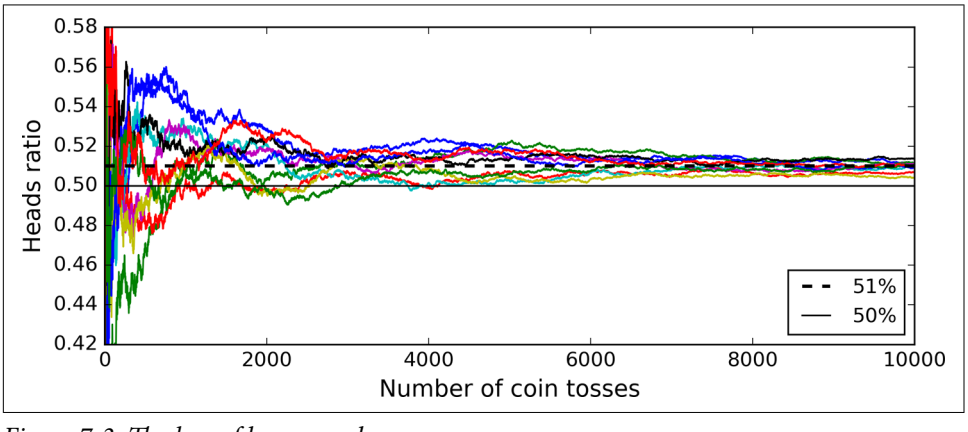

Figure 7-3. The law of large numbers

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case since they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

The following code creates and trains a voting classifier in Scikit-Learn, composed of three diverse classifiers (the training set is the moons dataset, introduced in Chapter 5):

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
        estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
        voting='hard'
    )
voting_clf.fit(X_train, y_train)
```

Let's look at each classifier's accuracy on the test set:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
>>>     clf.fit(X_train, y_train)
>>>     y_pred = clf.predict(X_test)
>>>     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

There you have it! The voting classifier slightly outperforms all the individual classifiers.

If all classifiers are able to estimate class probabilities (i.e., they have a `predict_proba()` method), then you can tell Scikit-Learn to predict the class with the highest class probability, averaged over all the individual classifiers. This is called *soft voting*. It often achieves higher performance than hard voting because it gives more weight to highly confident votes. All you need to do is replace `voting="hard"` with `voting="soft"` and ensure that all classifiers can estimate class probabilities. This is not the case of the `SVC` class by default, so you need to set its `probability` hyperparameter to `True` (this will make the `SVC` class use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method). If you modify the preceding code to use soft voting, you will find that the voting classifier achieves over 91% accuracy!

Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set. When sampling is performed *with* replacement, this method is called *bagging*[1] (short for *bootstrap aggregating*[2]). When sampling is performed *without* replacement, it is called *pasting*.[3]

In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented in Figure 7-4.

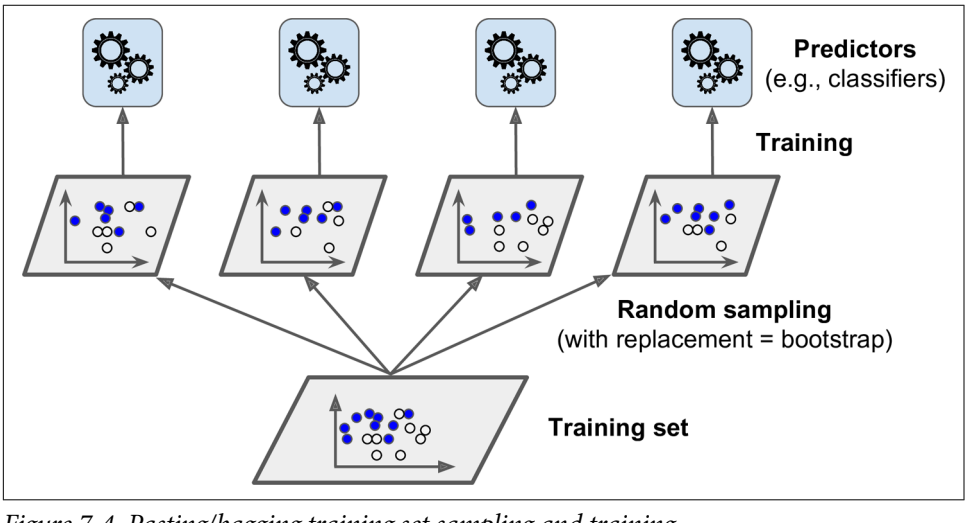

Figure 7-4. Pasting/bagging training set sampling and training

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the *statistical mode* (i.e., the most frequent prediction, just like a hard voting classifier) for classification, or the average for regression. Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.[4] Generally, the net result is that the

1 "Bagging Predictors," L. Breiman (1996).

2 In statistics, resampling with replacement is called *bootstrapping*.

3 "Pasting small votes for classification in large databases and on-line," L. Breiman (1999).

4 Bias and variance were introduced in Chapter 4.

ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

As you can see in Figure 7-4, predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons why bagging and pasting are such popular methods: they scale very well.

Bagging and Pasting in Scikit-Learn

Scikit-Learn offers a simple API for both bagging and pasting with the `BaggingClassifier` class (or `BaggingRegressor` for regression). The following code trains an ensemble of 500 Decision Tree classifiers,[5] each trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`). The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions (–1 tells Scikit-Learn to use all available cores):

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
        DecisionTreeClassifier(), n_estimators=500,
        max_samples=100, bootstrap=True, n_jobs=-1
    )
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

The `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with Decision Trees classifiers.

Figure 7-5 compares the decision boundary of a single Decision Tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moons dataset. As you can see, the ensemble's predictions will likely generalize much better than the single Decision Tree's predictions: the ensemble has a comparable bias but a smaller variance (it makes roughly the same number of errors on the training set, but the decision boundary is less irregular).

5 `max_samples` can alternatively be set to a float between 0.0 and 1.0, in which case the max number of instances to sample is equal to the size of the training set times `max_samples`.

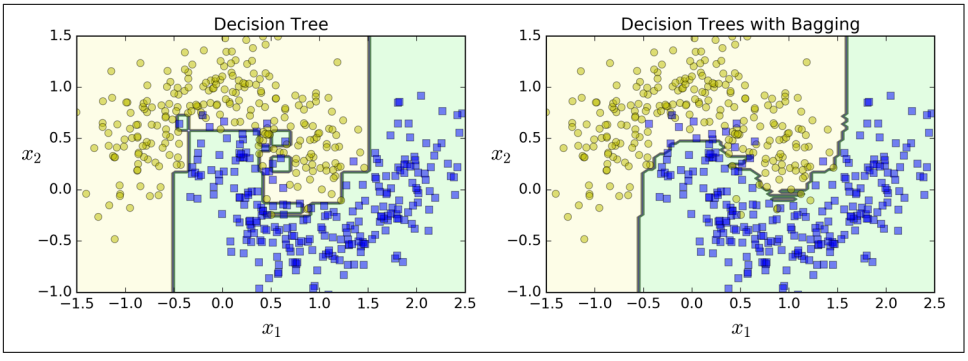

Figure 7-5. A single Decision Tree versus a bagging ensemble of 500 trees

Bootstrapping introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting, but this also means that predictors end up being less correlated so the ensemble's variance is reduced. Overall, bagging often results in better models, which explains why it is generally preferred. However, if you have spare time and CPU power you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

Out-of-Bag Evaluation

With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a `BaggingClassifier` samples *m* training instances with replacement (`bootstrap=True`), where *m* is the size of the training set. This means that only about 63% of the training instances are sampled on average for each predictor.[6] The remaining 37% of the training instances that are not sampled are called *out-of-bag* (oob) instances. Note that they are not the same 37% for all predictors.

Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set or cross-validation. You can evaluate the ensemble itself by averaging out the oob evaluations of each predictor.

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training. The following code demonstrates this. The resulting evaluation score is available through the `oob_score_` variable:

```
>>> bag_clf = BaggingClassifier(
>>>         DecisionTreeClassifier(), n_estimators=500,
>>>         bootstrap=True, n_jobs=-1, oob_score=True)
```

6 As *m* grows, this ratio approaches 1 – exp(–1) ≈ 63.212%.

```
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.93066666666666664
```

According to this oob evaluation, this `BaggingClassifier` is likely to achieve about 93.1% accuracy on the test set. Let's verify this:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.93600000000000005
```

We get 93.6% accuracy on the test set—close enough!

The oob decision function for each training instance is also available through the `oob_decision_function_` variable. In this case (since the base estimator has a `predict_proba()` method) the decision function returns the class probabilities for each training instance. For example, the oob evaluation estimates that the second training instance has a 60.6% probability of belonging to the positive class (and 39.4% of belonging to the positive class):

```
>>> bag_clf.oob_decision_function_
array([[ 0.        ,  1.        ],
       [ 0.60588235,  0.39411765],
       [ 1.        ,  0.        ],
       ...
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.48958333,  0.51041667]])
```

Random Patches and Random Subspaces

The `BaggingClassifier` class supports sampling the features as well. This is controlled by two hyperparameters: `max_features` and `bootstrap_features`. They work the same way as `max_samples` and `bootstrap`, but for feature sampling instead of instance sampling. Thus, each predictor will be trained on a random subset of the input features.

This is particularly useful when you are dealing with high-dimensional inputs (such as images). Sampling both training instances and features is called the *Random Patches* method.[7] Keeping all training instances (i.e., `bootstrap=False` and `max_samples=1.0`) but sampling features (i.e., `bootstrap_features=True` and/or `max_features` smaller than 1.0) is called the *Random Subspaces* method.[8]

7 "Ensembles on Random Patches," G. Louppe and P. Geurts (2012).

8 "The random subspace method for constructing decision forests," Tin Kam Ho (1998).

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forests

As we have discussed, a Random Forest[9] is an ensemble of Decision Trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can instead use the `RandomForestClassifier` class, which is more convenient and optimized for Decision Trees[10] (similarly, there is a `RandomForestRegressor` class for regression tasks). The following code trains a Random Forest classifier with 500 trees (each limited to maximum 16 nodes), using all available CPU cores:

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself.[11]

The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node (see Chapter 6), it searches for the best feature among a random subset of features. This results in a greater tree diversity, which (once again) trades a higher bias for a lower variance, generally yielding an overall better model. The following `BaggingClassifier` is roughly equivalent to the previous `RandomForestClassifier`:

```
bag_clf = BaggingClassifier(
        DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
        n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1
    )
```

9 "Random Decision Forests," T. Ho (1995).

10 The `BaggingClassifier` class remains useful if you want a bag of something other than Decision Trees.

11 There are a few notable exceptions: `splitter` is absent (forced to `"random"`), `presort` is absent (forced to `False`), `max_samples` is absent (forced to `1.0`), and `base_estimator` is absent (forced to `DecisionTreeClassifier` with the provided hyperparameters).

Extra-Trees

When you are growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting (as discussed earlier). It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular Decision Trees do).

A forest of such extremely random trees is simply called an *Extremely Randomized Trees* ensemble[12] (or *Extra-Trees* for short). Once again, this trades more bias for a lower variance. It also makes Extra-Trees much faster to train than regular Random Forests since finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree.

You can create an Extra-Trees classifier using Scikit-Learn's `ExtraTreesClassifier` class. Its API is identical to the `RandomForestClassifier` class. Similarly, the `ExtraTreesRegressor` class has the same API as the `RandomForestRegressor` class.

It is hard to tell in advance whether a `RandomForestClassifier` will perform better or worse than an `ExtraTreesClassifier`. Generally, the only way to know is to try both and compare them using cross-validation (and tuning the hyperparameters using grid search).

Feature Importance

Lastly, if you look at a single Decision Tree, important features are likely to appear closer to the root of the tree, while unimportant features will often appear closer to the leaves (or not at all). It is therefore possible to get an estimate of a feature's importance by computing the average depth at which it appears across all trees in the forest. Scikit-Learn computes this automatically for every feature after training. You can access the result using the `feature_importances_` variable. For example, the following code trains a `RandomForestClassifier` on the iris dataset (introduced in Chapter 4) and outputs each feature's importance. It seems that the most important features are the petal length (44%) and width (42%), while sepal length and width are rather unimportant in comparison (11% and 2%, respectively):

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
>>>     print(name, score)
sepal length (cm) 0.112492250999
```

12 "Extremely randomized trees," P. Geurts, D. Ernst, L. Wehenkel (2005).

```
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Similarly, if you train a Random Forest classifier on the MNIST dataset (introduced in Chapter 3) and plot each pixel's importance, you get the image represented in Figure 7-6.

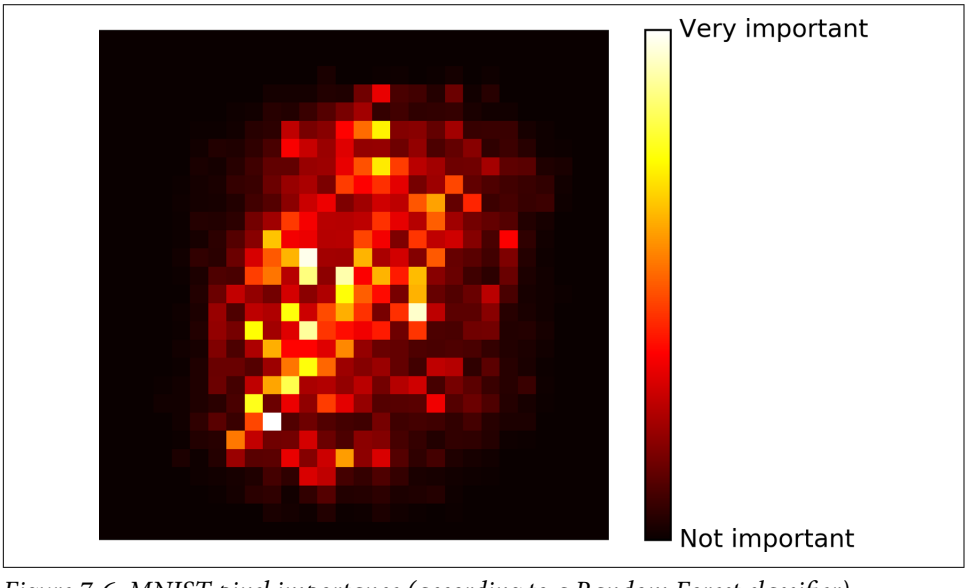

Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

Random Forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

Boosting

Boosting (originally called *hypothesis boosting*) refers to any Ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but by far the most popular are *AdaBoost*[13] (short for *Adaptive Boosting*) and *Gradient Boosting*. Let's start with AdaBoost.

13 "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," Yoav Freund, Robert E. Schapire (1997).

AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

For example, to build an AdaBoost classifier, a first base classifier (such as a Decision Tree) is trained and used to make predictions on the training set. The relative weight of misclassified training instances is then increased. A second classifier is trained using the updated weights and again it makes predictions on the training set, weights are updated, and so on (see Figure 7-7).

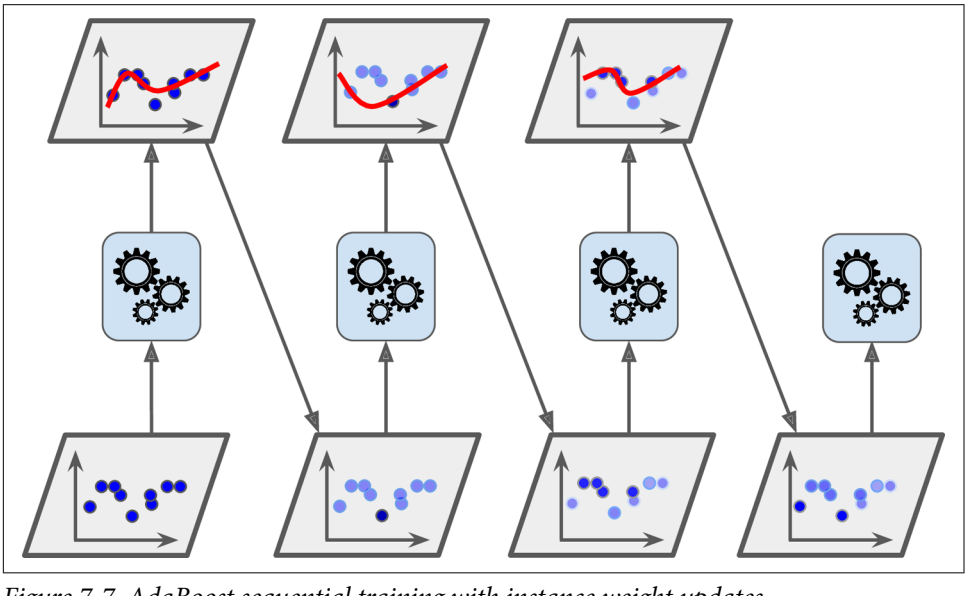

Figure 7-7. AdaBoost sequential training with instance weight updates

Figure 7-8 shows the decision boundaries of five consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel[14]). The first classifier gets many instances wrong, so their weights get boosted. The second classifier therefore does a better job on these instances, and so on. The plot on the right represents the same sequence of predictors except that the learning rate is halved (i.e., the misclassified instance weights are boosted half as much at every iteration). As you can see, this sequential learning technique has some similarities with Gradient Descent, except that instead of tweaking a single predictor's

14 This is just for illustrative purposes. SVMs are generally not good base predictors for AdaBoost, because they are slow and tend to be unstable with AdaBoost.

parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better.

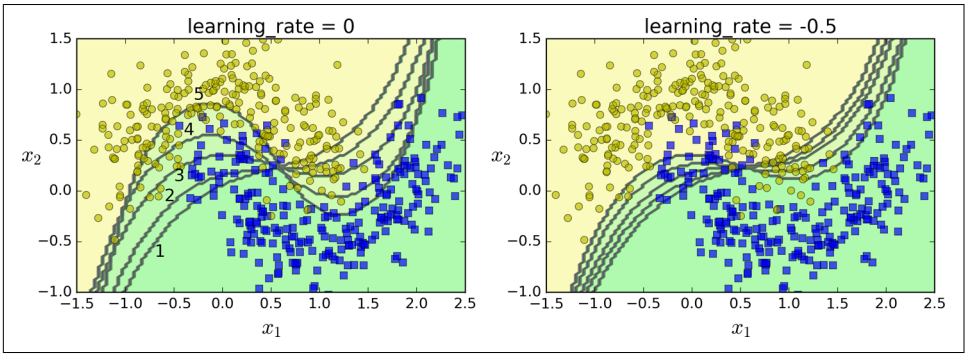

Figure 7-8. Decision boundaries of consecutive predictors

Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.

There is one important drawback to this sequential learning technique: it cannot be parallelized (or only partially), since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bagging or pasting.

Let's take a closer look at the AdaBoost algorithm. Each instance weight $w^{(i)}$ is initially set to $\frac{1}{m}$. A first predictor is trained and its weighted error rate r_1 is computed on the training set; see Equation 7-1.

Equation 7-1. Weighted error rate of the j^{th} predictor

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

where $\hat{y}_j^{(i)}$ is the j^{th} predictor's prediction for the i^{th} instance.

The predictor's weight α_j is then computed using Equation 7-2, where η is the learning rate hyperparameter (defaults to 1).[15] The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative.

Equation 7-2. Predictor weight

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Next the instance weights are updated using Equation 7-3: the misclassified instances are boosted.

Equation 7-3. Weight update rule

$$\text{for } i = 1, 2, \cdots, m$$
$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp\left(\alpha_j\right) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Then all the instance weights are normalized (i.e., divided by $\sum_{i=1}^{m} w^{(i)}$).

Finally, a new predictor is trained using the updated weights, and the whole process is repeated (the new predictor's weight is computed, the instance weights are updated, then another predictor is trained, and so on). The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights α_j. The predicted class is the one that receives the majority of weighted votes (see Equation 7-4).

Equation 7-4. AdaBoost predictions

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x}) = k}}^{N} \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

15 The original AdaBoost algorithm does not use a learning rate hyperparameter.

Scikit-Learn actually uses a multiclass version of AdaBoost called *SAMME*[16] (which stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*). When there are just two classes, SAMME is equivalent to AdaBoost. Moreover, if the predictors can estimate class probabilities (i.e., if they have a `predict_proba()` method), Scikit-Learn can use a variant of SAMME called *SAMME.R* (the *R* stands for "Real"), which relies on class probabilities rather than predictions and generally performs better.

The following code trains an AdaBoost classifier based on 200 *Decision Stumps* using Scikit-Learn's `AdaBoostClassifier` class (as you might expect, there is also an `AdaBoostRegressor` class). A Decision Stump is a Decision Tree with `max_depth=1`—in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
        DecisionTreeClassifier(max_depth=1), n_estimators=200,
        algorithm="SAMME.R", learning_rate=0.5
    )
ada_clf.fit(X_train, y_train)
```

If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.

Gradient Boosting

Another very popular Boosting algorithm is *Gradient Boosting*.[17] Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

Let's go through a simple regression example using Decision Trees as the base predictors (of course Gradient Boosting also works great with regression tasks). This is called *Gradient Tree Boosting*, or *Gradient Boosted Regression Trees* (*GBRT*). First, let's fit a `DecisionTreeRegressor` to the training set (for example, a noisy quadratic training set):

16 For more details, see "Multi-Class AdaBoost," J. Zhu et al. (2006).

17 First introduced in "Arcing the Edge," L. Breiman (1997).

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Now train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Then we train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

Figure 7-9 represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

A simpler way to train GBRT ensembles is to use Scikit-Learn's `GradientBoostingRegressor` class. Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of Decision Trees (e.g., `max_depth`, `min_samples_leaf`, and so on), as well as hyperparameters to control the ensemble training, such as the number of trees (`n_estimators`). The following code creates the same ensemble as the previous one:

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

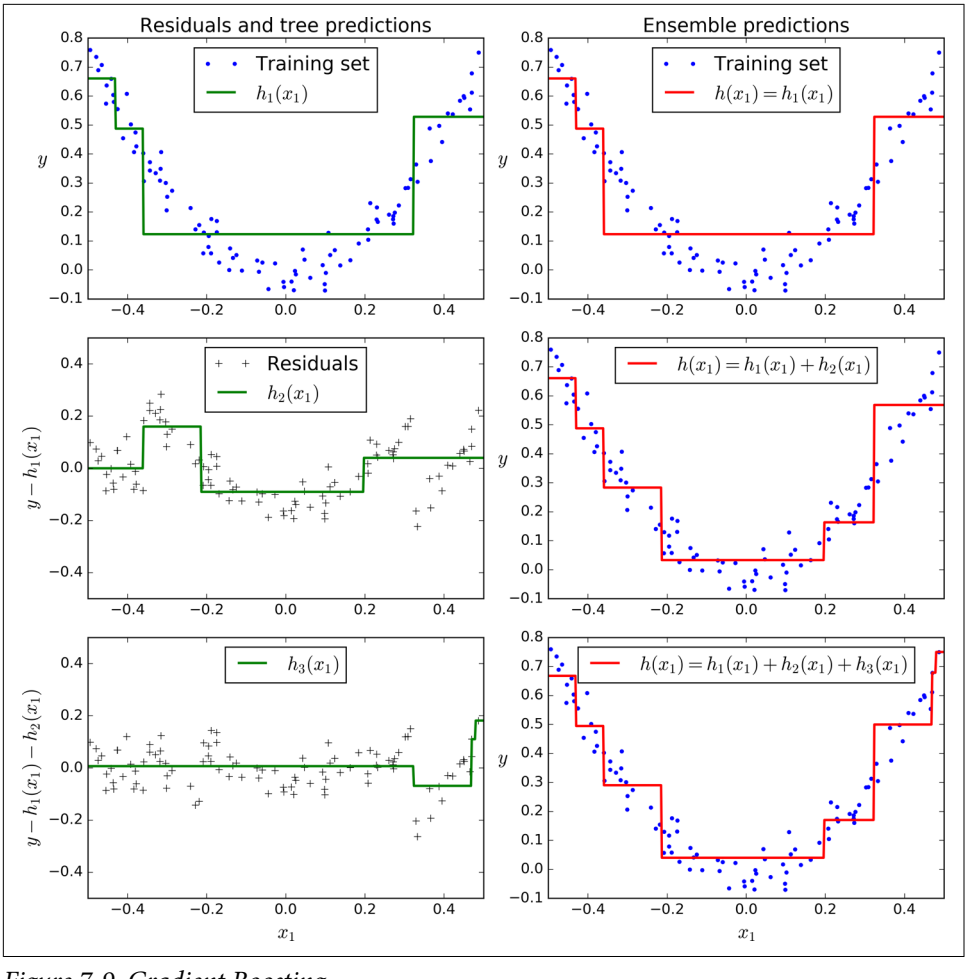

Figure 7-9. Gradient Boosting

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as `0.1`, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called *shrinkage*. Figure 7-10 shows two GBRT ensembles trained with a low learning rate: the one on the left does not have enough trees to fit the training set, while the one on the right has too many trees and overfits the training set.

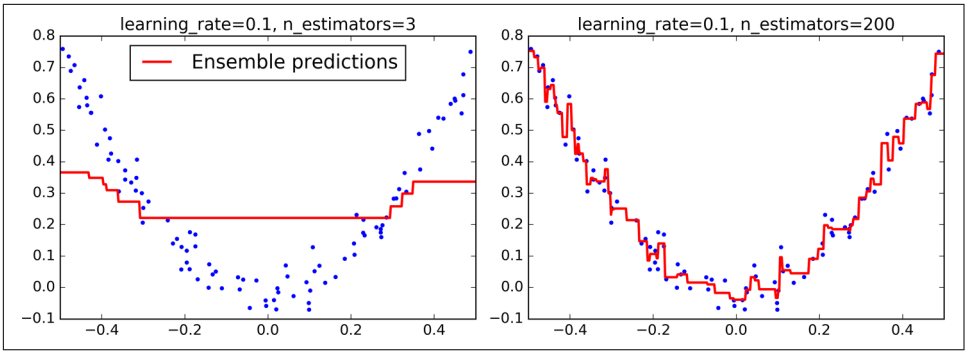

Figure 7-10. GBRT ensembles with not enough predictors (left) and too many (right)

In order to find the optimal number of trees, you can use early stopping (see Chapter 4). A simple way to implement this is to use the `staged_predict()` method: it returns an iterator over the predictions made by the ensemble at each stage of training (with one tree, two trees, etc.). The following code trains a GBRT ensemble with 120 trees, then measures the validation error at each stage of training to find the optimal number of trees, and finally trains another GBRT ensemble using the optimal number of trees:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors)

gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

The validation errors are represented on the left of Figure 7-11, and the best model's predictions are represented on the right.

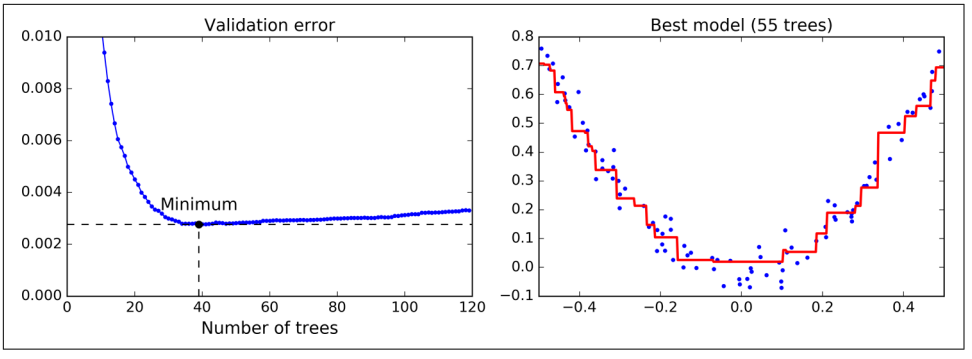

Figure 7-11. Tuning the number of trees using early stopping

It is also possible to implement early stopping by actually stopping training early (instead of training a large number of trees first and then looking back to find the optimal number). You can do so by setting `warm_start=True`, which makes Scikit-Learn keep existing trees when the `fit()` method is called, allowing incremental training. The following code stops training when the validation error does not improve for five iterations in a row:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping
```

The `GradientBoostingRegressor` class also supports a `subsample` hyperparameter, which specifies the fraction of training instances to be used for training each tree. For example, if `subsample=0.25`, then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this trades a higher bias for a lower variance. It also speeds up training considerably. This technique is called *Stochastic Gradient Boosting*.

It is possible to use Gradient Boosting with other cost functions. This is controlled by the `loss` hyperparameter (see Scikit-Learn's documentation for more details).

Stacking

The last Ensemble method we will discuss in this chapter is called *stacking* (short for *stacked generalization*).[18] It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation? Figure 7-12 shows such an ensemble performing a regression task on a new instance. Each of the bottom three predictors predicts a different value (3.1, 2.7, and 2.9), and then the final predictor (called a *blender*, or a *meta learner*) takes these predictions as inputs and makes the final prediction (3.0).

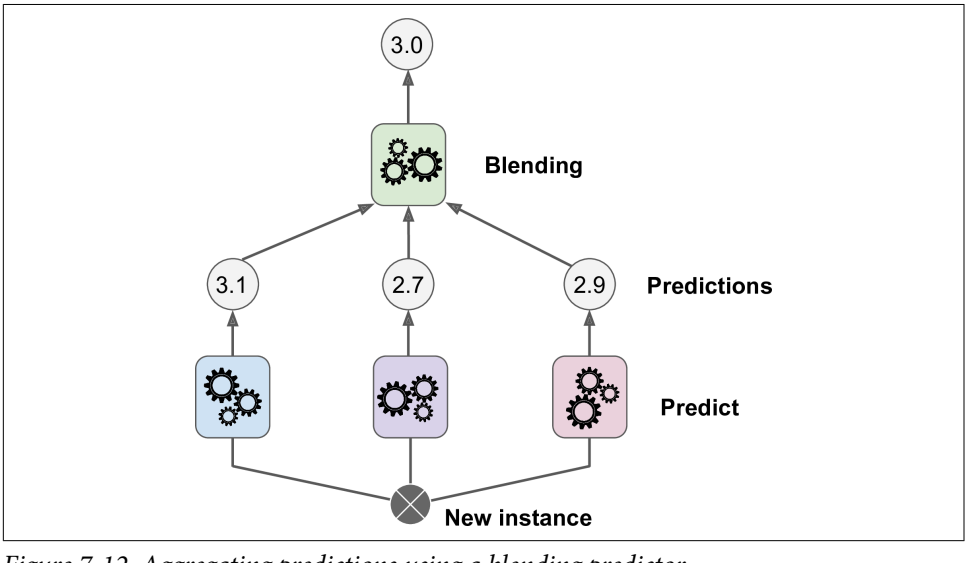

Figure 7-12. Aggregating predictions using a blending predictor

To train the blender, a common approach is to use a hold-out set.[19] Let's see how it works. First, the training set is split in two subsets. The first subset is used to train the predictors in the first layer (see Figure 7-13).

18 "Stacked Generalization," D. Wolpert (1992).

19 Alternatively, it is possible to use out-of-fold predictions. In some contexts this is called *stacking*, while using a hold-out set is called *blending*. However, for many people these terms are synonymous.

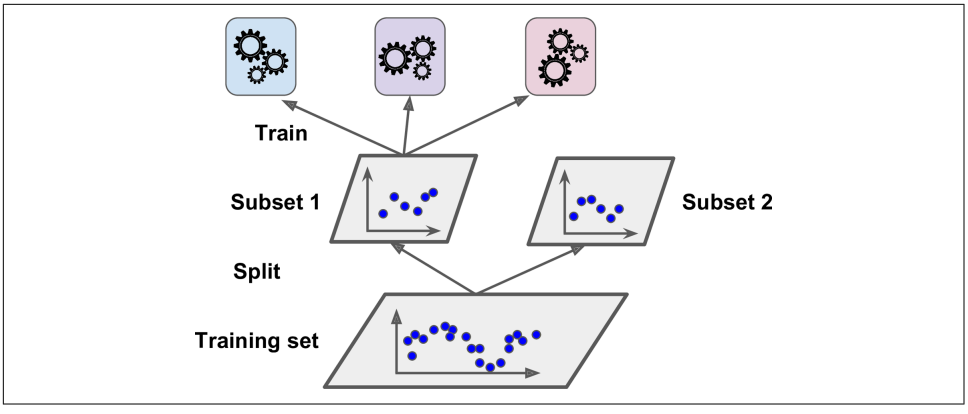

Figure 7-13. Training the first layer

Next, the first layer predictors are used to make predictions on the second (held-out) set (see Figure 7-14). This ensures that the predictions are "clean," since the predictors never saw these instances during training. Now for each instance in the hold-out set there are three predicted values. We can create a new training set using these predicted values as input features (which makes this new training set three-dimensional), and keeping the target values. The blender is trained on this new training set, so it learns to predict the target value given the first layer's predictions.

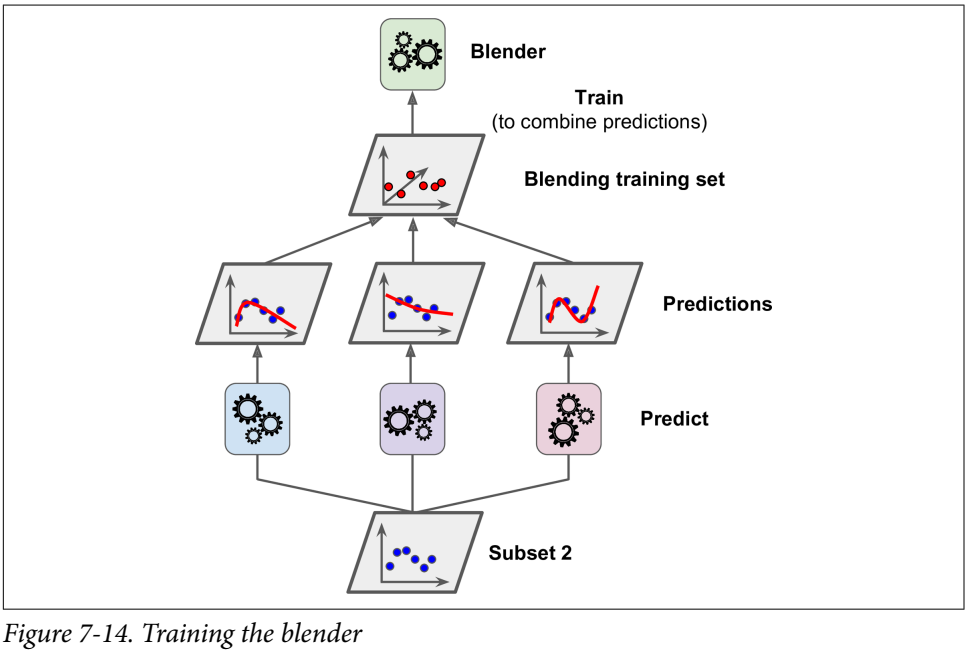

Figure 7-14. Training the blender

It is actually possible to train several different blenders this way (e.g., one using Linear Regression, another using Random Forest Regression, and so on): we get a whole layer of blenders. The trick is to split the training set into three subsets: the first one is used to train the first layer, the second one is used to create the training set used to train the second layer (using predictions made by the predictors of the first layer), and the third one is used to create the training set to train the third layer (using predictions made by the predictors of the second layer). Once this is done, we can make a prediction for a new instance by going through each layer sequentially, as shown in Figure 7-15.

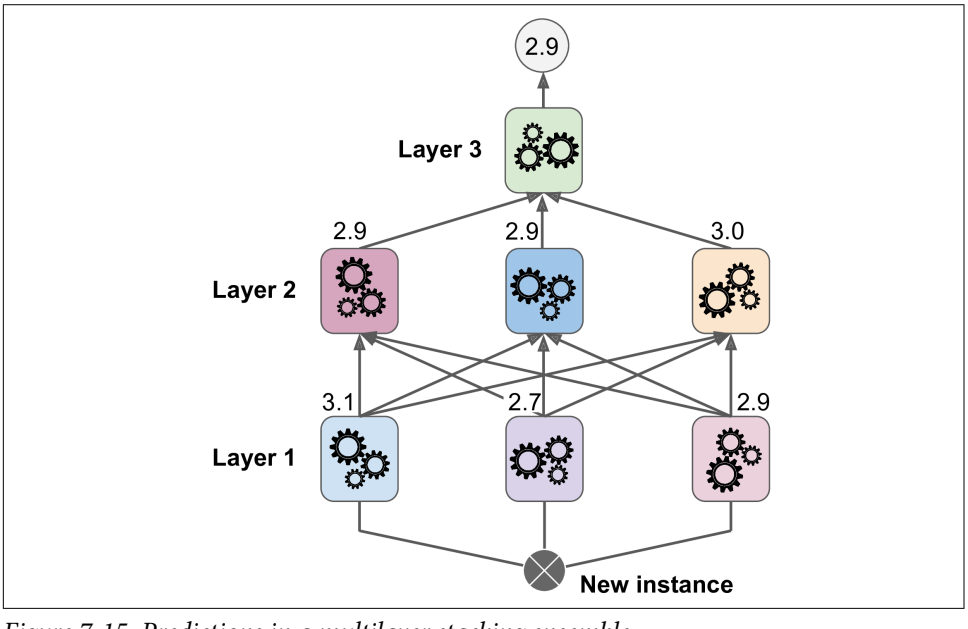

Figure 7-15. Predictions in a multilayer stacking ensemble

Unfortunately, Scikit-Learn does not support stacking directly, but it is not too hard to roll out your own implementation (see the following exercises). Alternatively, you can use an open source implementation such as `brew` (available at *https://github.com/viisar/brew*).

Exercises

1. If you have trained five different models on the exact same training data, and they all achieve 95% precision, is there any chance that you can combine these models to get better results? If so, how? If not, why?
2. What is the difference between hard and soft voting classifiers?

3. Is it possible to speed up training of a bagging ensemble by distributing it across multiple servers? What about pasting ensembles, boosting ensembles, random forests, or stacking ensembles?
4. What is the benefit of out-of-bag evaluation?
5. What makes Extra-Trees more random than regular Random Forests? How can this extra randomness help? Are Extra-Trees slower or faster than regular Random Forests?
6. If your AdaBoost ensemble underfits the training data, what hyperparameters should you tweak and how?
7. If your Gradient Boosting ensemble overfits the training set, should you increase or decrease the learning rate?
8. Load the MNIST data (introduced in Chapter 3), and split it into a training set, a validation set, and a test set (e.g., use the first 40,000 instances for training, the next 10,000 for validation, and the last 10,000 for testing). Then train various classifiers, such as a Random Forest classifier, an Extra-Trees classifier, and an SVM. Next, try to combine them into an ensemble that outperforms them all on the validation set, using a soft or hard voting classifier. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?
9. Run the individual classifiers from the previous exercise to make predictions on the validation set, and create a new training set with the resulting predictions: each training instance is a vector containing the set of predictions from all your classifiers for an image, and the target is the image's class. Congratulations, you have just trained a blender, and together with the classifiers they form a stacking ensemble! Now let's evaluate the ensemble on the test set. For each image in the test set, make predictions with all your classifiers, then feed the predictions to the blender to get the ensemble's predictions. How does it compare to the voting classifier you trained earlier?

Solutions to these exercises are available in Appendix A.

CHAPTER 8
Dimensionality Reduction

Many Machine Learning problems involve thousands or even millions of features for each training instance. Not only does this make training extremely slow, it can also make it much harder to find a good solution, as we will see. This problem is often referred to as the *curse of dimensionality*.

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images (introduced in Chapter 3): the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information. Figure 7-6 confirms that these pixels are utterly unimportant for the classification task. Moreover, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information.

Reducing dimensionality does lose some information (just like compressing an image to JPEG can degrade its quality), so even though it will speed up training, it may also make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain. So you should first try to train your system with the original data before considering using dimensionality reduction if training is too slow. In some cases, however, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance (but in general it won't; it will just speed up training).

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization (or *DataViz*). Reducing the number of dimensions down to two

(or three) makes it possible to plot a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns, such as clusters.

In this chapter we will discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Then, we will present the two main approaches to dimensionality reduction (projection and Manifold Learning), and we will go through three of the most popular dimensionality reduction techniques: PCA, Kernel PCA, and LLE.

The Curse of Dimensionality

We are so used to living in three dimensions[1] that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our mind (see Figure 8-1), let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.

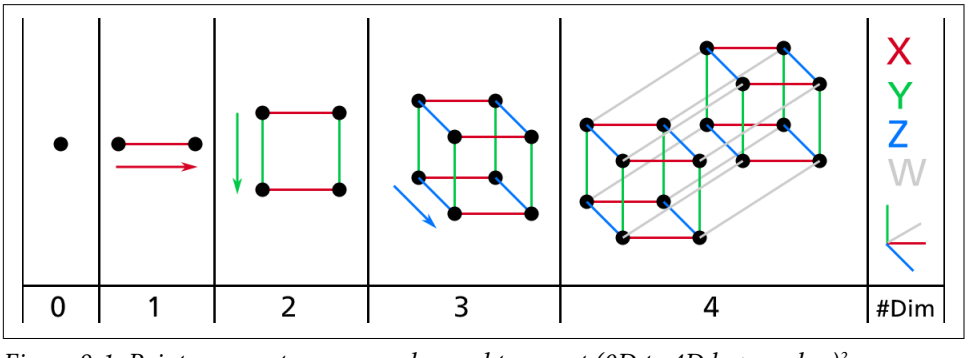

Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

It turns out that many things behave very differently in high-dimensional space. For example, if you pick a random point in a unit square (a 1×1 square), it will have only about a 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be "extreme" along any dimension). But in a 10,000-dimensional unit hypercube (a $1 \times 1 \times \cdots \times 1$ cube, with ten thousand 1s), this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border.[3]

1 Well, four dimensions if you count time, and a few more if you are a string theorist.

2 Watch a rotating tesseract projected into 3D space at *http://goo.gl/OM7ktJ*. Image by Wikipedia user Nerd-Boy1392 (Creative Commons BY-SA 3.0). Reproduced from *https://en.wikipedia.org/wiki/Tesseract*.

3 Fun fact: anyone you know is probably an extremist in at least one dimension (e.g., how much sugar they put in their coffee), if you consider enough dimensions.

Here is a more troublesome difference: if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If you pick two random points in a unit 3D cube, the average distance will be roughly 0.66. But what about two points picked randomly in a 1,000,000-dimensional hypercube? Well, the average distance, believe it or not, will be about 408.25 (roughly $\sqrt{1,000,000/6}$)! This is quite counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? This fact implies that high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. Of course, this also means that a new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations. In short, the more dimensions the training set has, the greater the risk of overfitting it.

In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features (much less than in the MNIST problem), you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

Main Approaches for Dimensionality Reduction

Before we dive into specific dimensionality reduction algorithms, let's take a look at the two main approaches to reducing dimensionality: projection and Manifold Learning.

Projection

In most real-world problems, training instances are *not* spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated (as discussed earlier for MNIST). As a result, all training instances actually lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space. This sounds very abstract, so let's look at an example. In Figure 8-2 you can see a 3D dataset represented by the circles.

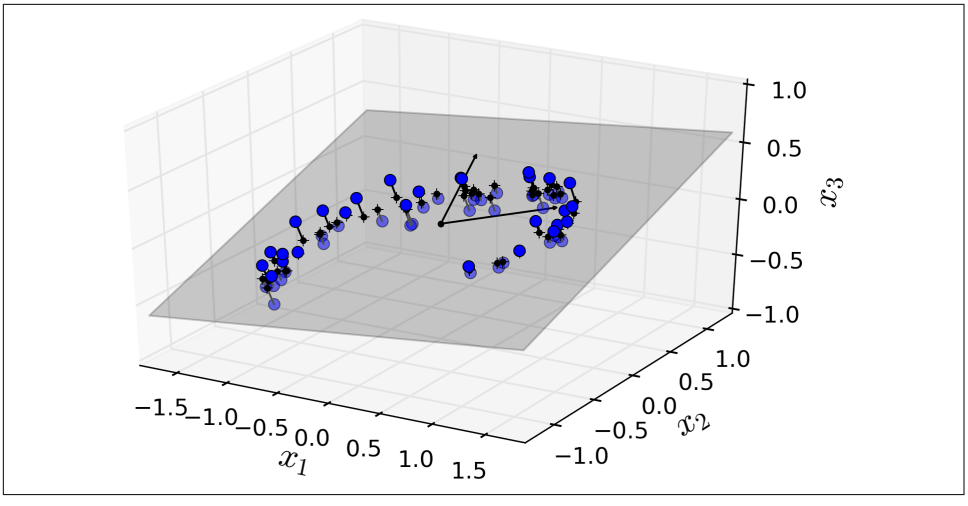

Figure 8-2. A 3D dataset lying close to a 2D subspace

Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space. Now if we project every training instance perpendicularly onto this subspace (as represented by the short lines connecting the instances to the plane), we get the new 2D dataset shown in Figure 8-3. Ta-da! We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features z_1 and z_2 (the coordinates of the projections on the plane).

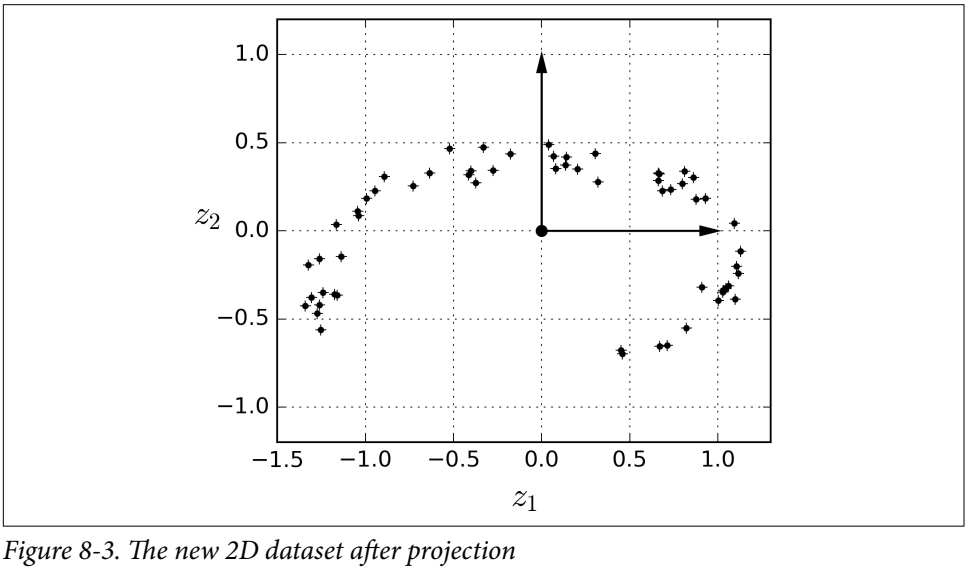

Figure 8-3. The new 2D dataset after projection

However, projection is not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the famous *Swiss roll* toy dataset represented in Figure 8-4.

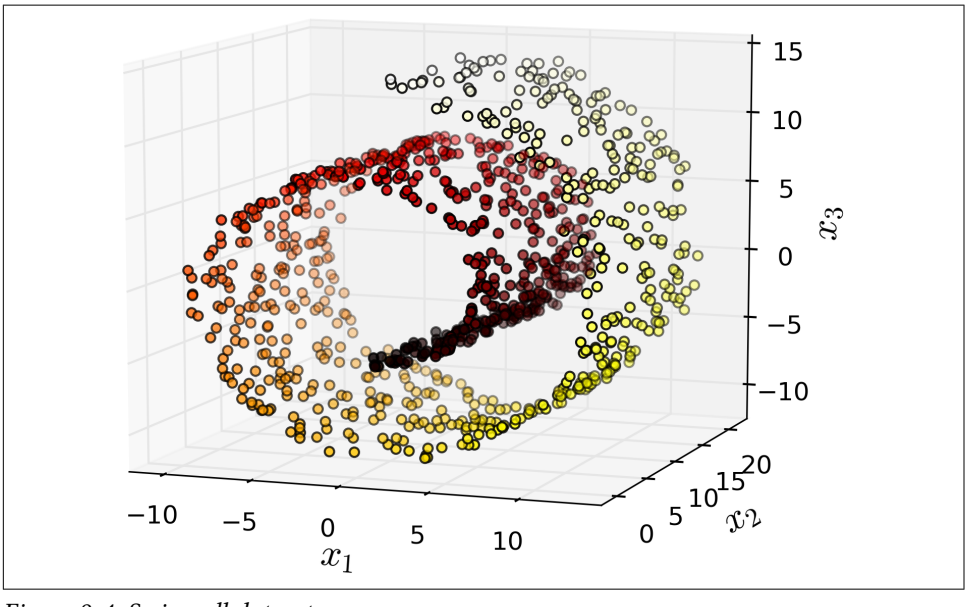

Figure 8-4. Swiss roll dataset

Simply projecting onto a plane (e.g., by dropping x_3) would squash different layers of the Swiss roll together, as shown on the left of Figure 8-5. However, what you really want is to unroll the Swiss roll to obtain the 2D dataset on the right of Figure 8-5.

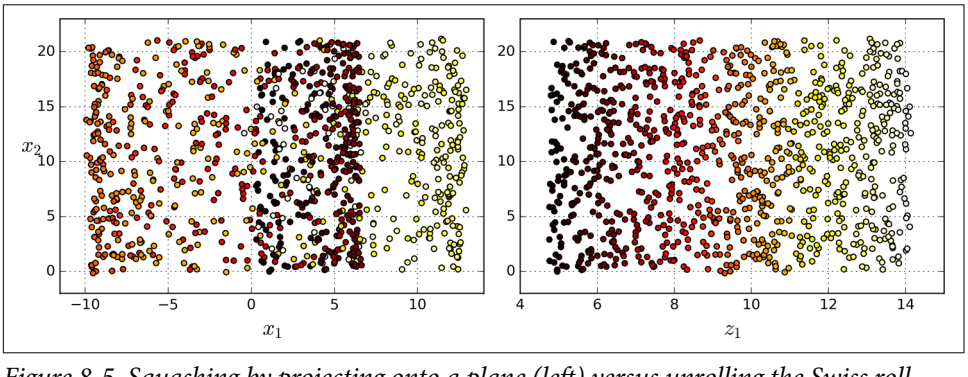

Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

Manifold Learning

The Swiss roll is an example of a 2D *manifold*. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a d-dimensional manifold is a part of an n-dimensional space (where $d < n$) that locally resembles a d-dimensional hyperplane. In the case of the Swiss roll, $d = 2$ and $n = 3$: it locally resembles a 2D plane, but it is rolled in the third dimension.

Many dimensionality reduction algorithms work by modeling the *manifold* on which the training instances lie; this is called *Manifold Learning*. It relies on the *manifold assumption*, also called the *manifold hypothesis*, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

Once again, think about the MNIST dataset: all handwritten digit images have some similarities. They are made of connected lines, the borders are white, they are more or less centered, and so on. If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits. In other words, the degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you would have if you were allowed to generate any image you wanted. These constraints tend to squeeze the dataset into a lower-dimensional manifold.

The manifold assumption is often accompanied by another implicit assumption: that the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of Figure 8-6 the Swiss roll is split into two classes: in the 3D space (on the left), the decision boundary would be fairly complex, but in the 2D unrolled manifold space (on the right), the decision boundary is a simple straight line.

However, this assumption does not always hold. For example, in the bottom row of Figure 8-6, the decision boundary is located at $x_1 = 5$. This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

In short, if you reduce the dimensionality of your training set before training a model, it will definitely speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

Hopefully you now have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds. The rest of this chapter will go through some of the most popular algorithms.

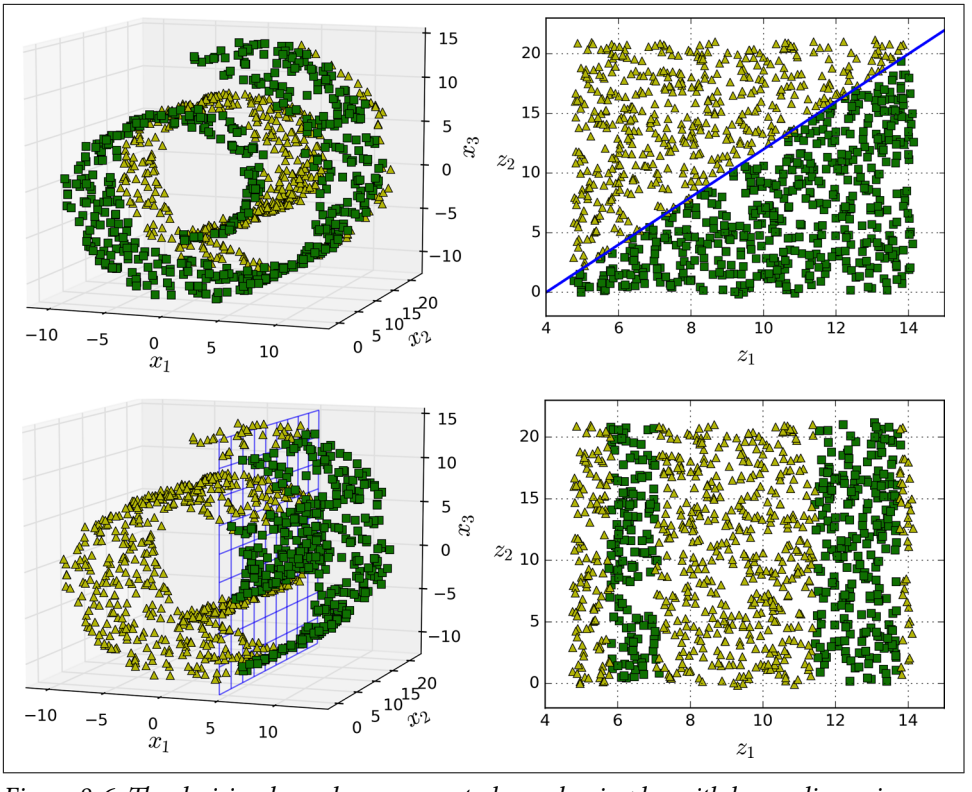

Figure 8-6. The decision boundary may not always be simpler with lower dimensions

PCA

Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.

Preserving the Variance

Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left of Figure 8-7, along with three different axes (i.e., one-dimensional hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the projection onto the solid line preserves the maximum variance, while the projection onto the dotted line preserves very little variance, and the projection onto the dashed line preserves an intermediate amount of variance.

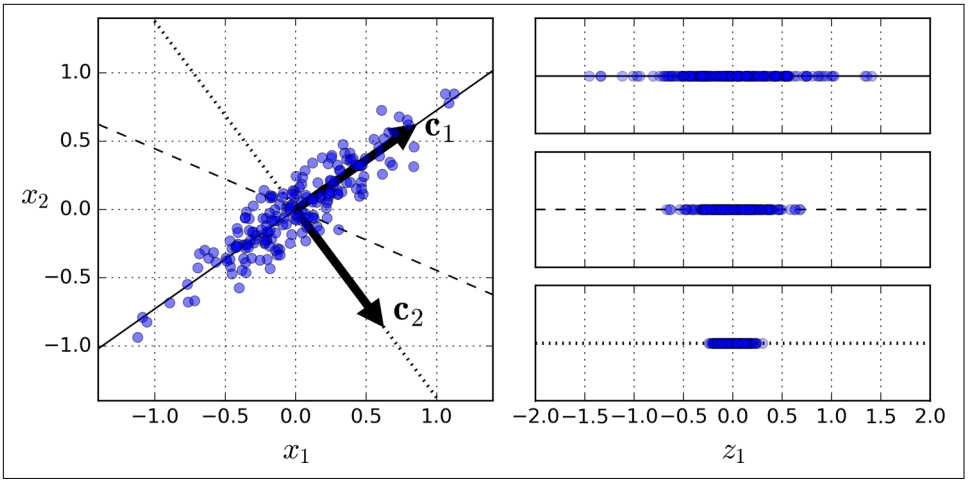

Figure 8-7. Selecting the subspace onto which to project

It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the rather simple idea behind PCA.[4]

Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. In Figure 8-7, it is the solid line. It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of remaining variance. In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset.

The unit vector that defines the i^{th} axis is called the i^{th} *principal component* (PC). In Figure 8-7, the 1^{st} PC is $\mathbf{c}_1$ and the 2^{nd} PC is $\mathbf{c}_2$. In Figure 8-2 the first two PCs are represented by the orthogonal arrows in the plane, and the third PC would be orthogonal to the plane (pointing up or down).

4 "On Lines and Planes of Closest Fit to Systems of Points in Space," K. Pearson (1901).

The direction of the principal components is not stable: if you perturb the training set slightly and run PCA again, some of the new PCs may point in the opposite direction of the original PCs. However, they will generally still lie on the same axes. In some cases, a pair of PCs may even rotate or swap, but the plane they define will generally remain the same.

So how can you find the principal components of a training set? Luckily, there is a standard matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the dot product of three matrices $\mathbf{U} \cdot \Sigma \cdot \mathbf{V}^T$, where $\mathbf{V}^T$ contains all the principal components that we are looking for, as shown in Equation 8-1.

Equation 8-1. Principal components matrix

$$\mathbf{V}^T = \begin{pmatrix} | & | & & | \\ \mathbf{c_1} & \mathbf{c_2} & \cdots & \mathbf{c_n} \\ | & | & & | \end{pmatrix}$$

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the training set, then extracts the first two PCs:

```
X_centered = X - X.mean(axis=0)
U, s, V = np.linalg.svd(X_centered)
c1 = V.T[:, 0]
c2 = V.T[:, 1]
```

PCA assumes that the dataset is centered around the origin. As we will see, Scikit-Learn's PCA classes take care of centering the data for you. However, if you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

Projecting Down to d Dimensions

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible. For example, in Figure 8-2 the 3D dataset is projected down to the 2D plane defined by the first two principal components, preserving a large part of the dataset's variance. As a result, the 2D projection looks very much like the original 3D dataset.

To project the training set onto the hyperplane, you can simply compute the dot product of the training set matrix $\mathbf{X}$ by the matrix $\mathbf{W}_d$, defined as the matrix contain-

ing the first d principal components (i.e., the matrix composed of the first d columns of $\mathbf{V}^T$), as shown in Equation 8-2.

Equation 8-2. Projecting the training set down to d dimensions

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X} \cdot \mathbf{W}_d$$

The following Python code projects the training set onto the plane defined by the first two principal components:

```
W2 = V.T[:, :2]
X2D = X_centered.dot(W2)
```

There you have it! You now know how to reduce the dimensionality of any dataset down to any number of dimensions, while preserving as much variance as possible.

Using Scikit-Learn

Scikit-Learn's `PCA` class implements PCA using SVD decomposition just like we did before. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

After fitting the `PCA` transformer to the dataset, you can access the principal components using the `components_` variable (note that it contains the PCs as horizontal vectors, so, for example, the first principal component is equal to `pca.components_.T[:, 0]`).

Explained Variance Ratio

Another very useful piece of information is the *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable. It indicates the proportion of the dataset's variance that lies along the axis of each principal component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in Figure 8-2:

```
>>> print(pca.explained_variance_ratio_)
array([ 0.84248607,  0.14631839])
```

This tells you that 84.2% of the dataset's variance lies along the first axis, and 14.6% lies along the second axis. This leaves less than 1.2% for the third axis, so it is reasonable to assume that it probably carries little information.

Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization—in that case you will generally want to reduce the dimensionality down to 2 or 3.

The following code computes PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```
pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

You could then set `n_components=d` and run PCA again. However, there is a much better option: instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between `0.0` and `1.0`, indicating the ratio of variance you wish to preserve:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X)
```

Yet another option is to plot the explained variance as a function of the number of dimensions (simply plot `cumsum`; see Figure 8-8). There will usually be an elbow in the curve, where the explained variance stops growing fast. You can think of this as the intrinsic dimensionality of the dataset. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

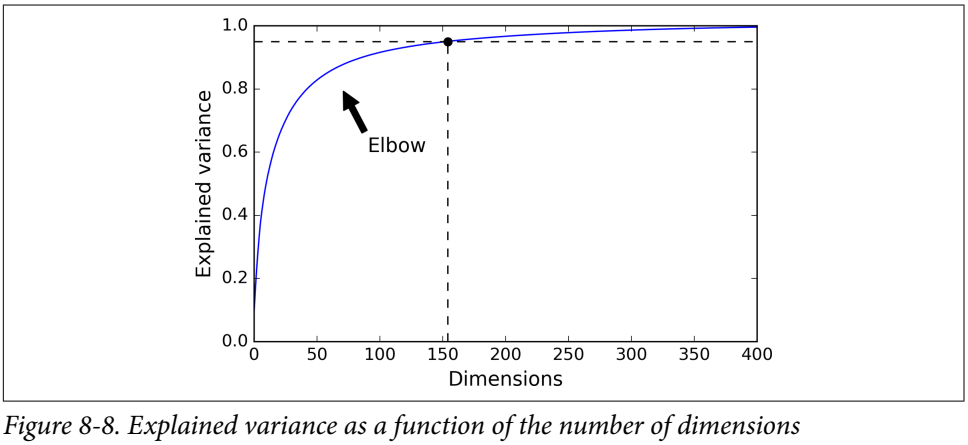

Figure 8-8. Explained variance as a function of the number of dimensions

PCA for Compression

Obviously after dimensionality reduction, the training set takes up much less space. For example, try applying PCA to the MNIST dataset while preserving 95% of its variance. You should find that each instance will have just over 150 features, instead of the original 784 features. So while most of the variance is preserved, the dataset is now less than 20% of its original size! This is a reasonable compression ratio, and you can see how this can speed up a classification algorithm (such as an SVM classifier) tremendously.

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. Of course this won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be quite close to the original data. The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*. For example, the following code compresses the MNIST dataset down to 154 dimensions, then uses the `inverse_transform()` method to decompress it back to 784 dimensions. Figure 8-9 shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact.

```
pca = PCA(n_components = 154)
X_mnist_reduced = pca.fit_transform(X_mnist)
X_mnist_recovered = pca.inverse_transform(X_mnist_reduced)
```

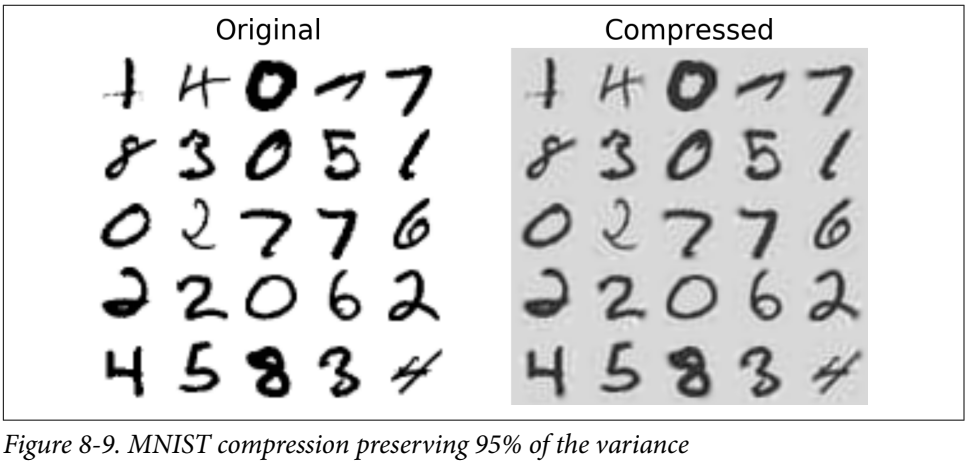

Figure 8-9. MNIST compression preserving 95% of the variance

The equation of the inverse transformation is shown in Equation 8-3.

Equation 8-3. PCA inverse transformation, back to the original number of dimensions

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}} \cdot \mathbf{W}_d^T$$

Incremental PCA

One problem with the preceding implementation of PCA is that it requires the whole training set to fit in memory in order for the SVD algorithm to run. Fortunately, *Incremental PCA* (IPCA) algorithms have been developed: you can split the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time. This is useful for large training sets, and also to apply PCA online (i.e., on the fly, as new instances arrive).

The following code splits the MNIST dataset into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to Scikit-Learn's `IncrementalPCA` class[5] to reduce the dimensionality of the MNIST dataset down to 154 dimensions (just like before). Note that you must call the `partial_fit()` method with each mini-batch rather than the `fit()` method with the whole training set:

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_mnist, n_batches):
    inc_pca.partial_fit(X_batch)

X_mnist_reduced = inc_pca.transform(X_mnist)
```

Alternatively, you can use NumPy's `memmap` class, which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory; the class loads only the data it needs in memory, when it needs it. Since the `IncrementalPCA` class uses only a small part of the array at any given time, the memory usage remains under control. This makes it possible to call the usual `fit()` method, as you can see in the following code:

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))

batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

5 Scikit-Learn uses the algorithm described in "Incremental Learning for Robust Visual Tracking," D. Ross et al. (2007).

Randomized PCA

Scikit-Learn offers yet another option to perform PCA, called *Randomized PCA*. This is a stochastic algorithm that quickly finds an approximation of the first d principal components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$, so it is dramatically faster than the previous algorithms when d is much smaller than n.

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_mnist)
```

Kernel PCA

In Chapter 5 we discussed the kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the *feature space*), enabling nonlinear classification and regression with Support Vector Machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the *original space*.

It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called *Kernel PCA* (kPCA).[6] It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

For example, the following code uses Scikit-Learn's `KernelPCA` class to perform kPCA with an RBF kernel (see Chapter 5 for more details about the RBF kernel and the other kernels):

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

Figure 8-10 shows the Swiss roll, reduced to two dimensions using a linear kernel (equivalent to simply using the `PCA` class), an RBF kernel, and a sigmoid kernel (Logistic).

6 "Kernel Principal Component Analysis," B. Schölkopf, A. Smola, K. Müller (1999).

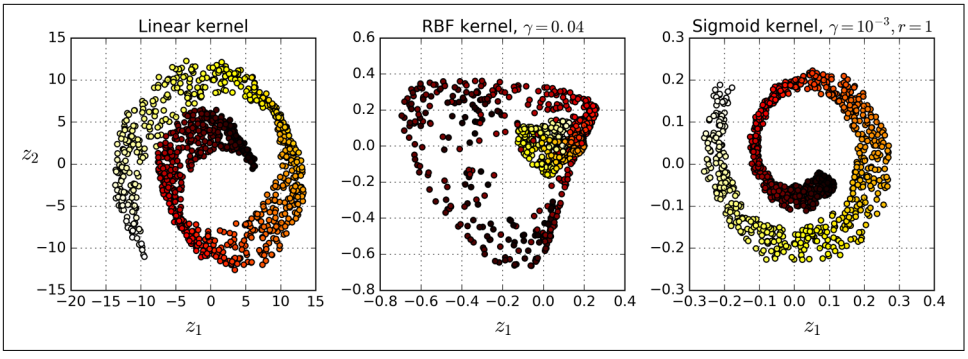

Figure 8-10. Swiss roll reduced to 2D using kPCA with various kernels

Selecting a Kernel and Tuning Hyperparameters

As kPCA is an unsupervised learning algorithm, there is no obvious performance measure to help you select the best kernel and hyperparameter values. However, dimensionality reduction is often a preparation step for a supervised learning task (e.g., classification), so you can simply use grid search to select the kernel and hyperparameters that lead to the best performance on that task. For example, the following code creates a two-step pipeline, first reducing dimensionality to two dimensions using kPCA, then applying Logistic Regression for classification. Then it uses `GridSearchCV` to find the best kernel and gamma value for kPCA in order to get the best classification accuracy at the end of the pipeline:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression())
    ])

param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

The best kernel and hyperparameters are then available through the `best_params_` variable:

```
>>> print(grid_search.best_params_)
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

Another approach, this time entirely unsupervised, is to select the kernel and hyperparameters that yield the lowest reconstruction error. However, reconstruction is not as easy as with linear PCA. Here's why. Figure 8-11 shows the original Swiss roll 3D dataset (top left), and the resulting 2D dataset after kPCA is applied using an RBF kernel (top right). Thanks to the kernel trick, this is mathematically equivalent to mapping the training set to an infinite-dimensional feature space (bottom right) using the *feature map* φ, then projecting the transformed training set down to 2D using linear PCA. Notice that if we could invert the linear PCA step for a given instance in the reduced space, the reconstructed point would lie in feature space, not in the original space (e.g., like the one represented by an x in the diagram). Since the feature space is infinite-dimensional, we cannot compute the reconstructed point, and therefore we cannot compute the true reconstruction error. Fortunately, it is possible to find a point in the original space that would map close to the reconstructed point. This is called the reconstruction *pre-image*. Once you have this pre-image, you can measure its squared distance to the original instance. You can then select the kernel and hyperparameters that minimize this reconstruction pre-image error.

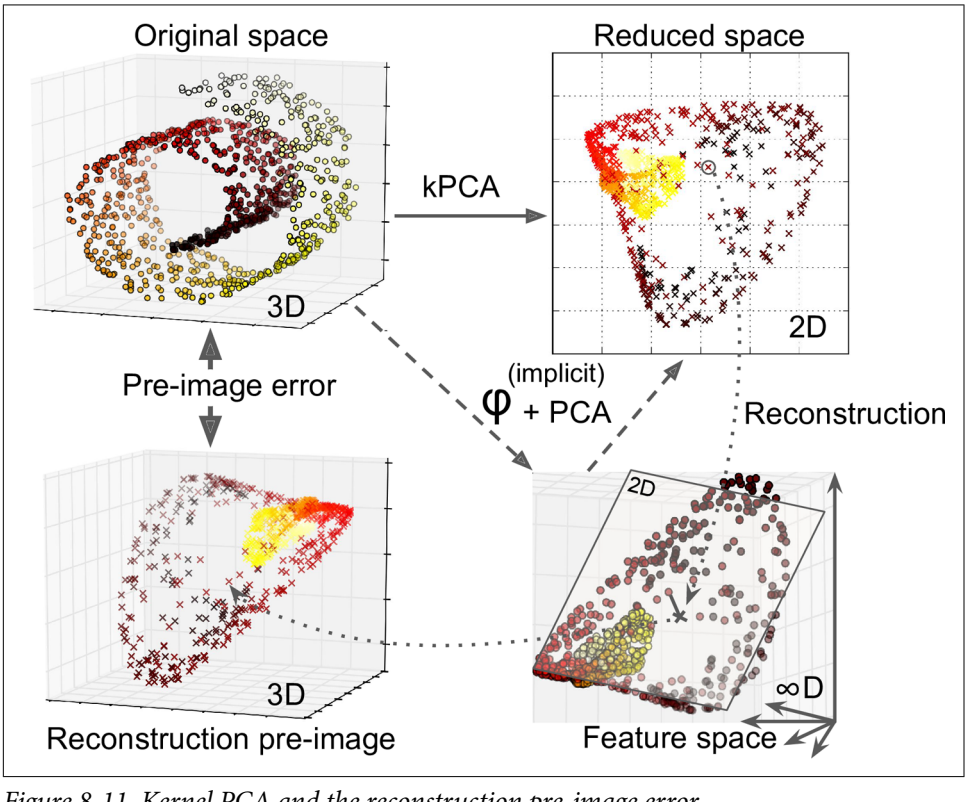

Figure 8-11. Kernel PCA and the reconstruction pre-image error

You may be wondering how to perform this reconstruction. One solution is to train a supervised regression model, with the projected instances as the training set and the original instances as the targets. Scikit-Learn will do this automatically if you set `fit_inverse_transform=True`, as shown in the following code:[7]

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                    fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```

By default, `fit_inverse_transform=False` and `KernelPCA` has no `inverse_transform()` method. This method only gets created when you set `fit_inverse_transform=True`.

You can then compute the reconstruction pre-image error:

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
32.786308795766132
```

Now you can use grid search with cross-validation to find the kernel and hyperparameters that minimize this pre-image reconstruction error.

LLE

Locally Linear Embedding (LLE)[8] is another very powerful *nonlinear dimensionality reduction* (NLDR) technique. It is a Manifold Learning technique that does not rely on projections like the previous algorithms. In a nutshell, LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly). This makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

For example, the following code uses Scikit-Learn's `LocallyLinearEmbedding` class to unroll the Swiss roll. The resulting 2D dataset is shown in Figure 8-12. As you can see, the Swiss roll is completely unrolled and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the left part of the unrolled Swiss roll is squeezed, while the right part is stretched. Nevertheless, LLE did a pretty good job at modeling the manifold.

7 Scikit-Learn uses the algorithm based on Kernel Ridge Regression described in Gokhan H. Bakır, Jason Weston, and Bernhard Scholkopf, "Learning to Find Pre-images" (Tubingen, Germany: Max Planck Institute for Biological Cybernetics, 2004).

8 "Nonlinear Dimensionality Reduction by Locally Linear Embedding," S. Roweis, L. Saul (2000).

```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

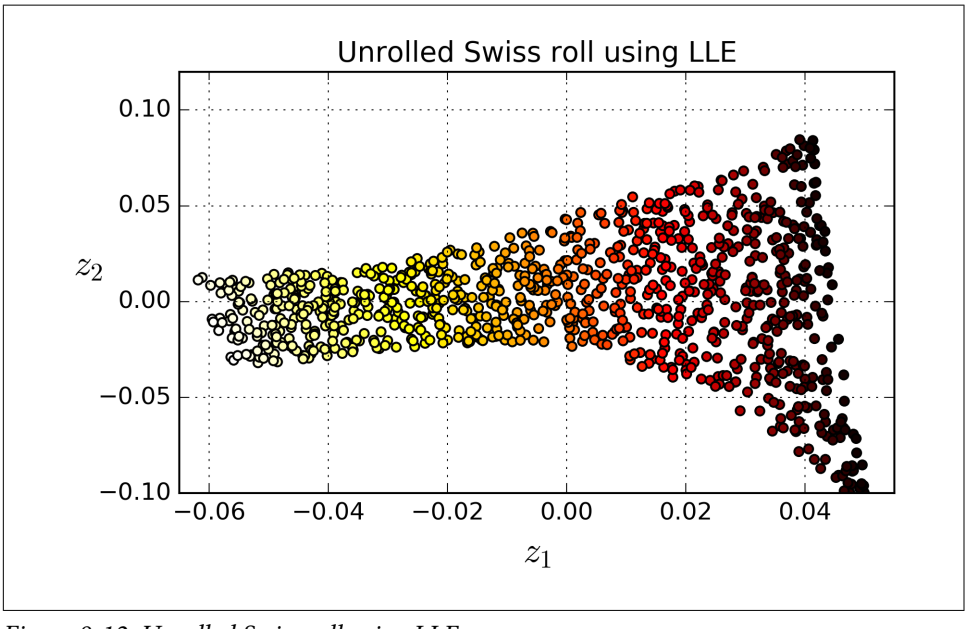

Figure 8-12. Unrolled Swiss roll using LLE

Here's how LLE works: first, for each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its k closest neighbors (in the preceding code $k = 10$), then tries to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors. More specifically, it finds the weights $w_{i,j}$ such that the squared distance between $\mathbf{x}^{(i)}$ and $\sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)}$ is as small as possible, assuming $w_{i,j} = 0$ if $\mathbf{x}^{(j)}$ is not one of the k closest neighbors of $\mathbf{x}^{(i)}$. Thus the first step of LLE is the constrained optimization problem described in Equation 8-4, where $\mathbf{W}$ is the weight matrix containing all the weights $w_{i,j}$. The second constraint simply normalizes the weights for each training instance $\mathbf{x}^{(i)}$.

Equation 8-4. LLE step 1: linearly modeling local relationships

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \| \mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j} \mathbf{x}^{(j)} \|^2$$

$$\text{subject to} \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{for } i = 1, 2, \cdots, m \end{cases}$$

After this step, the weight matrix $\widehat{\mathbf{W}}$ (containing the weights $\widehat{w}_{i,j}$) encodes the local linear relationships between the training instances. Now the second step is to map the training instances into a d-dimensional space (where $d < n$) while preserving these local relationships as much as possible. If $\mathbf{z}^{(i)}$ is the image of $\mathbf{x}^{(i)}$ in this d-dimensional space, then we want the squared distance between $\mathbf{z}^{(i)}$ and $\sum_{j=1}^{m} \widehat{w}_{i,j} \mathbf{z}^{(j)}$ to be as small as possible. This idea leads to the unconstrained optimization problem described in Equation 8-5. It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that $\mathbf{Z}$ is the matrix containing all $\mathbf{z}^{(i)}$.

Equation 8-5. LLE step 2: reducing dimensionality while preserving relationships

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \| \mathbf{z}^{(i)} - \sum_{j=1}^{m} \widehat{w}_{i,j} \mathbf{z}^{(j)} \|^2$$

Scikit-Learn's LLE implementation has the following computational complexity: $O(m \log(m) n \log(k))$ for finding the k nearest neighbors, $O(mnk^3)$ for optimizing the weights, and $O(dm^2)$ for constructing the low-dimensional representations. Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets.

Other Dimensionality Reduction Techniques

There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn. Here are some of the most popular:

- *Multidimensional Scaling* (MDS) reduces dimensionality while trying to preserve the distances between the instances (see Figure 8-13).

- *Isomap* creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances*[9] between the instances.
- *t-Distributed Stochastic Neighbor Embedding* (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space (e.g., to visualize the MNIST images in 2D).
- *Linear Discriminant Analysis* (LDA) is actually a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm such as an SVM classifier.

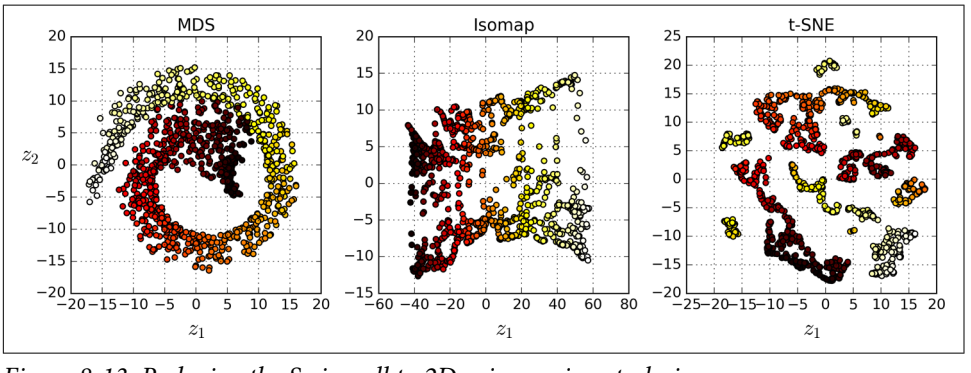

Figure 8-13. Reducing the Swiss roll to 2D using various techniques

Exercises

1. What are the main motivations for reducing a dataset's dimensionality? What are the main drawbacks?
2. What is the curse of dimensionality?
3. Once a dataset's dimensionality has been reduced, is it possible to reverse the operation? If so, how? If not, why?
4. Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?
5. Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%. How many dimensions will the resulting dataset have?

9 The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes.

6. In what cases would you use vanilla PCA, Incremental PCA, Randomized PCA, or Kernel PCA?
7. How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?
8. Does it make any sense to chain two different dimensionality reduction algorithms?
9. Load the MNIST dataset (introduced in Chapter 3) and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%. Train a new Random Forest classifier on the reduced dataset and see how long it takes. Was training much faster? Next evaluate the classifier on the test set: how does it compare to the previous classifier?
10. Use t-SNE to reduce the MNIST dataset down to two dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image's target class. Alternatively, you can write colored digits at the location of each instance, or even plot scaled-down versions of the digit images themselves (if you plot all digits, the visualization will be too cluttered, so you should either draw a random sample or plot an instance only if no other instance has already been plotted at a close distance). You should get a nice visualization with well-separated clusters of digits. Try using other dimensionality reduction algorithms such as PCA, LLE, or MDS and compare the resulting visualizations.

Solutions to these exercises are available in Appendix A.

PART II

Neural Networks and Deep Learning

CHAPTER 9

Up and Running with TensorFlow

TensorFlow is a powerful open source software library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning. Its basic principle is simple: you first define in Python a graph of computations to perform (for example, the one in Figure 9-1), and then TensorFlow takes that graph and runs it efficiently using optimized C++ code.

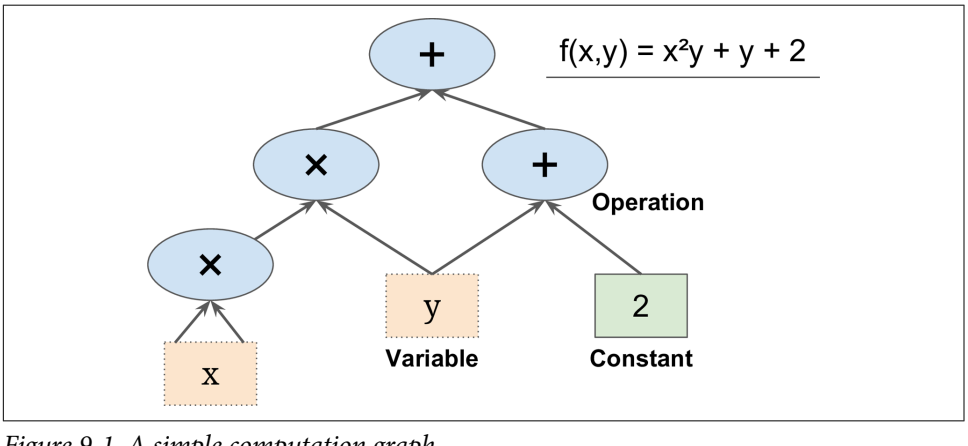

Figure 9-1. A simple computation graph

Most importantly, it is possible to break up the graph into several chunks and run them in parallel across multiple CPUs or GPUs (as shown in Figure 9-2). TensorFlow also supports distributed computing, so you can train colossal neural networks on humongous training sets in a reasonable amount of time by splitting the computations across hundreds of servers (see Chapter 12). TensorFlow can train a network with millions of parameters on a training set composed of billions of instances with millions of features each. This should come as no surprise, since TensorFlow was

developed by the Google Brain team and it powers many of Google's large-scale services, such as Google Cloud Speech, Google Photos, and Google Search.

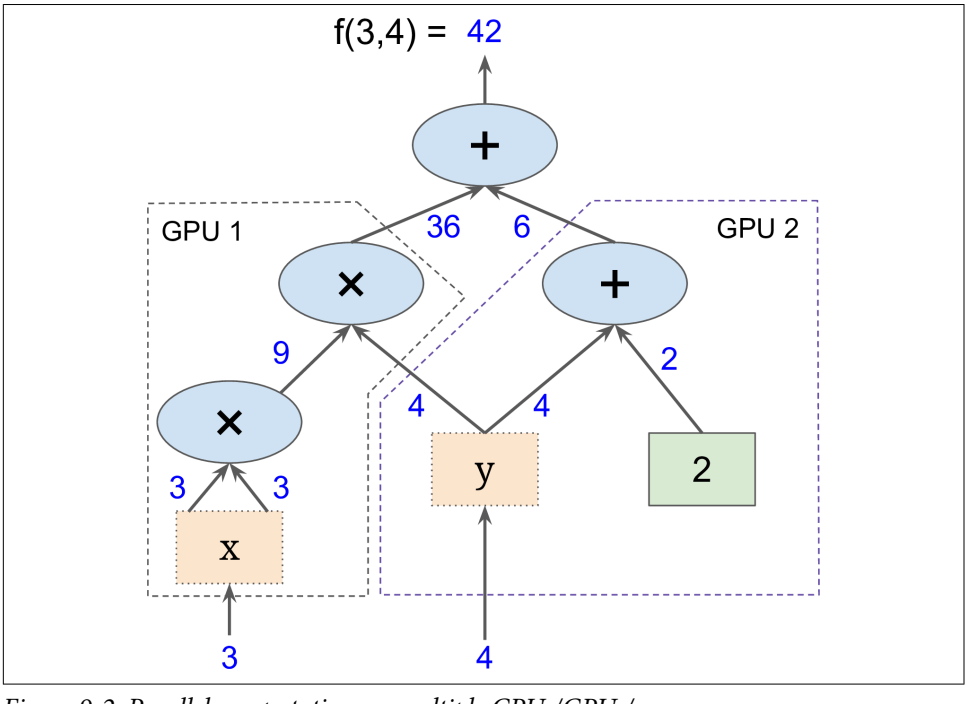

Figure 9-2. Parallel computation on multiple CPUs/GPUs/servers

When TensorFlow was open-sourced in November 2015, there were already many popular open source libraries for Deep Learning (Table 9-1 lists a few), and to be fair most of TensorFlow's features already existed in one library or another. Nevertheless, TensorFlow's clean design, scalability, flexibility,[1] and great documentation (not to mention Google's name) quickly boosted it to the top of the list. In short, TensorFlow was designed to be flexible, scalable, and production-ready, and existing frameworks arguably hit only two out of the three of these. Here are some of TensorFlow's highlights:

- It runs not only on Windows, Linux, and macOS, but also on mobile devices, including both iOS and Android.

1 TensorFlow is not limited to neural networks or even Machine Learning; you could run quantum physics simulations if you wanted.

- It provides a very simple Python API called *TF.Learn*[2] (`tensorflow.contrib.learn`), compatible with Scikit-Learn. As you will see, you can use it to train various types of neural networks in just a few lines of code. It was previously an independent project called *Scikit Flow* (or *skflow*).
- It also provides another simple API called *TF-slim* (`tensorflow.contrib.slim`) to simplify building, training, and evaluating neural networks.
- Several other high-level APIs have been built independently on top of TensorFlow, such as Keras or Pretty Tensor.
- Its main Python API offers much more flexibility (at the cost of higher complexity) to create all sorts of computations, including any neural network architecture you can think of.
- It includes highly efficient C++ implementations of many ML operations, particularly those needed to build neural networks. There is also a C++ API to define your own high-performance operations.
- It provides several advanced optimization nodes to search for the parameters that minimize a cost function. These are very easy to use since TensorFlow automatically takes care of computing the gradients of the functions you define. This is called *automatic differentiating* (or *autodiff*).
- It also comes with a great visualization tool called *TensorBoard* that allows you to browse through the computation graph, view learning curves, and more.
- Google also launched a cloud service to run TensorFlow graphs.
- Last but not least, it has a dedicated team of passionate and helpful developers, and a growing community contributing to improving it. It is one of the most popular open source projects on GitHub, and more and more great projects are being built on top of it (for examples, check out the resources page on *https://www.tensorflow.org/*, or *https://github.com/jtoy/awesome-tensorflow*). To ask technical questions, you should use *http://stackoverflow.com/* and tag your question with `"tensorflow"`. You can file bugs and feature requests through GitHub. For general discussions, join the Google group.

In this chapter, we will go through the basics of TensorFlow, from installation to creating, running, saving, and visualizing simple computational graphs. Mastering these basics is important before you build your first neural network (which we will do in the next chapter).

2 Not to be confused with the TFLearn library, which is an independent project.

Table 9-1. Open source Deep Learning libraries (not an exhaustive list)

Library	API	Platforms	Started by	Year
Caffe	Python, C++, Matlab	Linux, macOS, Windows	Y. Jia, UC Berkeley (BVLC)	2013
Deeplearning4j	Java, Scala, Clojure	Linux, macOS, Windows, Android	A. Gibson, J.Patterson	2014
H2O	Python, R	Linux, macOS, Windows	H2O.ai	2014
MXNet	Python, C++, others	Linux, macOS, Windows, iOS, Android	DMLC	2015
TensorFlow	Python, C++	Linux, macOS, Windows, iOS, Android	Google	2015
Theano	Python	Linux, macOS, iOS	University of Montreal	2010
Torch	C++, Lua	Linux, macOS, iOS, Android	R. Collobert, K. Kavukcuoglu, C. Farabet	2002

Installation

Let's get started! 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
```

Next, install TensorFlow:

```
$ pip3 install --upgrade tensorflow
```

For GPU support, you need to install `tensorflow-gpu` instead of `tensorflow`. See Chapter 12 for more details.

To test your installation, type the following command. It should output the version of TensorFlow you installed.

```
$ python3 -c 'import tensorflow; print(tensorflow.__version__)'
1.0.0
```

Creating Your First Graph and Running It in a Session

The following code creates the graph represented in Figure 9-1:

```
import tensorflow as tf

x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

That's all there is to it! The most important thing to understand is that this code does not actually perform any computation, even though it looks like it does (especially the last line). It just creates a computation graph. In fact, even the variables are not initialized yet. To evaluate this graph, you need to open a TensorFlow *session* and use it to initialize the variables and evaluate `f`. A TensorFlow session takes care of placing the operations onto *devices* such as CPUs and GPUs and running them, and it holds all the variable values.[3] The following code creates a session, initializes the variables, and evaluates, and `f` then closes the session (which frees up resources):

```
>>> sess = tf.Session()
>>> sess.run(x.initializer)
>>> sess.run(y.initializer)
>>> result = sess.run(f)
>>> print(result)
42
>>> sess.close()
```

Having to repeat `sess.run()` all the time is a bit cumbersome, but fortunately there is a better way:

```
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
```

Inside the `with` block, the session is set as the default session. Calling `x.initializer.run()` is equivalent to calling `tf.get_default_session().run(x.initializer)`, and similarly `f.eval()` is equivalent to calling `tf.get_default_session().run(f)`. This makes the code easier to read. Moreover, the session is automatically closed at the end of the block.

Instead of manually running the initializer for every single variable, you can use the `global_variables_initializer()` function. Note that it does not actually perform the initialization immediately, but rather creates a node in the graph that will initialize all variables when it is run:

```
init = tf.global_variables_initializer()  # prepare an init node

with tf.Session() as sess:
    init.run()  # actually initialize all the variables
    result = f.eval()
```

Inside Jupyter or within a Python shell you may prefer to create an `InteractiveSession`. The only difference from a regular `Session` is that when an `InteractiveSession` is created it automatically sets itself as the default session, so you don't need a

3 In distributed TensorFlow, variable values are stored on the servers instead of the session, as we will see in Chapter 12.

`with` block (but you do need to close the session manually when you are done with it):

```
>>> sess = tf.InteractiveSession()
>>> init.run()
>>> result = f.eval()
>>> print(result)
42
>>> sess.close()
```

A TensorFlow program is typically split into two parts: the first part builds a computation graph (this is called the *construction phase*), and the second part runs it (this is the *execution phase*). The construction phase typically builds a computation graph representing the ML model and the computations required to train it. The execution phase generally runs a loop that evaluates a training step repeatedly (for example, one step per mini-batch), gradually improving the model parameters. We will go through an example shortly.

Managing Graphs

Any node you create is automatically added to the default graph:

```
>>> x1 = tf.Variable(1)
>>> x1.graph is tf.get_default_graph()
True
```

In most cases this is fine, but sometimes you may want to manage multiple independent graphs. You can do this by creating a new `Graph` and temporarily making it the default graph inside a `with` block, like so:

```
>>> graph = tf.Graph()
>>> with graph.as_default():
...     x2 = tf.Variable(2)
...
>>> x2.graph is graph
True
>>> x2.graph is tf.get_default_graph()
False
```

In Jupyter (or in a Python shell), it is common to run the same commands more than once while you are experimenting. As a result, you may end up with a default graph containing many duplicate nodes. One solution is to restart the Jupyter kernel (or the Python shell), but a more convenient solution is to just reset the default graph by running `tf.reset_default_graph()`.

Lifecycle of a Node Value

When you evaluate a node, TensorFlow automatically determines the set of nodes that it depends on and it evaluates these nodes first. For example, consider the following code:

```
w = tf.constant(3)
x = w + 2
y = x + 5
z = x * 3

with tf.Session() as sess:
    print(y.eval())  # 10
    print(z.eval())  # 15
```

First, this code defines a very simple graph. Then it starts a session and runs the graph to evaluate `y`: TensorFlow automatically detects that `y` depends on `w`, which depends on `x`, so it first evaluates `w`, then `x`, then `y`, and returns the value of `y`. Finally, the code runs the graph to evaluate `z`. Once again, TensorFlow detects that it must first evaluate `w` and `x`. It is important to note that it will *not* reuse the result of the previous evaluation of `w` and `x`. In short, the preceding code evaluates `w` and `x` twice.

All node values are dropped between graph runs, except variable values, which are maintained by the session across graph runs (queues and readers also maintain some state, as we will see in Chapter 12). A variable starts its life when its initializer is run, and it ends when the session is closed.

If you want to evaluate `y` and `z` efficiently, without evaluating `w` and `x` twice as in the previous code, you must ask TensorFlow to evaluate both `y` and `z` in just one graph run, as shown in the following code:

```
with tf.Session() as sess:
    y_val, z_val = sess.run([y, z])
    print(y_val)  # 10
    print(z_val)  # 15
```

In single-process TensorFlow, multiple sessions do not share any state, even if they reuse the same graph (each session would have its own copy of every variable). In distributed TensorFlow (see Chapter 12), variable state is stored on the servers, not in the sessions, so multiple sessions can share the same variables.

Linear Regression with TensorFlow

TensorFlow operations (also called *ops* for short) can take any number of inputs and produce any number of outputs. For example, the addition and multiplication ops each take two inputs and produce one output. Constants and variables take no input

(they are called *source ops*). The inputs and outputs are multidimensional arrays, called *tensors* (hence the name "tensor flow"). Just like NumPy arrays, tensors have a type and a shape. In fact, in the Python API tensors are simply represented by NumPy ndarrays. They typically contain floats, but you can also use them to carry strings (arbitrary byte arrays).

In the examples so far, the tensors just contained a single scalar value, but you can of course perform computations on arrays of any shape. For example, the following code manipulates 2D arrays to perform Linear Regression on the California housing dataset (introduced in Chapter 2). It starts by fetching the dataset; then it adds an extra bias input feature ($x_0 = 1$) to all training instances (it does so using NumPy so it runs immediately); then it creates two TensorFlow constant nodes, `X` and `y`, to hold this data and the targets,[4] and it uses some of the matrix operations provided by TensorFlow to define `theta`. These matrix functions—`transpose()`, `matmul()`, and `matrix_inverse()`—are self-explanatory, but as usual they do not perform any computations immediately; instead, they create nodes in the graph that will perform them when the graph is run. You may recognize that the definition of `theta` corresponds to the Normal Equation ($\hat{\theta} = \mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$; see Chapter 4). Finally, the code creates a session and uses it to evaluate `theta`.

```
import numpy as np
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
    theta_value = theta.eval()
```

The main benefit of this code versus computing the Normal Equation directly using NumPy is that TensorFlow will automatically run this on your GPU card if you have one (provided you installed TensorFlow with GPU support, of course; see Chapter 12 for more details).

4 Note that `housing.target` is a 1D array, but we need to reshape it to a column vector to compute `theta`. Recall that NumPy's `reshape()` function accepts –1 (meaning "unspecified") for one of the dimensions: that dimension will be computed based on the array's length and the remaining dimensions.

Implementing Gradient Descent

Let's try using Batch Gradient Descent (introduced in Chapter 4) instead of the Normal Equation. First we will do this by manually computing the gradients, then we will use TensorFlow's autodiff feature to let TensorFlow compute the gradients automatically, and finally we will use a couple of TensorFlow's out-of-the-box optimizers.

When using Gradient Descent, remember that it is important to first normalize the input feature vectors, or else training may be much slower. You can do this using TensorFlow, NumPy, Scikit-Learn's `StandardScaler`, or any other solution you prefer. The following code assumes that this normalization has already been done.

Manually Computing the Gradients

The following code should be fairly self-explanatory, except for a few new elements:

- The `random_uniform()` function creates a node in the graph that will generate a tensor containing random values, given its shape and value range, much like NumPy's `rand()` function.
- The `assign()` function creates a node that will assign a new value to a variable. In this case, it implements the Batch Gradient Descent step $\theta^{(\text{next step})} = \theta - \eta\nabla_\theta \text{MSE}(\theta)$.
- The main loop executes the training step over and over again (`n_epochs` times), and every 100 iterations it prints out the current Mean Squared Error (`mse`). You should see the MSE go down at every iteration.

```
n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
```

```
            if epoch % 100 == 0:
                print("Epoch", epoch, "MSE =", mse.eval())
            sess.run(training_op)

        best_theta = theta.eval()
```

Using autodiff

The preceding code works fine, but it requires mathematically deriving the gradients from the cost function (MSE). In the case of Linear Regression, it is reasonably easy, but if you had to do this with deep neural networks you would get quite a headache: it would be tedious and error-prone. You could use *symbolic differentiation* to automatically find the equations for the partial derivatives for you, but the resulting code would not necessarily be very efficient.

To understand why, consider the function $f(x)= \exp(\exp(\exp(x)))$. If you know calculus, you can figure out its derivative $f'(x) = \exp(x) \times \exp(\exp(x)) \times \exp(\exp(\exp(x)))$. If you code $f(x)$ and $f'(x)$ separately and exactly as they appear, your code will not be as efficient as it could be. A more efficient solution would be to write a function that first computes $\exp(x)$, then $\exp(\exp(x))$, then $\exp(\exp(\exp(x)))$, and returns all three. This gives you $f(x)$ directly (the third term), and if you need the derivative you can just multiply all three terms and you are done. With the naïve approach you would have had to call the `exp` function nine times to compute both $f(x)$ and $f'(x)$. With this approach you just need to call it three times.

It gets worse when your function is defined by some arbitrary code. Can you find the equation (or the code) to compute the partial derivatives of the following function? Hint: don't even try.

```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

Fortunately, TensorFlow's autodiff feature comes to the rescue: it can automatically and efficiently compute the gradients for you. Simply replace the `gradients = ...` line in the Gradient Descent code in the previous section with the following line, and the code will continue to work just fine:

```
gradients = tf.gradients(mse, [theta])[0]
```

The `gradients()` function takes an op (in this case `mse`) and a list of variables (in this case just `theta`), and it creates a list of ops (one per variable) to compute the gradients of the op with regards to each variable. So the `gradients` node will compute the gradient vector of the MSE with regards to `theta`.

There are four main approaches to computing gradients automatically. They are summarized in Table 9-2. TensorFlow uses *reverse-mode autodiff*, which is perfect (efficient and accurate) when there are many inputs and few outputs, as is often the case in neural networks. It computes all the partial derivatives of the outputs with regards to all the inputs in just $n_{\text{outputs}} + 1$ graph traversals.

Table 9-2. Main solutions to compute gradients automatically

Technique	Nb of graph traversals to compute all gradients	Accuracy	Supports arbitrary code	Comment
Numerical differentiation	$n_{\text{inputs}} + 1$	Low	Yes	Trivial to implement
Symbolic differentiation	N/A	High	No	Builds a very different graph
Forward-mode autodiff	n_{inputs}	High	Yes	Uses *dual numbers*
Reverse-mode autodiff	$n_{\text{outputs}} + 1$	High	Yes	Implemented by TensorFlow

If you are interested in how this magic works, check out Appendix D.

Using an Optimizer

So TensorFlow computes the gradients for you. But it gets even easier: it also provides a number of optimizers out of the box, including a Gradient Descent optimizer. You can simply replace the preceding `gradients = ...` and `training_op = ...` lines with the following code, and once again everything will just work fine:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

If you want to use a different type of optimizer, you just need to change one line. For example, you can use a momentum optimizer (which often converges much faster than Gradient Descent; see Chapter 11) by defining the optimizer like this:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

Feeding Data to the Training Algorithm

Let's try to modify the previous code to implement Mini-batch Gradient Descent. For this, we need a way to replace `X` and `y` at every iteration with the next mini-batch. The simplest way to do this is to use placeholder nodes. These nodes are special because they don't actually perform any computation, they just output the data you tell them to output at runtime. They are typically used to pass the training data to TensorFlow during training. If you don't specify a value at runtime for a placeholder, you get an exception.

To create a placeholder node, you must call the `placeholder()` function and specify the output tensor's data type. Optionally, you can also specify its shape, if you want to

enforce it. If you specify `None` for a dimension, it means "any size." For example, the following code creates a placeholder node `A`, and also a node `B = A + 5`. When we evaluate `B`, we pass a `feed_dict` to the `eval()` method that specifies the value of `A`. Note that `A` must have rank 2 (i.e., it must be two-dimensional) and there must be three columns (or else an exception is raised), but it can have any number of rows.

```
>>> A = tf.placeholder(tf.float32, shape=(None, 3))
>>> B = A + 5
>>> with tf.Session() as sess:
...     B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
...     B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
...
>>> print(B_val_1)
[[ 6.  7.  8.]]
>>> print(B_val_2)
[[  9.  10.  11.]
 [ 12.  13.  14.]]
```

You can actually feed the output of *any* operations, not just placeholders. In this case TensorFlow does not try to evaluate these operations; it uses the values you feed it.

To implement Mini-batch Gradient Descent, we only need to tweak the existing code slightly. First change the definition of `X` and `y` in the construction phase to make them placeholder nodes:

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
```

Then define the batch size and compute the total number of batches:

```
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

Finally, in the execution phase, fetch the mini-batches one by one, then provide the value of `X` and `y` via the `feed_dict` parameter when evaluating a node that depends on either of them.

```
def fetch_batch(epoch, batch_index, batch_size):
    [...] # load the data from disk
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
```

```
best_theta = theta.eval()
```

We don't need to pass the value of `X` and `y` when evaluating `theta` since it does not depend on either of them.

Saving and Restoring Models

Once you have trained your model, you should save its parameters to disk so you can come back to it whenever you want, use it in another program, compare it to other models, and so on. Moreover, you probably want to save checkpoints at regular intervals during training so that if your computer crashes during training you can continue from the last checkpoint rather than start over from scratch.

TensorFlow makes saving and restoring a model very easy. Just create a `Saver` node at the end of the construction phase (after all variable nodes are created); then, in the execution phase, just call its `save()` method whenever you want to save the model, passing it the session and path of the checkpoint file:

```
[...]
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
[...]
init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:  # checkpoint every 100 epochs
            save_path = saver.save(sess, "/tmp/my_model.ckpt")

        sess.run(training_op)

    best_theta = theta.eval()
    save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

Restoring a model is just as easy: you create a `Saver` at the end of the construction phase just like before, but then at the beginning of the execution phase, instead of initializing the variables using the `init` node, you call the `restore()` method of the `Saver` object:

```
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    [...]
```

By default a `Saver` saves and restores all variables under their own name, but if you need more control, you can specify which variables to save or restore, and what names to use. For example, the following `Saver` will save or restore only the `theta` variable under the name `weights`:

```
saver = tf.train.Saver({"weights": theta})
```

Visualizing the Graph and Training Curves Using TensorBoard

So now we have a computation graph that trains a Linear Regression model using Mini-batch Gradient Descent, and we are saving checkpoints at regular intervals. Sounds sophisticated, doesn't it? However, we are still relying on the `print()` function to visualize progress during training. There is a better way: enter TensorBoard. If you feed it some training stats, it will display nice interactive visualizations of these stats in your web browser (e.g., learning curves). You can also provide it the graph's definition and it will give you a great interface to browse through it. This is very useful to identify errors in the graph, to find bottlenecks, and so on.

The first step is to tweak your program a bit so it writes the graph definition and some training stats—for example, the training error (MSE)—to a log directory that TensorBoard will read from. You need to use a different log directory every time you run your program, or else TensorBoard will merge stats from different runs, which will mess up the visualizations. The simplest solution for this is to include a timestamp in the log directory name. Add the following code at the beginning of the program:

```
from datetime import datetime

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)
```

Next, add the following code at the very end of the construction phase:

```
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

The first line creates a node in the graph that will evaluate the MSE value and write it to a TensorBoard-compatible binary log string called a *summary*. The second line creates a `FileWriter` that you will use to write summaries to logfiles in the log directory. The first parameter indicates the path of the log directory (in this case something like *tf_logs/run-20160906091959/*, relative to the current directory). The second (optional) parameter is the graph you want to visualize. Upon creation, the `FileWriter` creates the log directory if it does not already exist (and its parent directories if needed), and writes the graph definition in a binary logfile called an *events file*.

Next you need to update the execution phase to evaluate the `mse_summary` node regularly during training (e.g., every 10 mini-batches). This will output a summary that you can then write to the events file using the `file_writer`. Here is the updated code:

```
[...]
for batch_index in range(n_batches):
    X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
    if batch_index % 10 == 0:
        summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
        step = epoch * n_batches + batch_index
        file_writer.add_summary(summary_str, step)
    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
[...]
```

Avoid logging training stats at every single training step, as this would significantly slow down training.

Finally, you want to close the `FileWriter` at the end of the program:

```
file_writer.close()
```

Now run this program: it will create the log directory and write an events file in this directory, containing both the graph definition and the MSE values. Open up a shell and go to your working directory, then type **`ls -l tf_logs/run*`** to list the contents of the log directory:

```
$ cd $ML_PATH                # Your ML working directory (e.g., $HOME/ml)
$ ls -l tf_logs/run*
total 40
-rw-r--r-- 1 ageron staff 18620 Sep 6 11:10 events.out.tfevents.1472553182.mymac
```

If you run the program a second time, you should see a second directory in the *tf_logs/* directory:

```
$ ls -l tf_logs/
total 0
drwxr-xr-x  3 ageron  staff  102 Sep  6 10:07 run-20160906091959
drwxr-xr-x  3 ageron  staff  102 Sep  6 10:22 run-20160906092202
```

Great! Now it's time to fire up the TensorBoard server. You need to activate your virtualenv environment if you created one, then start the server by running the `tensorboard` command, pointing it to the root log directory. This starts the TensorBoard web server, listening on port 6006 (which is "goog" written upside down):

```
$ source env/bin/activate
$ tensorboard --logdir tf_logs/
Starting TensorBoard  on port 6006
(You can navigate to http://0.0.0.0:6006)
```

Next open a browser and go to *http://0.0.0.0:6006/* (or *http://localhost:6006/*). Welcome to TensorBoard! In the Events tab you should see MSE on the right. If you click on it, you will see a plot of the MSE during training, for both runs (Figure 9-3). You can check or uncheck the runs you want to see, zoom in or out, hover over the curve to get details, and so on.

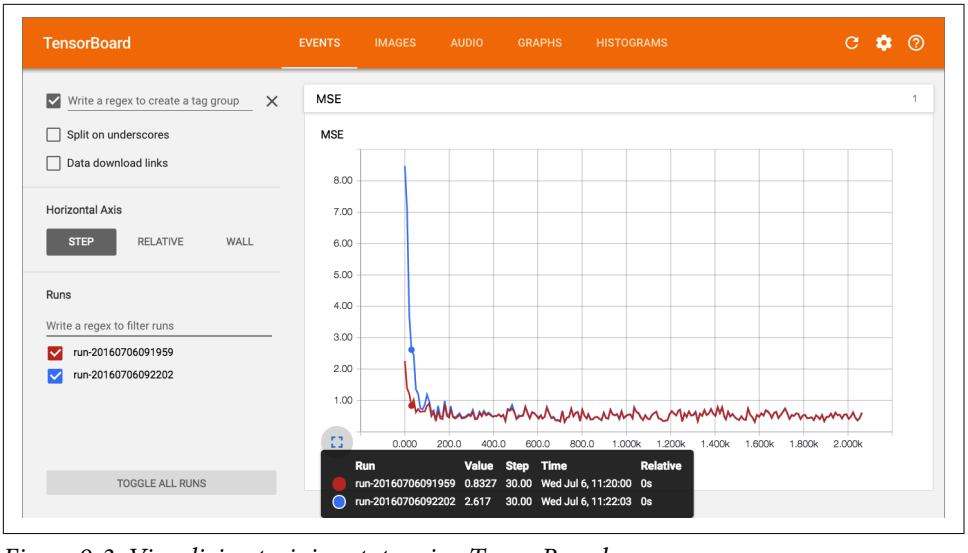

Figure 9-3. Visualizing training stats using TensorBoard

Now click on the Graphs tab. You should see the graph shown in Figure 9-4.

To reduce clutter, the nodes that have many *edges* (i.e., connections to other nodes) are separated out to an auxiliary area on the right (you can move a node back and forth between the main graph and the auxiliary area by right-clicking on it). Some parts of the graph are also collapsed by default. For example, try hovering over the `gradients` node, then click on the ⊕ icon to expand this subgraph. Next, in this subgraph, try expanding the `mse_grad` subgraph.

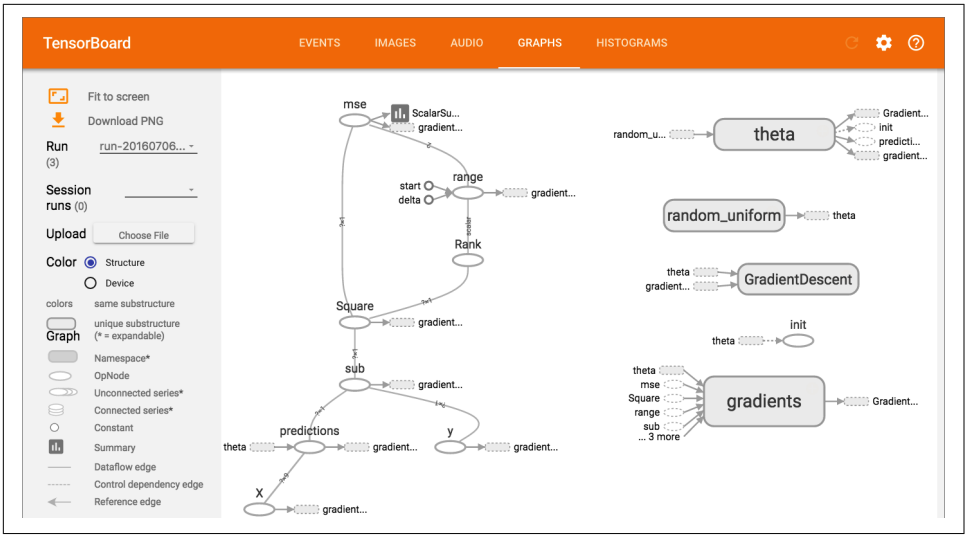

Figure 9-4. Visualizing the graph using TensorBoard

If you want to take a peek at the graph directly within Jupyter, you can use the `show_graph()` function available in the notebook for this chapter. It was originally written by A. Mordvintsev in his great deepdream tutorial notebook. Another option is to install E. Jang's TensorFlow debugger tool which includes a Jupyter extension for graph visualization (and more).

Name Scopes

When dealing with more complex models such as neural networks, the graph can easily become cluttered with thousands of nodes. To avoid this, you can create *name scopes* to group related nodes. For example, let's modify the previous code to define the `error` and `mse` ops within a name scope called `"loss"`:

```
with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

The name of each op defined within the scope is now prefixed with `"loss/"`:

```
>>> print(error.op.name)
loss/sub
>>> print(mse.op.name)
loss/mse
```

In TensorBoard, the `mse` and `error` nodes now appear inside the `loss` namespace, which appears collapsed by default (Figure 9-5).

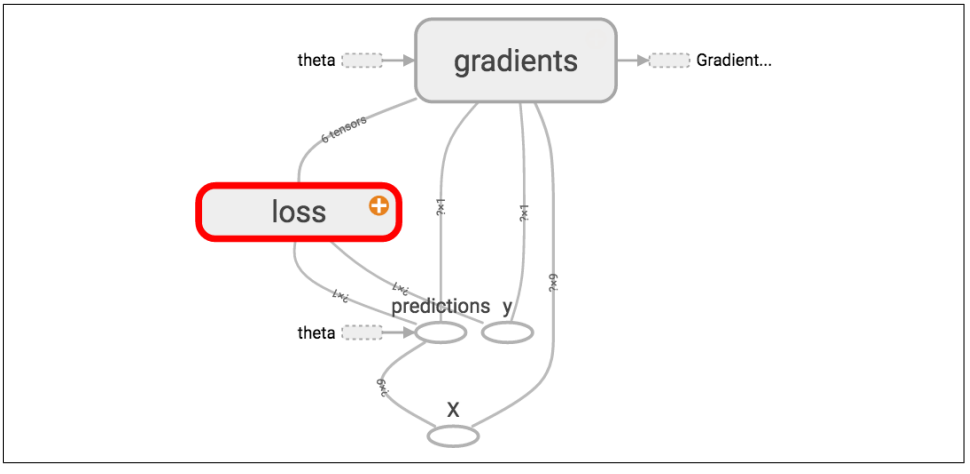

Figure 9-5. A collapsed namescope in TensorBoard

Modularity

Suppose you want to create a graph that adds the output of two *rectified linear units* (ReLU). A ReLU computes a linear function of the inputs, and outputs the result if it is positive, and 0 otherwise, as shown in Equation 9-1.

Equation 9-1. Rectified linear unit

$$h_{\mathbf{w},b}(\mathbf{X}) = \max(\mathbf{X} \cdot \mathbf{w} + b, 0)$$

The following code does the job, but it's quite repetitive:

```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")

w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")

z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")

relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z1, 0., name="relu2")

output = tf.add(relu1, relu2, name="output")
```

Such repetitive code is hard to maintain and error-prone (in fact, this code contains a cut-and-paste error; did you spot it?). It would become even worse if you wanted to

add a few more ReLUs. Fortunately, TensorFlow lets you stay DRY (Don't Repeat Yourself): simply create a function to build a ReLU. The following code creates five ReLUs and outputs their sum (note that `add_n()` creates an operation that will compute the sum of a list of tensors):

```
def relu(X):
    w_shape = (int(X.get_shape()[1]), 1)
    w = tf.Variable(tf.random_normal(w_shape), name="weights")
    b = tf.Variable(0.0, name="bias")
    z = tf.add(tf.matmul(X, w), b, name="z")
    return tf.maximum(z, 0., name="relu")

n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")
```

Note that when you create a node, TensorFlow checks whether its name already exists, and if it does it appends an underscore followed by an index to make the name unique. So the first ReLU contains nodes named `"weights"`, `"bias"`, `"z"`, and `"relu"` (plus many more nodes with their default name, such as `"MatMul"`); the second ReLU contains nodes named `"weights_1"`, `"bias_1"`, and so on; the third ReLU contains nodes named `"weights_2"`, `"bias_2"`, and so on. TensorBoard identifies such series and collapses them together to reduce clutter (as you can see in Figure 9-6).

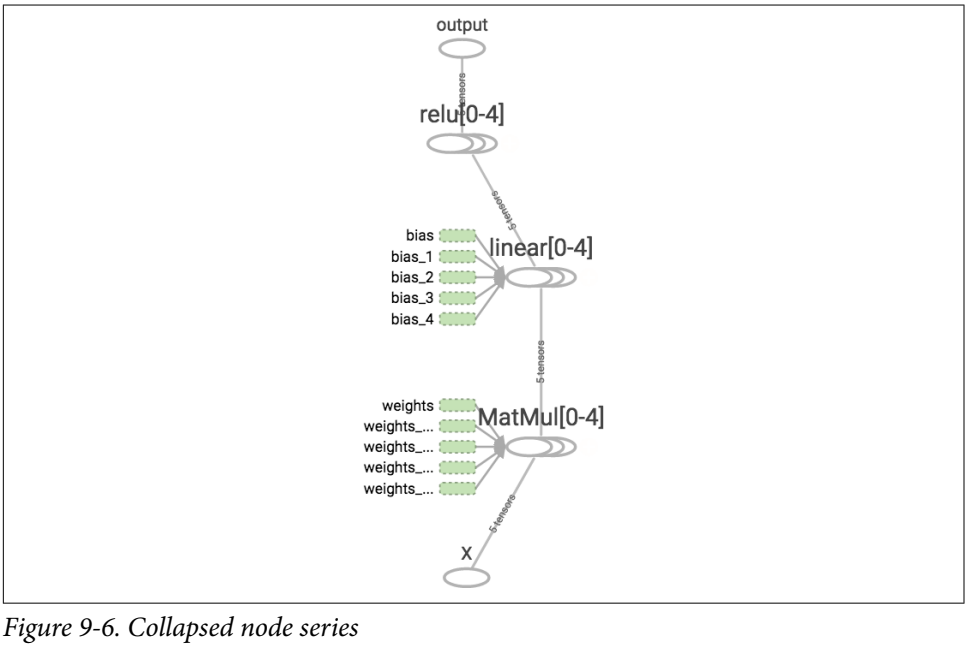

Figure 9-6. Collapsed node series

Using name scopes, you can make the graph much clearer. Simply move all the content of the `relu()` function inside a name scope. Figure 9-7 shows the resulting graph. Notice that TensorFlow also gives the name scopes unique names by appending `_1`, `_2`, and so on.

```
def relu(X):
    with tf.name_scope("relu"):
        [...]
```

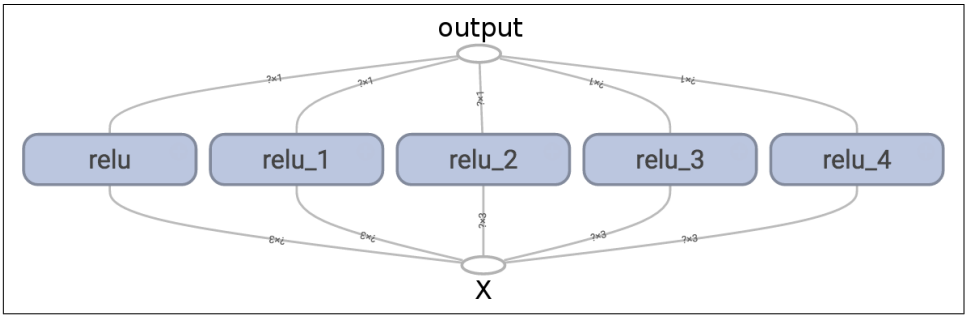

Figure 9-7. A clearer graph using name-scoped units

Sharing Variables

If you want to share a variable between various components of your graph, one simple option is to create it first, then pass it as a parameter to the functions that need it. For example, suppose you want to control the ReLU threshold (currently hardcoded to 0) using a shared `threshold` variable for all ReLUs. You could just create that variable first, and then pass it to the `relu()` function:

```
def relu(X, threshold):
    with tf.name_scope("relu"):
        [...]
        return tf.maximum(z, threshold, name="max")

threshold = tf.Variable(0.0, name="threshold")
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X, threshold) for i in range(5)]
output = tf.add_n(relus, name="output")
```

This works fine: now you can control the threshold for all ReLUs using the `threshold` variable. However, if there are many shared parameters such as this one, it will be painful to have to pass them around as parameters all the time. Many people create a Python dictionary containing all the variables in their model, and pass it around to every function. Others create a class for each module (e.g., a `ReLU` class using class variables to handle the shared parameter). Yet another option is to set the shared variable as an attribute of the `relu()` function upon the first call, like so:

```
def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        [...]
        return tf.maximum(z, relu.threshold, name="max")
```

TensorFlow offers another option, which may lead to slightly cleaner and more modular code than the previous solutions.[5] This solution is a bit tricky to understand at first, but since it is used a lot in TensorFlow it is worth going into a bit of detail. The idea is to use the `get_variable()` function to create the shared variable if it does not exist yet, or reuse it if it already exists. The desired behavior (creating or reusing) is controlled by an attribute of the current `variable_scope()`. For example, the following code will create a variable named `"relu/threshold"` (as a scalar, since `shape=()`, and using `0.0` as the initial value):

```
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
```

Note that if the variable has already been created by an earlier call to `get_variable()`, this code will raise an exception. This behavior prevents reusing variables by mistake. If you want to reuse a variable, you need to explicitly say so by setting the variable scope's `reuse` attribute to `True` (in which case you don't have to specify the shape or the initializer):

```
with tf.variable_scope("relu", reuse=True):
    threshold = tf.get_variable("threshold")
```

This code will fetch the existing `"relu/threshold"` variable, or raise an exception if it does not exist or if it was not created using `get_variable()`. Alternatively, you can set the `reuse` attribute to `True` inside the block by calling the scope's `reuse_variables()` method:

```
with tf.variable_scope("relu") as scope:
    scope.reuse_variables()
    threshold = tf.get_variable("threshold")
```

Once `reuse` is set to `True`, it cannot be set back to `False` within the block. Moreover, if you define other variable scopes inside this one, they will automatically inherit `reuse=True`. Lastly, only variables created by `get_variable()` can be reused this way.

5 Creating a `ReLU` class is arguably the cleanest option, but it is rather heavyweight.

Now you have all the pieces you need to make the `relu()` function access the `threshold` variable without having to pass it as a parameter:

```
def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold")  # reuse existing variable
        [...]
        return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"):  # create the variable
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
relus = [relu(X) for relu_index in range(5)]
output = tf.add_n(relus, name="output")
```

This code first defines the `relu()` function, then creates the `relu/threshold` variable (as a scalar that will later be initialized to `0.0`) and builds five ReLUs by calling the `relu()` function. The `relu()` function reuses the `relu/threshold` variable, and creates the other ReLU nodes.

Variables created using `get_variable()` are always named using the name of their `variable_scope` as a prefix (e.g., `"relu/threshold"`), but for all other nodes (including variables created with `tf.Variable()`) the variable scope acts like a new name scope. In particular, if a name scope with an identical name was already created, then a suffix is added to make the name unique. For example, all nodes created in the preceding code (except the `threshold` variable) have a name prefixed with `"relu_1/"` to `"relu_5/"`, as shown in Figure 9-8.

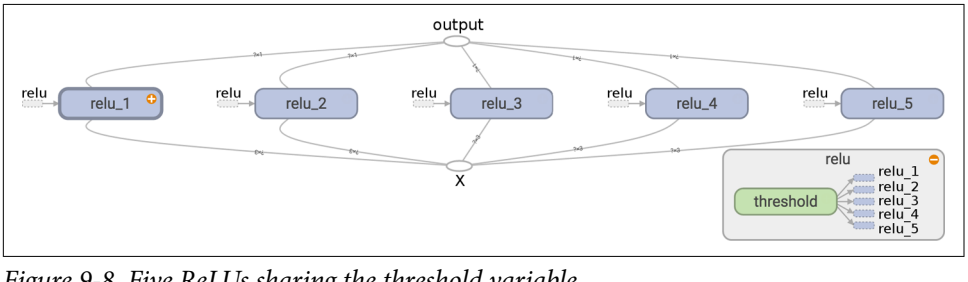

Figure 9-8. Five ReLUs sharing the threshold variable

It is somewhat unfortunate that the `threshold` variable must be defined outside the `relu()` function, where all the rest of the ReLU code resides. To fix this, the following code creates the `threshold` variable within the `relu()` function upon the first call, then reuses it in subsequent calls. Now the `relu()` function does not have to worry about name scopes or variable sharing: it just calls `get_variable()`, which will create

or reuse the `threshold` variable (it does not need to know which is the case). The rest of the code calls `relu()` five times, making sure to set `reuse=False` on the first call, and `reuse=True` for the other calls.

```
def relu(X):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    [...]
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = []
for relu_index in range(5):
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:
        relus.append(relu(X))
output = tf.add_n(relus, name="output")
```

The resulting graph is slightly different than before, since the shared variable lives within the first ReLU (see Figure 9-9).

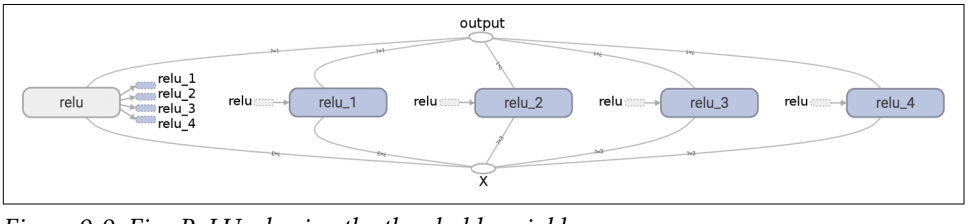

Figure 9-9. Five ReLUs sharing the threshold variable

This concludes this introduction to TensorFlow. We will discuss more advanced topics as we go through the following chapters, in particular many operations related to deep neural networks, convolutional neural networks, and recurrent neural networks as well as how to scale up with TensorFlow using multithreading, queues, multiple GPUs, and multiple servers.

Exercises

1. What are the main benefits of creating a computation graph rather than directly executing the computations? What are the main drawbacks?
2. Is the statement `a_val = a.eval(session=sess)` equivalent to `a_val = sess.run(a)`?
3. Is the statement `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` equivalent to `a_val, b_val = sess.run([a, b])`?
4. Can you run two graphs in the same session?

5. If you create a graph `g` containing a variable `w`, then start two threads and open a session in each thread, both using the same graph `g`, will each session have its own copy of the variable `w` or will it be shared?
6. When is a variable initialized? When is it destroyed?
7. What is the difference between a placeholder and a variable?
8. What happens when you run the graph to evaluate an operation that depends on a placeholder but you don't feed its value? What happens if the operation does not depend on the placeholder?
9. When you run a graph, can you feed the output value of any operation, or just the value of placeholders?
10. How can you set a variable to any value you want (during the execution phase)?
11. How many times does reverse-mode autodiff need to traverse the graph in order to compute the gradients of the cost function with regards to 10 variables? What about forward-mode autodiff? And symbolic differentiation?
12. Implement Logistic Regression with Mini-batch Gradient Descent using TensorFlow. Train it and evaluate it on the moons dataset (introduced in Chapter 5). Try adding all the bells and whistles:
    - Define the graph within a `logistic_regression()` function that can be reused easily.
    - Save checkpoints using a `Saver` at regular intervals during training, and save the final model at the end of training.
    - Restore the last checkpoint upon startup if training was interrupted.
    - Define the graph using nice scopes so the graph looks good in TensorBoard.
    - Add summaries to visualize the learning curves in TensorBoard.
    - Try tweaking some hyperparameters such as the learning rate or the mini-batch size and look at the shape of the learning curve.

Solutions to these exercises are available in Appendix A.

CHAPTER 10
Introduction to Artificial Neural Networks

Birds inspired us to fly, burdock plants inspired velcro, and nature has inspired many other inventions. It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the key idea that inspired *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 examining millions of past games and then playing against itself (DeepMind's AlphaGo).

In this chapter, we will introduce artificial neural networks, starting with a quick tour of the very first ANN architectures. Then we will present *Multi-Layer Perceptrons* (MLPs) and implement one using TensorFlow to tackle the MNIST digit classification problem (introduced in Chapter 3).

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.

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 dark era. In the early 1980s there was a revival of interest in ANNs as new network architectures were invented and better training techniques were developed. But by the 1990s, powerful alternative Machine Learning techniques such as Support Vector Machines (see Chapter 5) were favored by most researchers, as they seemed to offer better results and stronger theoretical foundations. Finally, we are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? There are a few good reasons to believe that this one is different and will have a much more profound impact on our lives:

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's Law, 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

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

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 cortexes (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.

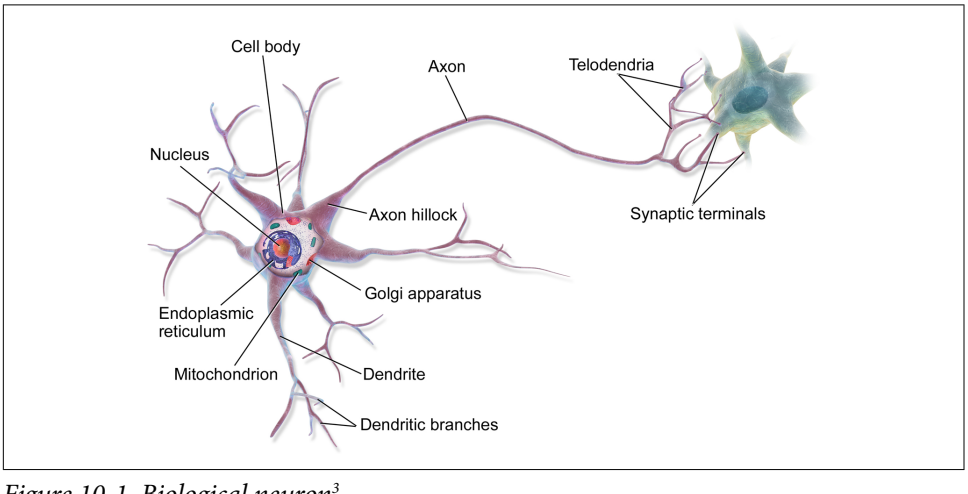

Figure 10-1. Biological neuron[3]

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 net-

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

works (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.

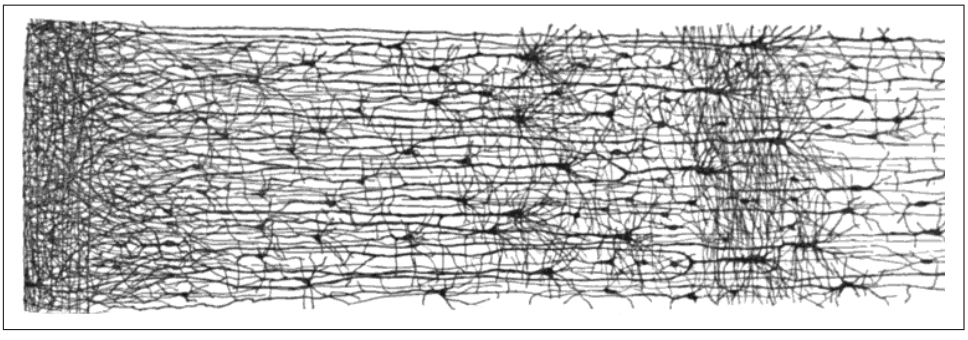

Figure 10-2. Multiple layers in a biological neural network (human cortex)[5]

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.

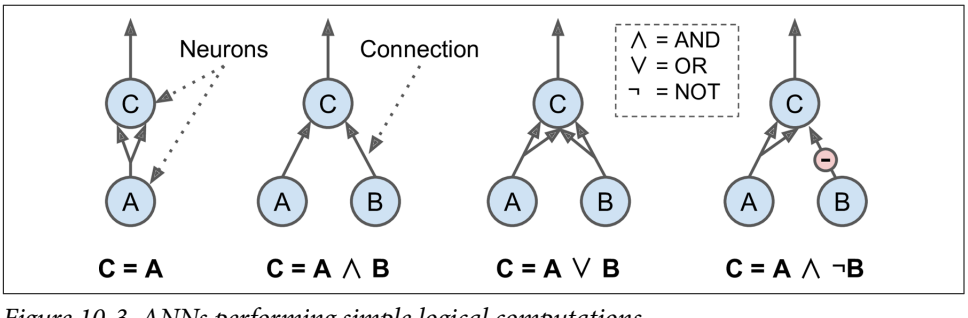

Figure 10-3. ANNs performing simple logical computations

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*.

- 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 *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 LTU computes a weighted sum of its inputs ($z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x}$), then applies a *step function* to that sum and outputs the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{w}^T \cdot \mathbf{x})$.

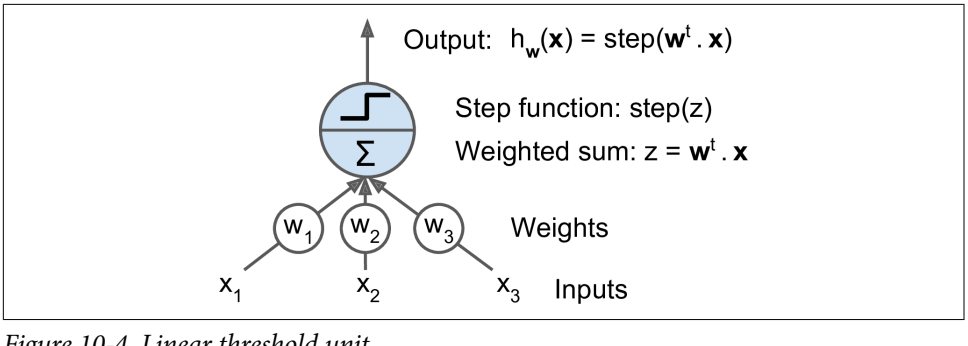

Figure 10-4. Linear threshold unit

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

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

A single LTU 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 LTU 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 an LTU 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 LTUs,[6] with each neuron connected to all the inputs. These connections are often represented using special passthrough neurons called *input neurons*: they just output whatever input they are fed. Moreover, an extra bias feature is generally added ($x_0 = 1$). This bias feature 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 multioutput classifier.

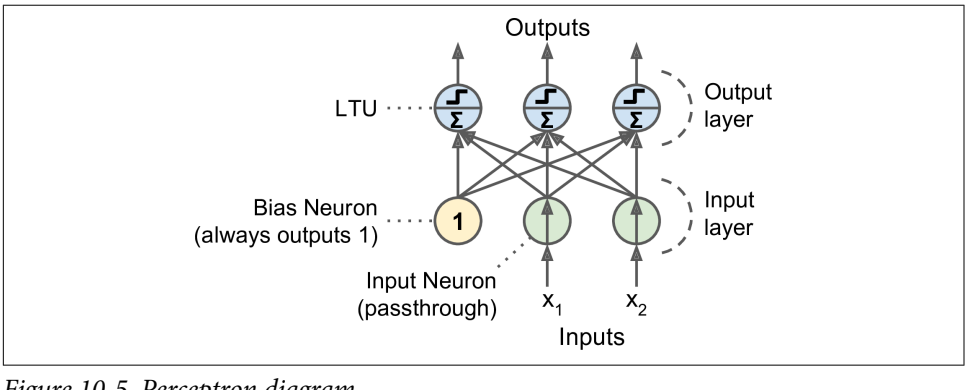

Figure 10-5. Perceptron diagram

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

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

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 does not reinforce connections that lead to the wrong output. 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 10-2.

Equation 10-2. Perceptron learning rule (weight update)

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

- $w_{i,j}$ is the connection weight between the ith input neuron and the jth output neuron.
- x_i is the ith input value of the current training instance.
- $\hat{y}_j$ is the output of the jth output neuron for the current training instance.
- y_j is the target output of the jth 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 LTU 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?
```

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.

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

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

You may have recognized 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: as a result, many researchers dropped *connectionism* altogether (i.e., the study of neural networks) 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, for each combination of inputs: with inputs (0, 0) or (1, 1) the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1.

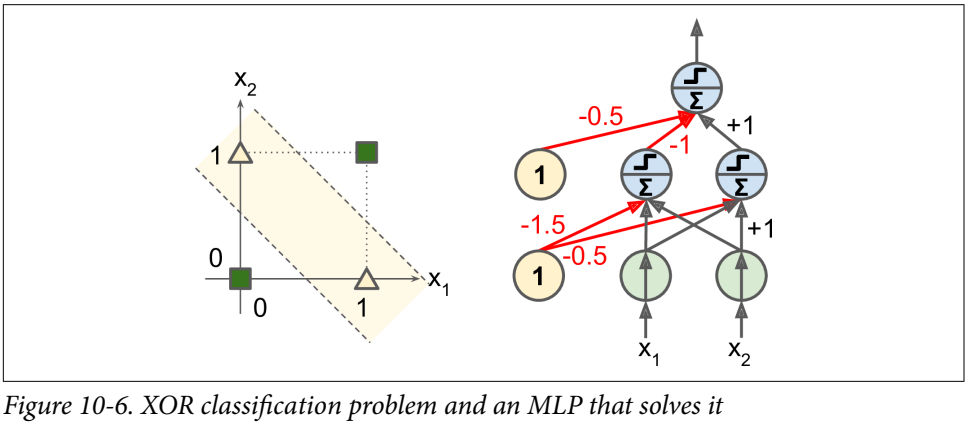

Figure 10-6. XOR classification problem and an MLP that solves it

Multi-Layer Perceptron and Backpropagation

An MLP is composed of one (passthrough) input layer, one or more layers of LTUs, called *hidden layers*, and one final layer of LTUs called the *output layer* (see Figure 10-7). Every layer except the output layer includes a bias neuron and is fully connected to the next layer. When an ANN has two or more hidden layers, it is called a *deep neural network* (DNN).

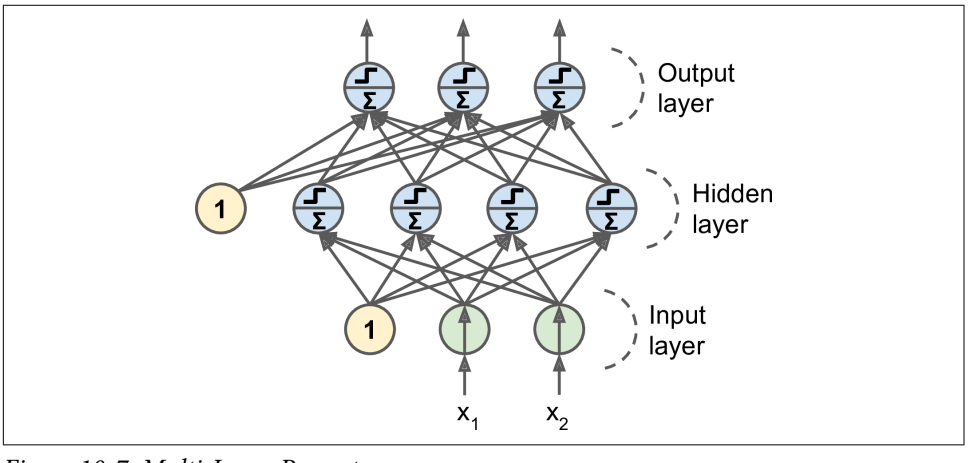

Figure 10-7. Multi-Layer Perceptron

For many years researchers struggled to find a way to train MLPs, without success. But in 1986, D. E. Rumelhart et al. published a groundbreaking article[8] introducing the *backpropagation* training algorithm.[9] Today we would describe it as Gradient Descent using reverse-mode autodiff (Gradient Descent was introduced in Chapter 4, and autodiff was discussed in Chapter 9).

For each training instance, the algorithm feeds it to the network and computes the output of every neuron in each consecutive layer (this is the forward pass, just like when making predictions). Then it measures the network's output error (i.e., the difference between the desired output and the actual output of the network), and it computes how much each neuron in the last hidden layer contributed to each output neuron's error. It then proceeds to measure how much of these error contributions came from each neuron in the previous hidden layer—and so on until the algorithm reaches the input layer. This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward in the network (hence the name of the algorithm). If you check out the

8 "Learning Internal Representations by Error Propagation," D. Rumelhart, G. Hinton, R. Williams (1986).

9 This algorithm was actually invented several times by various researchers in different fields, starting with P. Werbos in 1974.

reverse-mode autodiff algorithm in Appendix D, you will find that the forward and reverse passes of backpropagation simply perform reverse-mode autodiff. The last step of the backpropagation algorithm is a Gradient Descent step on all the connection weights in the network, using the error gradients measured earlier.

Let's make this even shorter: 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).

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. The backpropagation algorithm may be used with other *activation functions*, instead of 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 normalized (i.e., centered around 0) at the beginning of training. This often helps speed up convergence.

The ReLU function (introduced in Chapter 9)
: 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). However, in practice it works very well and has the advantage of being fast to compute. 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.

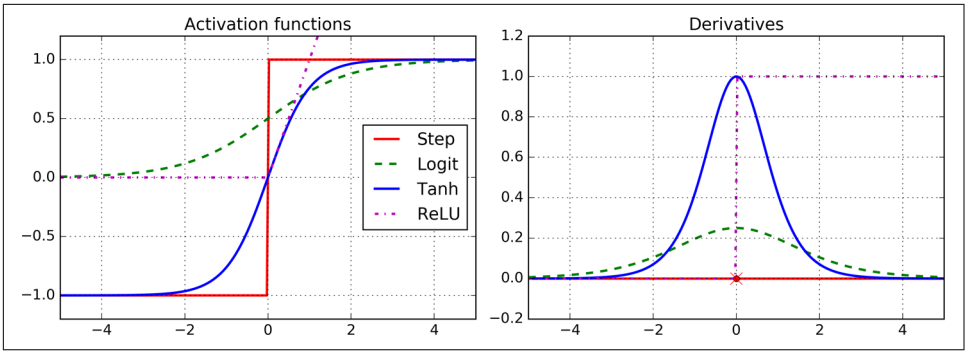

Figure 10-8. Activation functions and their derivatives

An MLP is often used for classification, with each output corresponding to a different binary class (e.g., spam/ham, urgent/not-urgent, and so on). When the classes are exclusive (e.g., classes 0 through 9 for digit image classification), the output layer is typically modified by replacing the individual activation functions by a shared *softmax* function (see Figure 10-9). The softmax function was introduced in Chapter 3. The output of each neuron corresponds to the estimated probability of the corresponding class. Note that 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).

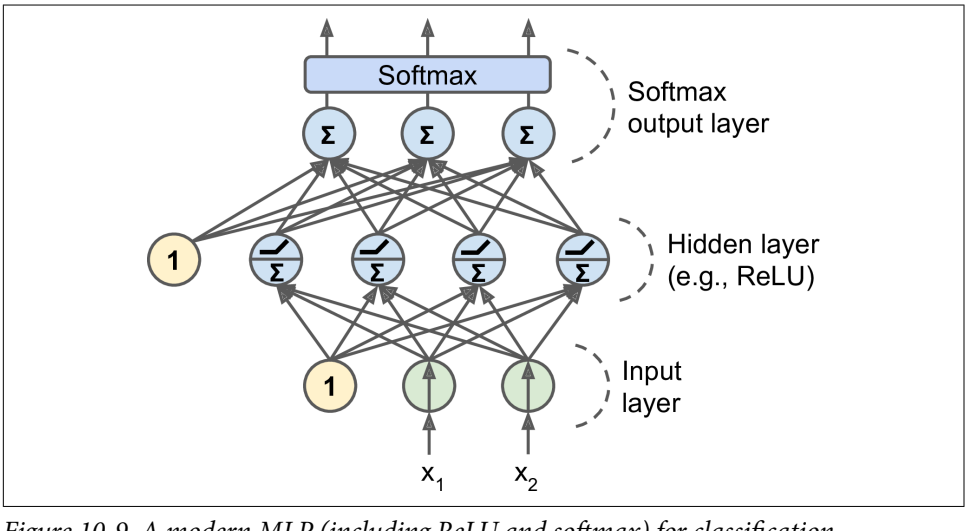

Figure 10-9. A modern MLP (including ReLU and softmax) for classification

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 the ReLU activation function generally works better in ANNs. This is one of the cases where the biological analogy was misleading.

Training an MLP with TensorFlow's High-Level API

The simplest way to train an MLP with TensorFlow is to use the high-level API TF.Learn, which is quite similar to Scikit-Learn's API. The `DNNClassifier` class makes it trivial to train a deep neural network with any number of hidden layers, and a softmax output layer to output estimated class probabilities. For example, the following code trains a DNN for classification with two hidden layers (one with 300 neurons, and the other with 100 neurons) and a softmax output layer with 10 neurons:

```
import tensorflow as tf

feature_columns = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300, 100], n_classes=10,
                                         feature_columns=feature_columns)
dnn_clf.fit(x=X_train, y=y_train, batch_size=50, steps=40000)
```

If you run this code on the MNIST dataset (after scaling it, e.g., by using Scikit-Learn's `StandardScaler`), you may actually get a model that achieves over 98.1% accuracy on the test set! That's better than the best model we trained in Chapter 3:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = list(dnn_clf.predict(X_test))
>>> accuracy_score(y_test, y_pred)
0.98180000000000001
```

The TF.Learn library also provides some convenience functions to evaluate models:

```
>>> dnn_clf.evaluate(X_test, y_test)
{'accuracy': 0.98180002, 'global_step': 40000, 'loss': 0.073678359}
```

Under the hood, the `DNNClassifier` class creates all the neuron layers, based on the ReLU activation function (we can change this by setting the `activation_fn` hyperparameter). The output layer relies on the softmax function, and the cost function is cross entropy (introduced in Chapter 4).

The TF.Learn API is still quite new, so some of the names and functions used in these examples may evolve a bit by the time you read this book. However, the general ideas should not change.

Training a DNN Using Plain TensorFlow

If you want more control over the architecture of the network, you may prefer to use TensorFlow's lower-level Python API (introduced in Chapter 9). In this section we will build the same model as before using this API, and we will implement Mini-batch Gradient Descent to train it on the MNIST dataset. The first step is the construction phase, building the TensorFlow graph. The second step is the execution phase, where you actually run the graph to train the model.

Construction Phase

Let's start. First we need to import the `tensorflow` library. Then we must specify the number of inputs and outputs, and set the number of hidden neurons in each layer:

```
import tensorflow as tf

n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

Next, just like you did in Chapter 9, you can use placeholder nodes to represent the training data and targets. The shape of `X` is only partially defined. We know that it will be a 2D tensor (i.e., a matrix), with instances along the first dimension and features along the second dimension, and we know that the number of features is going to be 28 x 28 (one feature per pixel), but we don't know yet how many instances each training batch will contain. So the shape of `X` is `(None, n_inputs)`. Similarly, we know that `y` will be a 1D tensor with one entry per instance, but again we don't know the size of the training batch at this point, so the shape is `(None)`.

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

Now let's create the actual neural network. The placeholder `X` will act as the input layer; during the execution phase, it will be replaced with one training batch at a time (note that all the instances in a training batch will be processed simultaneously by the neural network). Now you need to create the two hidden layers and the output layer. The two hidden layers are almost identical: they differ only by the inputs they are connected to and by the number of neurons they contain. The output layer is also very similar, but it uses a softmax activation function instead of a ReLU activation function. So let's create a `neuron_layer()` function that we will use to create one layer at a time. It will need parameters to specify the inputs, the number of neurons, the activation function, and the name of the layer:

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
```

```
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="weights")
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")
        z = tf.matmul(X, W) + b
        if activation=="relu":
            return tf.nn.relu(z)
        else:
            return z
```

Let's go through this code line by line:

1. First we create a name scope using the name of the layer: it will contain all the computation nodes for this neuron layer. This is optional, but the graph will look much nicer in TensorBoard if its nodes are well organized.
2. Next, we get the number of inputs by looking up the input matrix's shape and getting the size of the second dimension (the first dimension is for instances).
3. The next three lines create a `W` variable that will hold the weights matrix. It will be a 2D tensor containing all the connection weights between each input and each neuron; hence, its shape will be `(n_inputs, n_neurons)`. It will be initialized randomly, using a truncated[10] normal (Gaussian) distribution with a standard deviation of $2/\sqrt{n_{inputs}}$. Using this specific standard deviation helps the algorithm converge much faster (we will discuss this further in Chapter 11; it is one of those small tweaks to neural networks that have had a tremendous impact on their efficiency). It is important to initialize connection weights randomly for all hidden layers to avoid any symmetries that the Gradient Descent algorithm would be unable to break.[11]
4. The next line creates a `b` variable for biases, initialized to 0 (no symmetry issue in this case), with one bias parameter per neuron.
5. Then we create a subgraph to compute $\mathbf{z} = \mathbf{X} \cdot \mathbf{W} + \mathbf{b}$. This vectorized implementation will efficiently compute the weighted sums of the inputs plus the bias term for each and every neuron in the layer, for all the instances in the batch in just one shot.
6. Finally, if the `activation` parameter is set to `"relu"`, the code returns `relu(z)` (i.e., $\max(0, \mathbf{z})$), or else it just returns `z`.

10 Using a truncated normal distribution rather than a regular normal distribution ensures that there won't be any large weights, which could slow down training.

11 For example, if you set all the weights to 0, then all neurons will output 0, and the error gradient will be the same for all neurons in a given hidden layer. The Gradient Descent step will then update all the weights in exactly the same way in each layer, so they will all remain equal. In other words, despite having hundreds of neurons per layer, your model will act as if there were only one neuron per layer. It is not going to fly.

Okay, so now you have a nice function to create a neuron layer. Let's use it to create the deep neural network! The first hidden layer takes `X` as its input. The second takes the output of the first hidden layer as its input. And finally, the output layer takes the output of the second hidden layer as its input.

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "outputs")
```

Notice that once again we used a name scope for clarity. Also note that `logits` is the output of the neural network *before* going through the softmax activation function: for optimization reasons, we will handle the softmax computation later.

As you might expect, TensorFlow comes with many handy functions to create standard neural network layers, so there's often no need to define your own `neuron_layer()` function like we just did. For example, TensorFlow's `fully_connected()` function creates a fully connected layer, where all the inputs are connected to all the neurons in the layer. It takes care of creating the `weights` and `biases` variables, with the proper initialization strategy, and it uses the ReLU activation function by default (we can change this using the `activation_fn` argument). As we will see in Chapter 11, it also supports regularization and normalization parameters. Let's tweak the preceding code to use the `fully_connected()` function instead of our `neuron_layer()` function. Simply import the function and replace the `dnn` construction section with the following code:

```
from tensorflow.contrib.layers import fully_connected

with tf.name_scope("dnn"):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, scope="outputs",
                             activation_fn=None)
```

The `tensorflow.contrib` package contains many useful functions, but it is a place for experimental code that has not yet graduated to be part of the main TensorFlow API. So the `fully_connected()` function (and any other `contrib` code) may change or move in the future.

Now that we have the neural network model ready to go, we need to define the cost function that we will use to train it. Just as we did for Softmax Regression in Chapter 4, we will use cross entropy. As we discussed earlier, cross entropy will penalize models that estimate a low probability for the target class. TensorFlow provides several functions to compute cross entropy. We will use `sparse_softmax_cross_entropy_with_logits()`: it computes the cross entropy based on the

"logits" (i.e., the output of the network *before* going through the softmax activation function), and it expects labels in the form of integers ranging from 0 to the number of classes minus 1 (in our case, from 0 to 9). This will give us a 1D tensor containing the cross entropy for each instance. We can then use TensorFlow's `reduce_mean()` function to compute the mean cross entropy over all instances.

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
                   labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

The `sparse_softmax_cross_entropy_with_logits()` function is equivalent to applying the softmax activation function and then computing the cross entropy, but it is more efficient, and it properly takes care of corner cases like logits equal to 0. This is why we did not apply the softmax activation function earlier. There is also another function called `softmax_cross_entropy_with_logits()`, which takes labels in the form of one-hot vectors (instead of ints from 0 to the number of classes minus 1).

We have the neural network model, we have the cost function, and now we need to define a `GradientDescentOptimizer` that will tweak the model parameters to minimize the cost function. Nothing new; it's just like we did in Chapter 9:

```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

The last important step in the construction phase is to specify how to evaluate the model. We will simply use accuracy as our performance measure. First, for each instance, determine if the neural network's prediction is correct by checking whether or not the highest logit corresponds to the target class. For this you can use the `in_top_k()` function. This returns a 1D tensor full of boolean values, so we need to cast these booleans to floats and then compute the average. This will give us the network's overall accuracy.

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

And, as usual, we need to create a node to initialize all variables, and we will also create a `Saver` to save our trained model parameters to disk:

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Phew! This concludes the construction phase. This was fewer than 40 lines of code, but it was pretty intense: we created placeholders for the inputs and the targets, we created a function to build a neuron layer, we used it to create the DNN, we defined the cost function, we created an optimizer, and finally we defined the performance measure. Now on to the execution phase.

Execution Phase

This part is much shorter and simpler. First, let's load MNIST. We could use Scikit-Learn for that as we did in previous chapters, but TensorFlow offers its own helper that fetches the data, scales it (between 0 and 1), shuffles it, and provides a simple function to load one mini-batches a time. So let's use it instead:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
```

Now we define the number of epochs that we want to run, as well as the size of the mini-batches:

```
n_epochs = 400
batch_size = 50
```

And now we can train the model:

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,
                                            y: mnist.test.labels})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

This code opens a TensorFlow session, and it runs the `init` node that initializes all the variables. Then it runs the main training loop: at each epoch, the code iterates through a number of mini-batches that corresponds to the training set size. Each mini-batch is fetched via the `next_batch()` method, and then the code simply runs the training operation, feeding it the current mini-batch input data and targets. Next, at the end of each epoch, the code evaluates the model on the last mini-batch and on the full training set, and it prints out the result. Finally, the model parameters are saved to disk.

Using the Neural Network

Now that the neural network is trained, you can use it to make predictions. To do that, you can reuse the same construction phase, but change the execution phase like this:

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    X_new_scaled = [...]  # some new images (scaled from 0 to 1)
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

First the code loads the model parameters from disk. Then it loads some new images that you want to classify. Remember to apply the same feature scaling as for the training data (in this case, scale it from 0 to 1). Then the code evaluates the `logits` node. If you wanted to know all the estimated class probabilities, you would need to apply the `softmax()` function to the logits, but if you just want to predict a class, you can simply pick the class that has the highest logit value (using the `argmax()` function does the trick).

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 topology* (how neurons are interconnected), 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 initialization logic, and much more. How do you know what combination of hyperparameters is the best for your task?

Of course, you can use grid search with cross-validation to find the right hyperparameters, like you did in previous chapters, but since there are many hyperparameters to tune, and since training a neural network on a large dataset takes a lot of time, you will only be able to explore a tiny part of the hyperparameter space in a reasonable amount of time. It is much better to use randomized search, as we discussed in Chapter 2. Another option is to use a tool such as Oscar, which implements more complex algorithms to help you find a good set of hyperparameters quickly.

It helps to have an idea of what values are reasonable for each hyperparameter, so you can restrict the search space. Let's start with the number of hidden layers.

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, making them much faster to train.

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 DNNs 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).

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 13), 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 x 28 = 784 input neurons and 10 output neurons. As for the hidden layers, a common practice is to size them to form a funnel, 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 two hidden layers, the first with 300 neurons and the second with 100. However, this practice is not as common now, and you may simply use the same size for all hidden layers—for example, all hidden layers with 150 neurons: that's just one hyperparameter to tune instead of one per layer. 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 black 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, especially *dropout*, as we will see in Chapter 11). This has been dubbed the "stretch pants" approach:[12] 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.

Activation Functions

In most cases you can use the ReLU activation function in the hidden layers (or one of its variants, as we will see in Chapter 11). It is a bit faster to compute than other activation functions, and Gradient Descent does not get stuck as much on plateaus, thanks to the fact that it does not saturate for large input values (as opposed to the logistic function or the hyperbolic tangent function, which saturate at 1).

For the output layer, the softmax activation function is generally a good choice for classification tasks (when the classes are mutually exclusive). For regression tasks, you can simply use no activation function at all.

This concludes this introduction to artificial neural networks. In the following chapters, we will discuss techniques to train very deep nets, and distribute training across multiple servers and GPUs. Then we will explore a few other popular neural network architectures: convolutional neural networks, recurrent neural networks, and autoencoders.[13]

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

13 A few extra ANN architectures are presented in Appendix E.

Exercises

1. 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: $A \oplus B = (A \land \neg B) \lor (\neg A \land B)$.
2. Why is it generally preferable to use a Logistic Regression classifier rather than a classical Perceptron (i.e., a single layer of linear threshold units trained using the Perceptron training algorithm)? How can you tweak a Perceptron to make it equivalent to a Logistic Regression classifier?
3. Why was the logistic activation function a key ingredient in training the first MLPs?
4. Name three popular activation functions. Can you draw them?
5. 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 $\mathbf{b}_o$.
6. 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.
7. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
8. 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?
9. Train a deep MLP on the MNIST dataset and see if you can get over 98% precision. Just like in the last exercise of Chapter 9, try adding all the bells and whistles

(i.e., save checkpoints, restore the last checkpoint in case of an interruption, add summaries, plot learning curves using TensorBoard, and so on).

Solutions to these exercises are available in Appendix A.

CHAPTER 11

Training Deep Neural Nets

In Chapter 10 we introduced artificial neural networks and trained our first deep neural network. But it was a very shallow DNN, with only two 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 (say) 10 layers, 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, with such a large network, training would be extremely slow.
- Third, a model with millions of parameters would severely risk overfitting the training set.

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 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 Chapter 14). 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 Bengio[1] 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).

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.

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

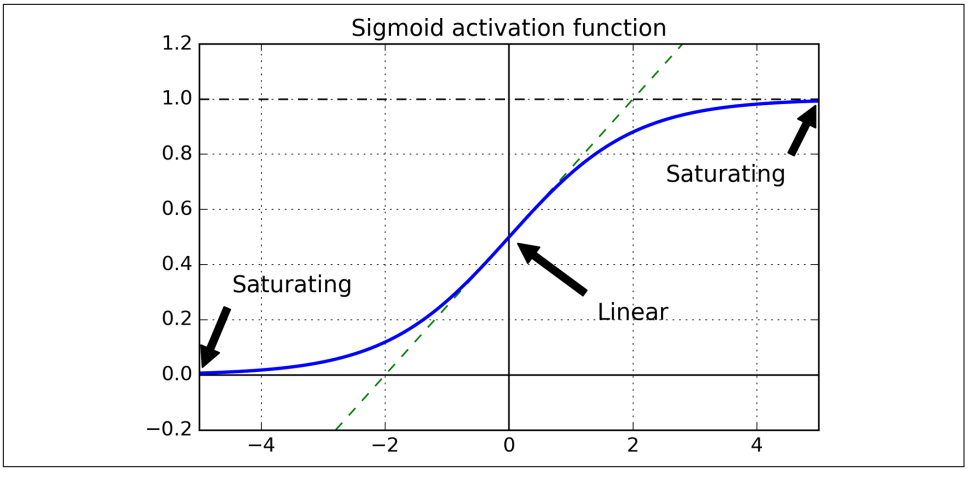

Figure 11-1. Logistic activation function saturation

Xavier 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 input and output connections, but they proposed a good compromise that has proven to work very well in practice: the connection weights must be initialized randomly as described in Equation 11-1, where n_{inputs} and n_{outputs} are the number of input and output connections for the layer whose weights are being initialized (also called *fan-in* and *fan-out*). This initialization strategy is often called *Xavier initialization* (after the author's first name), or sometimes *Glorot initialization*.

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.

Equation 11-1. Xavier initialization (when using the logistic activation function)

Normal distribution with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Or a uniform distribution between -r and +r, with $r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

When the number of input connections is roughly equal to the number of output connections, you get simpler equations (e.g., $\sigma = 1/\sqrt{n_{\text{inputs}}}$ or $r = \sqrt{3}/\sqrt{n_{\text{inputs}}}$). We used this simplified strategy in Chapter 10.[3]

Using the Xavier initialization strategy can speed up training considerably, and it is one of the tricks that led to the current success of Deep Learning. Some recent papers[4] have provided similar strategies for different activation functions, as shown in Table 11-1. 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).

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

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

By default, the `fully_connected()` function (introduced in Chapter 10) uses Xavier initialization (with a uniform distribution). You can change this to He initialization by using the `variance_scaling_initializer()` function like this:

```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = fully_connected(X, n_hidden1, weights_initializer=he_init, scope="h1")
```

3 This simplified strategy was actually already proposed much earlier—for example, in the 1998 book *Neural Networks: Tricks of the Trade* by Genevieve Orr and Klaus-Robert Müller (Springer).

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

He initialization considers only the fan-in, not the average between fan-in and fan-out like in Xavier initialization. This is also the default for the `variance_scaling_initializer()` function, but you can change this by setting the argument `mode="FAN_AVG"`.

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. During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it will start outputting 0. When this happen, the neuron is unlikely to come back to life since the gradient of the ReLU function is 0 when its input is negative.

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 recent paper[5] 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). 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

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

was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

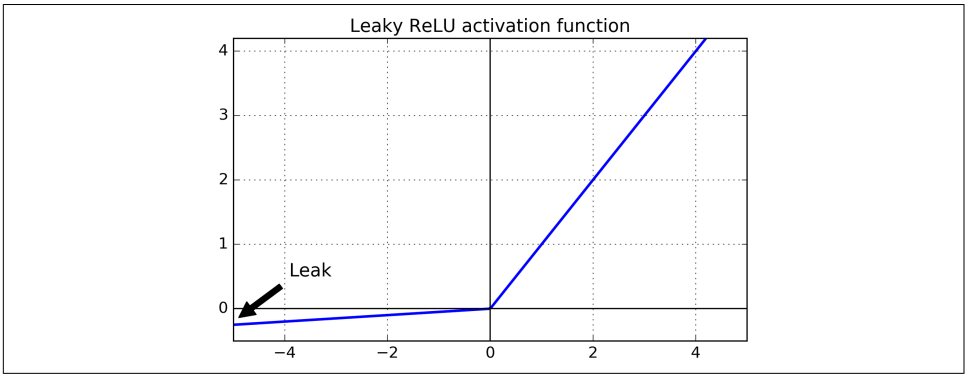

Figure 11-2. Leaky ReLU

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

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

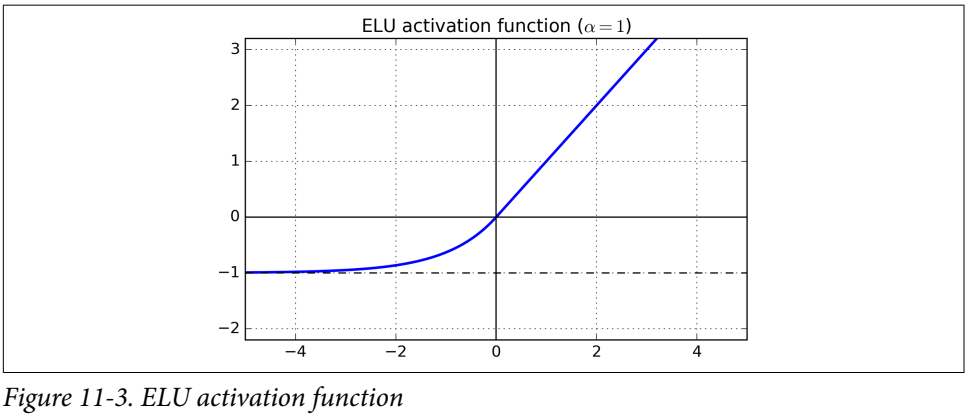

Figure 11-3. ELU activation function

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

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 dying units issue.
- Third, 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.

So which activation function should you use for the hidden layers of your deep neural networks? Although your mileage will vary, in general ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic. If you care a lot about runtime performance, then you may prefer leaky ReLUs over ELUs. If you don't want to tweak yet another hyperparameter, you may just use the default α values suggested earlier (0.01 for the leaky ReLU, and 1 for ELU). 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.

TensorFlow offers an `elu()` function that you can use to build your neural network. Simply set the `activation_fn` argument when calling the `fully_connected()` function, like this:

```
hidden1 = fully_connected(X, n_hidden1, activation_fn=tf.nn.elu)
```

TensorFlow does not have a predefined function for leaky ReLUs, but it is easy enough to define:

```
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = fully_connected(X, n_hidden1, activation_fn=leaky_relu)
```

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,[7] Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) to address the vanishing/exploding gradients problems, and more generally the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change (which they call the *Internal Covariate Shift* problem).

The technique consists of adding an operation in the model just before the activation function of each layer, simply zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting). In other words, this operation lets the model learn the optimal scale and mean of the inputs for each layer.

In order to zero-center and normalize the inputs, the algorithm needs to estimate the inputs' mean and standard deviation. It does so by evaluating the mean and standard deviation of the inputs 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. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$
3. $\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \widehat{\mathbf{x}}^{(i)} + \beta$

- μ_B is the empirical mean, evaluated over the whole mini-batch B.
- σ_B is the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B is the number of instances in the mini-batch.

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

- $\widehat{\mathbf{x}}^{(i)}$ is the zero-centered and normalized input.
- γ is the scaling parameter for the layer.
- β is the shifting parameter (offset) for the layer.
- ϵ is a tiny number to avoid division by zero (typically 10^{-3}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a scaled and shifted version of the inputs.

At test time, there is no mini-batch to compute the empirical mean and standard deviation, so instead you simply use the whole training set's mean and standard deviation. These are typically efficiently computed during training using a moving average. So, in total, four parameters are learned for each batch-normalized layer: γ (scale), β (offset), μ (mean), and σ (standard deviation).

The authors demonstrated that this technique considerably improved all the deep neural networks they experimented with. The vanishing 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 the chapter).

Batch Normalization does, however, add some complexity to the model (although it removes the need for normalizing the input data since the first hidden layer will take care of that, provided it is batch-normalized). 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 at first while Gradient Descent is searching for the optimal scales and offsets for each layer, but it accelerates once it has found reasonably good values.

Implementing Batch Normalization with TensorFlow

TensorFlow provides a `batch_normalization()` function that simply centers and normalizes the inputs, but you must compute the mean and standard deviation yourself (based on the mini-batch data during training or on the full dataset during testing, as just discussed) and pass them as parameters to this function, and you must also handle the creation of the scaling and offset parameters (and pass them to this function). It is doable, but not the most convenient approach. Instead, you should use the `batch_norm()` function, which handles all this for you. You can either call it directly or tell the `fully_connected()` function to use it, such as in the following code:

```
import tensorflow as tf
from tensorflow.contrib.layers import batch_norm

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

is_training = tf.placeholder(tf.bool, shape=(), name='is_training')
bn_params = {
    'is_training': is_training,
    'decay': 0.99,
    'updates_collections': None
}

hidden1 = fully_connected(X, n_hidden1, scope="hidden1",
                          normalizer_fn=batch_norm, normalizer_params=bn_params)
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2",
                          normalizer_fn=batch_norm, normalizer_params=bn_params)
logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="outputs",
                         normalizer_fn=batch_norm, normalizer_params=bn_params)
```

Let's walk through this code. The first lines are fairly self-explanatory, until we define the `is_training` placeholder, which will either be `True` or `False`. This will be used to tell the `batch_norm()` function whether it should use the current mini-batch's mean and standard deviation (during training) or the running averages that it keeps track of (during testing).

Next we define `bn_params`, which is a dictionary that defines the parameters that will be passed to the `batch_norm()` function, including `is_training` of course. The algorithm uses *exponential decay* to compute the running averages, which is why it requires the `decay` parameters. Given a new value v, the running average $\hat{v}$ is updated through the equation $\hat{v} \leftarrow \hat{v} \times \text{decay} + v \times (1 - \text{decay})$. A good decay 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). Finally, `updates_collections` should be set to `None` if you want the `batch_norm()` function to update the running averages right before it performs batch normalization during training (i.e., when `is_training=True`). If you don't set this parameter, by default TensorFlow will just add the operations that update the running averages to a collection of operations that you must run yourself.

Lastly, we create the layers by calling the `fully_connected()` function, just like we did in Chapter 10, but this time we tell it to use the `batch_norm()` function (with the parameters `nb_params`) to normalize the inputs right before calling the activation function.

Note that by default `batch_norm()` only centers, normalizes, and shifts the inputs; it does not scale them (i.e., γ is fixed to 1). This makes sense for layers with no activation function or with the ReLU activation function, since the next layer's weights can take care of scaling, but for any other activation function, you should add `"scale": True` to `bn_params`.

You may have noticed that defining the preceding three layers was fairly repetitive since several parameters were identical. To avoid repeating the same parameters over and over again, you can create an *argument scope* using the `arg_scope()` function: the first parameter is a list of functions, and the other parameters will be passed to these functions automatically. The last three lines of the preceding code can be modified like so:

```
[...]

with tf.contrib.framework.arg_scope(
        [fully_connected],
        normalizer_fn=batch_norm,
        normalizer_params=bn_params):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, scope="outputs",
                             activation_fn=None)
```

It may not look much better than before in this small example, but if you have 10 layers and want to set the activation function, the initializers, the normalizers, the regularizers, and so on, it will make your code much more readable.

The rest of the construction phase is the same as in Chapter 10: define the cost function, create an optimizer, tell it to minimize the cost function, define the evaluation operations, create a `Saver`, and so on.

The execution phase is also pretty much the same, with one exception. Whenever you run an operation that depends on the `batch_norm` layer, you need to set the `is_training` placeholder to `True` or `False`:

```
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op,
                     feed_dict={is_training: True, X: X_batch, y: y_batch})
        accuracy_score = accuracy.eval(
            feed_dict={is_training: False, X: X_test_scaled, y: y_test}))
        print(accuracy_score)
```

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

Gradient Clipping

A 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 mostly useful for recurrent neural networks; see Chapter 14). This is called *Gradient Clipping*.[8] In general people now prefer Batch Normalization, but it's still useful to know about Gradient Clipping and how to implement it.

In TensorFlow, the optimizer's `minimize()` function takes care of both computing the gradients and applying them, so you must instead call the optimizer's `compute_gradients()` method first, then create an operation to clip the gradients using the `clip_by_value()` function, and finally create an operation to apply the clipped gradients using the optimizer's `apply_gradients()` method:

```
threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

You would then run this `training_op` at every training step, as usual. It will compute the gradients, clip them between –1.0 and 1.0, and apply them. The threshold is a hyperparameter you can tune.

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

8 "On the difficulty of training recurrent neural networks," R. Pascanu et al. (2013).

to the one you are trying to tackle, 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, so you should try to reuse parts of the first network (see Figure 11-4).

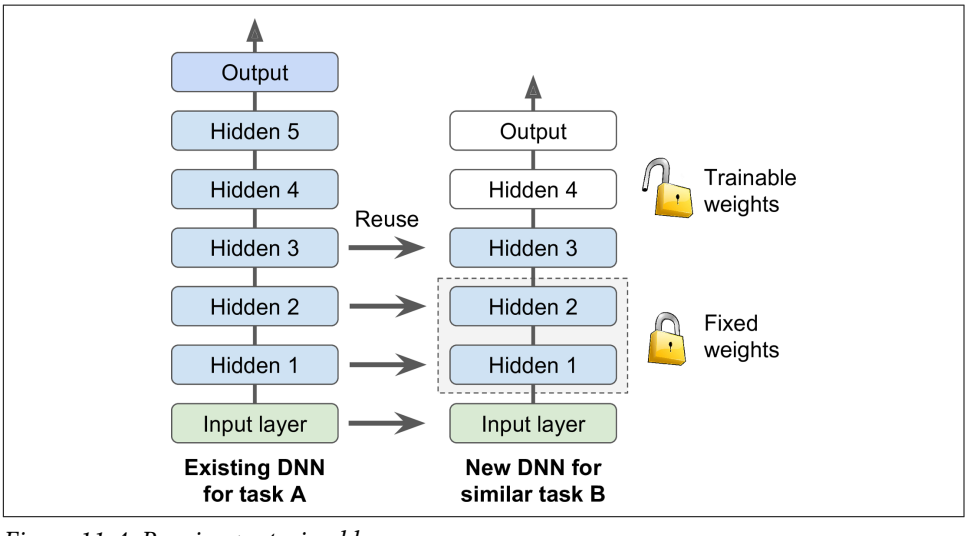

Figure 11-4. Reusing pretrained layers

If the input pictures of your new task don't have the same size as the ones used in the original task, you will have to add a preprocessing step to resize them to the size expected by the original model. More generally, transfer learning will work only well if the inputs have similar low-level features.

Reusing a TensorFlow Model

If the original model was trained using TensorFlow, you can simply restore it and train it on the new task:

```
[...] # construct the original model

with tf.Session() as sess:
    saver.restore(sess, "./my_original_model.ckpt")
    [...] # Train it on your new task
```

However, in general you will want to reuse only part of the original model (as we will discuss in a moment). A simple solution is to configure the `Saver` to restore only a subset of the variables from the original model. For example, the following code restores only hidden layers 1, 2, and 3:

```
[...] # build new model with the same definition as before for hidden layers 1-3

init = tf.global_variables_initializer()

reuse_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[123]")
reuse_vars_dict = dict([(var.name, var.name) for var in reuse_vars])
original_saver = tf.Saver(reuse_vars_dict) # saver to restore the original model

new_saver = tf.Saver() # saver to save the new model

with tf.Session() as sess:
    sess.run(init)
    original_saver.restore("./my_original_model.ckpt") # restore layers 1 to 3
    [...] # train the new model
    new_saver.save("./my_new_model.ckpt") # save the whole model
```

First we build the new model, making sure to copy the original model's hidden layers 1 to 3. We also create a node to initialize all variables. Then we get the list of all variables that were just created with `"trainable=True"` (which is the default), and we keep only the ones whose scope matches the regular expression `"hidden[123]"` (i.e., we get all trainable variables in hidden layers 1 to 3). Next we create a dictionary mapping the name of each variable in the original model to its name in the new model (generally you want to keep the exact same names). Then we create a `Saver` that will restore only these variables, and we create another `Saver` to save the entire new model, not just layers 1 to 3. We then start a session and initialize all variables in the model, then restore the variable values from the original model's layers 1 to 3. Finally, we train the model on the new task and save it.

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.

Reusing Models from Other Frameworks

If the model was trained using another framework, you will need to load the weights manually (e.g., using Theano code if it was trained with Theano), then assign them to the appropriate variables. This can be quite tedious. For example, the following code shows how you would copy the weight and biases from the first hidden layer of a model trained using another framework:

```
original_w = [...] # Load the weights from the other framework
original_b = [...] # Load the biases from the other framework

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
[...] # # Build the rest of the model

# Get a handle on the variables created by fully_connected()
with tf.variable_scope("", default_name="", reuse=True):  # root scope
    hidden1_weights = tf.get_variable("hidden1/weights")
    hidden1_biases = tf.get_variable("hidden1/biases")

# Create nodes to assign arbitrary values to the weights and biases
original_weights = tf.placeholder(tf.float32, shape=(n_inputs, n_hidden1))
original_biases = tf.placeholder(tf.float32, shape=(n_hidden1))
assign_hidden1_weights = tf.assign(hidden1_weights, original_weights)
assign_hidden1_biases = tf.assign(hidden1_biases, original_biases)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    sess.run(assign_hidden1_weights, feed_dict={original_weights: original_w})
    sess.run(assign_hidden1_biases, feed_dict={original_biases: original_b})
    [...] # Train the model on your new task
```

Freezing the Lower Layers

It is likely that the lower layers of the first DNN have learned to detect low-level features in pictures that will be useful across both image classification tasks, so you can just reuse these layers as they are. It is generally a good idea to "freeze" their weights when training the new DNN: if the lower-layer weights are fixed, then the higher-layer weights will be easier to train (because they won't have to learn a moving target). To freeze the lower layers during training, the simplest solution is to give the optimizer the list of variables to train, excluding the variables from the lower layers:

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[34]|outputs")
training_op = optimizer.minimize(loss, var_list=train_vars)
```

The first line gets the list of all trainable variables in hidden layers 3 and 4 and in the output layer. This leaves out the variables in the hidden layers 1 and 2. Next we provide this restricted list of trainable variables to the optimizer's `minimize()` function. Ta-da! Layers 1 and 2 are now frozen: they will not budge during training (these are often called *frozen layers*).

Caching the Frozen Layers

Since the frozen layers won't change, it is possible to cache the output of the topmost frozen layer for each training instance. Since training goes through the whole dataset many times, this will give you a huge speed boost as you will only need to go through the frozen layers once per training instance (instead of once per epoch). For example, you could first run the whole training set through the lower layers (assuming you have enough RAM):

```
hidden2_outputs = sess.run(hidden2, feed_dict={X: X_train})
```

Then during training, instead of building batches of training instances, you would build batches of outputs from hidden layer 2 and feed them to the training operation:

```
import numpy as np

n_epochs = 100
n_batches = 500

for epoch in range(n_epochs):
    shuffled_idx = rnd.permutation(len(hidden2_outputs))
    hidden2_batches = np.array_split(hidden2_outputs[shuffled_idx], n_batches)
    y_batches = np.array_split(y_train[shuffled_idx], n_batches)
    for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):
        sess.run(training_op, feed_dict={hidden2: hidden2_batch, y: y_batch})
```

The last line runs the training operation defined earlier (which freezes layers 1 and 2), and feeds it a batch of outputs from the second hidden layer (as well as the targets for that batch). Since we give TensorFlow the output of hidden layer 2, it does not try to evaluate it (or any node it depends on).

Tweaking, Dropping, or Replacing the Upper Layers

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.

Try freezing all the copied layers first, 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.

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.

Model Zoos

Where can you find a neural network trained for a task similar to the one you want to tackle? The first place to look is obviously in your own catalog of models. This is one good reason to save all your models and organize them so you can retrieve them later easily. Another option is to search in a *model zoo*. Many people train Machine Learning models for various tasks and kindly release their pretrained models to the public.

TensorFlow has its own model zoo available at *https://github.com/tensorflow/models*. In particular, it contains most of the state-of-the-art image classification nets such as VGG, Inception, and ResNet (see Chapter 13, and check out the *models/slim* directory), including the code, the pretrained models, and tools to download popular image datasets.

Another popular model zoo is Caffe's Model Zoo. It also contains many computer vision models (e.g., LeNet, AlexNet, ZFNet, GoogLeNet, VGGNet, inception) trained on various datasets (e.g., ImageNet, Places Database, CIFAR10, etc.). Saumitro Dasgupta wrote a converter, which is available at *https://github.com/ethereon/caffe-tensorflow*.

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). That is, if you have 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 Appendix E) or autoencoders (see Chapter 15). 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 fine-tune the network using supervised learning (i.e., with backpropagation).

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 common to train DNNs purely using backpropagation. However, unsupervised pretraining (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.[9]

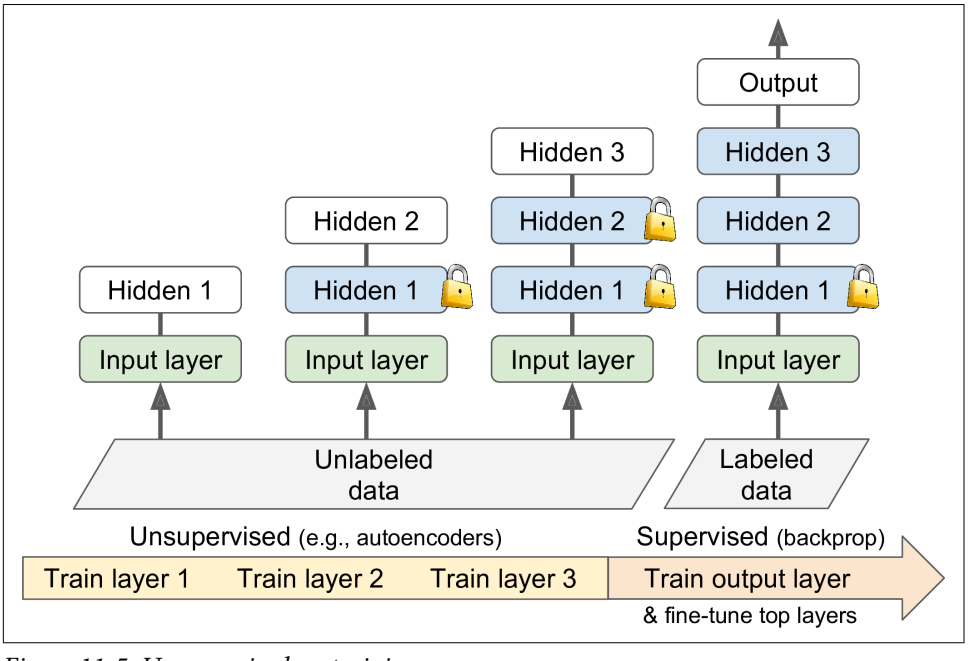

Figure 11-5. Unsupervised pretraining

Pretraining on an Auxiliary Task

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 internet and train a first neural network to detect whether or not two different pictures feature the same person. Such a

9 Another option is to come up with a supervised task for which you can easily gather a lot of labeled training data, then use transfer learning, as explained earlier. For example, if you want to train a model to identify your friends in pictures, you could download millions of faces on the internet and train a classifier to detect whether two faces are identical or not, then use this classifier to compare a new picture with each picture of your friends.

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.

It is often rather cheap to gather unlabeled training examples, but quite expensive to label them. In this situation, a common technique is to label all your training examples as "good," then generate many new training instances by corrupting the good ones, and label these corrupted instances as "bad." Then you can train a first neural network to classify instances as good or bad. For example, you could download millions of sentences, label them as "good," then randomly change a word in each sentence and label the resulting sentences as "bad." If a neural network can tell that "The dog sleeps" is a good sentence but "The dog they" is bad, it probably knows quite a lot about language. Reusing its lower layers will likely help in many language processing tasks.

Another approach is to train a first network to output a score for each training instance, and use a cost function that ensures that a good instance's score is greater than a bad instance's score by at least some margin. This is called *max margin 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. Another huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. In this section we will present the most popular ones: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam optimization.

Spoiler alert: the conclusion of this section is that you should almost always use Adam optimization,[10] so if you don't care about how it works, simply replace your `GradientDescentOptimizer` with an `AdamOptimizer` and skip to the next section! With just this small change, training will typically be several times faster. However, Adam optimization does have three hyperparameters that you can tune (plus the learning rate); the default values usually work fine, but if you ever need to tweak them it may be helpful to know what they do. Adam optimization combines several ideas from other optimization algorithms, so it is useful to look at these algorithms first.

10 At least for now: research is moving fast, especially in the field of optimization. Be sure to take a look at the latest and greatest optimizers every time a new version of TensorFlow is released.

Momentum optimization

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the very simple idea behind *Momentum optimization*, proposed by Boris Polyak in 1964.[11] In contrast, regular Gradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom.

Recall that Gradient Descent simply updates the weights θ by directly subtracting the gradient of the cost function $J(\theta)$ with regards to the weights ($\nabla_\theta J(\theta)$) multiplied by the learning rate η. The equation is: $\theta \leftarrow \theta - \eta\nabla_\theta J(\theta)$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it adds the local gradient to the *momentum vector* $\mathbf{m}$ (multiplied by the learning rate η), and it updates the weights by simply subtracting this momentum vector (see Equation 11-4). In other words, the gradient is used as an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, simply called the *momentum*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

Equation 11-4. Momentum algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \mathbf{m}$

You can easily verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate η multiplied by $\frac{1}{1-\beta}$. For example, if $\beta = 0.9$, then the terminal velocity is equal to 10 times the gradient times the learning rate, so Momentum optimization ends up going 10 times faster than Gradient Descent! This allows Momentum optimization to escape from plateaus much faster than Gradient Descent. In particular, we saw in Chapter 4 that when the inputs have very different scales the cost function will look like an elongated bowl (see Figure 4-7). Gradient Descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, Momentum optimization will roll down the bottom of the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use Batch Normalization, the upper layers will often end up having inputs with very

11 "Some methods of speeding up the convergence of iteration methods," B. Polyak (1964).

different scales, so using Momentum optimization helps a lot. It can also help roll past local optima.

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it is good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

Implementing Momentum optimization in TensorFlow is a no-brainer: just replace the `GradientDescentOptimizer` with the `MomentumOptimizer`, then lie back and profit!

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

The one drawback of Momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than Gradient Descent.

Nesterov Accelerated Gradient

One small variant to Momentum optimization, proposed by Yurii Nesterov in 1983,[12] is almost always faster than vanilla Momentum optimization. The idea of *Nesterov Momentum optimization*, or *Nesterov Accelerated Gradient* (NAG), is to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum (see Equation 11-5). The only difference from vanilla Momentum optimization is that the gradient is measured at $\theta + \beta\mathbf{m}$ rather than at θ.

Equation 11-5. Nesterov Accelerated Gradient algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta - \mathbf{m}$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than using the gradient at the original position, as you can see in Figure 11-6 (where ∇_1 represents the gradient of the cost function measured at the starting point θ, and ∇_2 represents the gradient at the point located at $\theta + \beta\mathbf{m}$). As you can see, the Nesterov update ends up

12 "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$," Yurii Nesterov (1983).

slightly closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular Momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push further across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus converges faster.

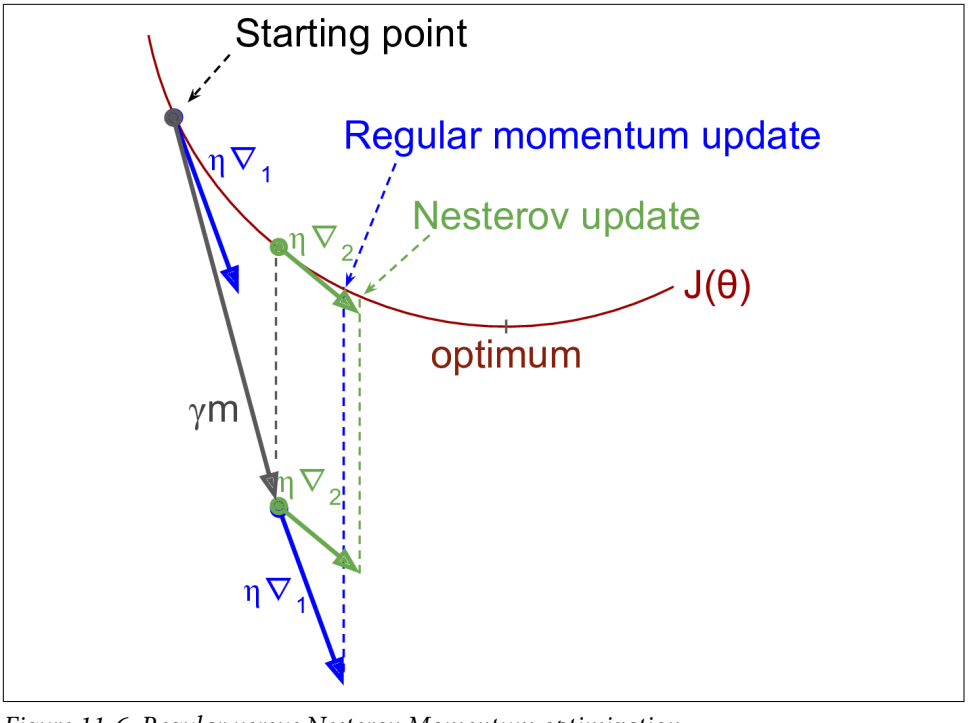

Figure 11-6. Regular versus Nesterov Momentum optimization

NAG will almost always speed up training compared to regular Momentum optimization. To use it, simply set `use_nesterov=True` when creating the `MomentumOptimizer`:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=True)
```

AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley. It would be nice if the algorithm could detect this early on and correct its direction to point a bit more toward the global optimum.

The *AdaGrad* algorithm[13] achieves this by scaling down the gradient vector along the steepest dimensions (see Equation 11-6):

Equation 11-6. AdaGrad algorithm

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

The first step accumulates the square of the gradients into the vector **s** (the $\otimes$ symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial / \partial\, \theta_i\, J(\theta))^2$ for each element s_i of the vector **s**; in other words, each s_i accumulates the squares of the partial derivative of the cost function with regards to parameter θ_i. If the cost function is steep along the ith dimension, then s_i will get larger and larger at each iteration.

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{\mathbf{s} + \epsilon}$ (the $\oslash$ symbol represents the element-wise division, and ϵ is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to computing $\theta_i \leftarrow \theta_i - \eta\, \partial / \partial \theta_i\, J(\theta) / \sqrt{s_i + \epsilon}$ for all parameters θ_i (simultaneously).

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see Figure 11-7). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η.

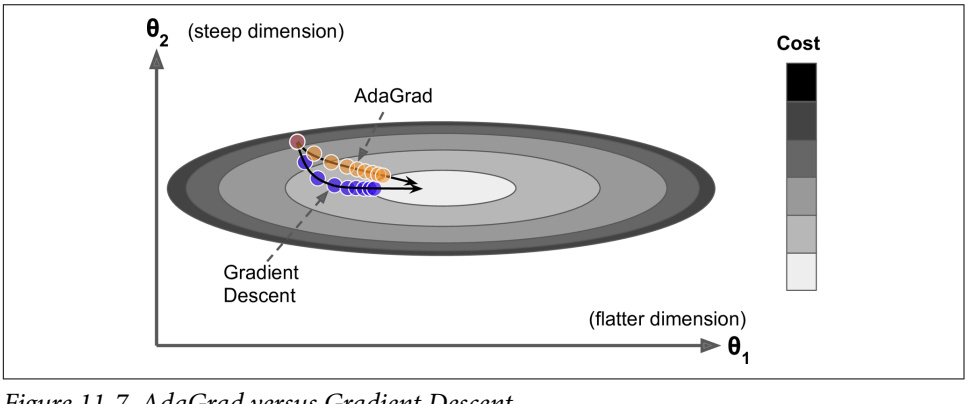

Figure 11-7. AdaGrad versus Gradient Descent

13 "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," J. Duchi et al. (2011).

AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though TensorFlow has an `AdagradOptimizer`, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though).

RMSProp

Although AdaGrad slows down a bit too fast and ends up never converging to the global optimum, the *RMSProp* algorithm[14] fixes this by accumulating only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training). It does so by using exponential decay in the first step (see Equation 11-7).

Equation 11-7. RMSProp algorithm

1. $\mathbf{s} \leftarrow \beta\mathbf{s} + (1 - \beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

The decay rate β is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, TensorFlow has an `RMSPropOptimizer` class:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. It also generally performs better than Momentum optimization and Nesterov Accelerated Gradients. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam Optimization

Adam,[15] which stands for *adaptive moment estimation*, combines the ideas of Momentum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it keeps

14 This algorithm was created by Tijmen Tieleman and Geoffrey Hinton in 2012, and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: *http://goo.gl/RsQeis*; video: *https://goo.gl/XUbIyJ*). Amusingly, since the authors have not written a paper to describe it, researchers often cite "slide 29 in lecture 6" in their papers.

15 "Adam: A Method for Stochastic Optimization," D. Kingma, J. Ba (2015).

track of an exponentially decaying average of past squared gradients (see Equation 11-8).[16]

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} + \left(1 - \beta_1\right) \nabla_\theta J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + \left(1 - \beta_2\right) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
3. $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - {\beta_1}^T}$
4. $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - {\beta_2}^T}$
5. $\theta \leftarrow \theta - \eta\, \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

- T represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both Momentum optimization and RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost **m** and **s** at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term ϵ is usually initialized to a tiny number such as 10^{-8}. These are the default values for TensorFlow's `AdamOptimizer` class, so you can simply use:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

In fact, since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter η. You can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

16 These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment*, while the variance is often called the *second moment*, hence the name of the algorithm.

All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*). The optimization literature contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are n^2 Hessians per output (where n is the number of parameters), as opposed to just n Jacobians per output. Since DNNs typically have tens of thousands of parameters, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

Training Sparse Models

All the optimization algorithms just presented produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One trivial way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to 0).

Another option is to apply strong ℓ_1 regularization during training, as it pushes the optimizer to zero out as many weights as it can (as discussed in Chapter 4 about Lasso Regression).

However, in some cases these techniques may remain insufficient. One last option is to apply *Dual Averaging*, often called *Follow The Regularized Leader* (FTRL), a technique proposed by Yurii Nesterov.[17] When used with ℓ_1 regularization, this technique often leads to very sparse models. TensorFlow implements a variant of FTRL called *FTRL-Proximal*[18] in the `FTRLOptimizer` class.

Learning Rate Scheduling

Finding a good learning rate can be tricky. If you set it way too high, training may actually diverge (as we discussed in Chapter 4). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never settling down (unless you use an adaptive learning rate optimization algorithm such as AdaGrad, RMSProp, or Adam, but even then it may take time to settle). If you have a limited computing budget, you may have to inter-

17 "Primal-Dual Subgradient Methods for Convex Problems," Yurii Nesterov (2005).

18 "Ad Click Prediction: a View from the Trenches," H. McMahan et al. (2013).

rupt training before it has converged properly, yielding a suboptimal solution (see Figure 11-8).

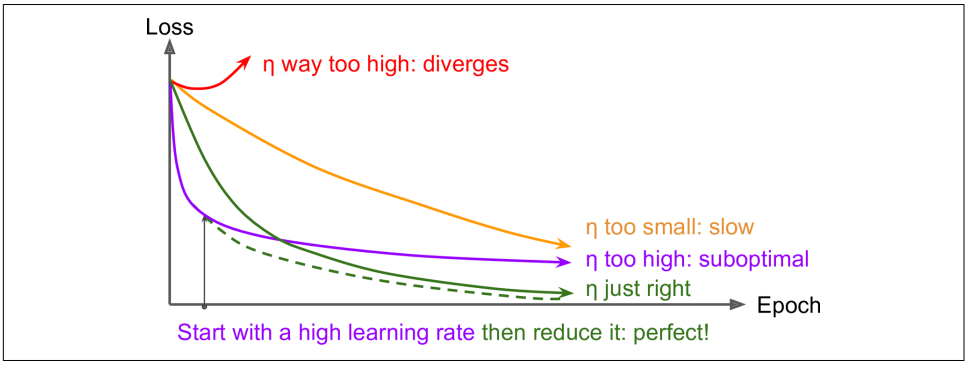

Figure 11-8. Learning curves for various learning rates η

You may be able to find a fairly good learning rate by training your network several times during just a few epochs using various learning rates and comparing the learning curves. The ideal learning rate will learn quickly and converge to good solution.

However, you can do better than a constant learning rate: if you start with a high learning rate and then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. These strategies are called *learning schedules* (we briefly introduced this concept in Chapter 4), the most common of which are:

Predetermined piecewise constant learning rate
: For example, set the learning rate to $\eta_0 = 0.1$ at first, then to $\eta_1 = 0.001$ after 50 epochs. Although this solution can work very well, it often requires fiddling around to figure out the right learning rates and when to use them.

Performance scheduling
: Measure the validation error every N steps (just like for early stopping) and reduce the learning rate by a factor of λ when the error stops dropping.

Exponential scheduling
: Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0\, 10^{-t/r}$. This works great, but it requires tuning η_0 and r. The learning rate will drop by a factor of 10 every r steps.

Power scheduling
: Set the learning rate to $\eta(t) = \eta_0\, (1 + t/r)^{-c}$. The hyperparameter c is typically set to 1. This is similar to exponential scheduling, but the learning rate drops much more slowly.

A 2013 paper[19] by Andrew Senior et al. compared the performance of some of the most popular learning schedules when training deep neural networks for speech recognition using Momentum optimization. The authors concluded that, in this setting, both performance scheduling and exponential scheduling performed well, but they favored exponential scheduling because it is simpler to implement, is easy to tune, and converged slightly faster to the optimal solution.

Implementing a learning schedule with TensorFlow is fairly straightforward:

```
initial_learning_rate = 0.1
decay_steps = 10000
decay_rate = 1/10
global_step = tf.Variable(0, trainable=False)
learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                           decay_steps, decay_rate)
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
training_op = optimizer.minimize(loss, global_step=global_step)
```

After setting the hyperparameter values, we create a nontrainable variable `global_step` (initialized to 0) to keep track of the current training iteration number. Then we define an exponentially decaying learning rate (with $\eta_0 = 0.1$ and $r = 10{,}000$) using TensorFlow's `exponential_decay()` function. Next, we create an optimizer (in this example, a `MomentumOptimizer`) using this decaying learning rate. Finally, we create the training operation by calling the optimizer's `minimize()` method; since we pass it the `global_step` variable, it will kindly take care of incrementing it. That's it!

Since AdaGrad, RMSProp, and Adam optimization automatically reduce the learning rate during training, it is not necessary to add an extra learning schedule. For other optimization algorithms, using exponential decay or performance scheduling can considerably speed up convergence.

Avoiding Overfitting Through Regularization

> With four parameters I can fit an elephant and with five I can make him wiggle his trunk.
>
> —John von Neumann, *cited by Enrico Fermi in Nature 427*

Deep neural networks typically have tens of thousands of parameters, sometimes even millions. With so many parameters, the network has an incredible amount of freedom and can fit a huge variety of complex datasets. But this great flexibility also means that it is prone to overfitting the training set.

19 "An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition," A. Senior et al. (2013).

With millions of parameters you can fit the whole zoo. In this section we will present some of the most popular regularization techniques for neural networks, and how to implement them with TensorFlow: early stopping, ℓ_1 and ℓ_2 regularization, dropout, max-norm regularization, and data augmentation.

Early Stopping

To avoid overfitting the training set, a great solution is early stopping (introduced in Chapter 4): just interrupt training when its performance on the validation set starts dropping.

One way to implement this with TensorFlow is to evaluate the model on a validation set at regular intervals (e.g., every 50 steps), and save a "winner" snapshot if it outperforms previous "winner" snapshots. Count the number of steps since the last "winner" snapshot was saved, and interrupt training when this number reaches some limit (e.g., 2,000 steps). Then restore the last "winner" snapshot.

Although early stopping works very well in practice, you can usually get much higher performance out of your network by combining it with other regularization techniques.

ℓ_1 and ℓ_2 Regularization

Just like you did in Chapter 4 for simple linear models, you can use ℓ_1 and ℓ_2 regularization to constrain a neural network's connection weights (but typically not its biases).

One way to do this using TensorFlow is to simply add the appropriate regularization terms to your cost function. For example, assuming you have just one hidden layer with weights `weights1` and one output layer with weights `weights2`, then you can apply ℓ_1 regularization like this:

```
[...] # construct the neural network
base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
reg_losses = tf.reduce_sum(tf.abs(weights1)) + tf.reduce_sum(tf.abs(weights2))
loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

However, if there are many layers, this approach is not very convenient. Fortunately, TensorFlow provides a better option. Many functions that create variables (such as `get_variable()` or `fully_connected()`) accept a `*_regularizer` argument for each created variable (e.g., `weights_regularizer`). You can pass any function that takes weights as an argument and returns the corresponding regularization loss. The `l1_regularizer()`, `l2_regularizer()`, and `l1_l2_regularizer()` functions return such functions. The following code puts all this together:

```
with arg_scope(
        [fully_connected],
```

```
        weights_regularizer=tf.contrib.layers.l1_regularizer(scale=0.01)):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="out")
```

This code creates a neural network with two hidden layers and one output layer, and it also creates nodes in the graph to compute the ℓ_1 regularization loss corresponding to each layer's weights. TensorFlow automatically adds these nodes to a special collection containing all the regularization losses. You just need to add these regularization losses to your overall loss, like this:

```
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([base_loss] + reg_losses, name="loss")
```

Don't forget to add the regularization losses to your overall loss, or else they will simply be ignored.

Dropout

The most popular regularization technique for deep neural networks is arguably *dropout*. It was proposed[20] by G. E. Hinton in 2012 and further detailed in a paper[21] by Nitish Srivastava et al., and it has proven to be highly successful: even the state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step (see Figure 11-9). The hyperparameter p is called the *dropout rate*, and it is typically set to 50%. After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss momentarily).

20 "Improving neural networks by preventing co-adaptation of feature detectors," G. Hinton et al. (2012).

21 "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," N. Srivastava et al. (2014).

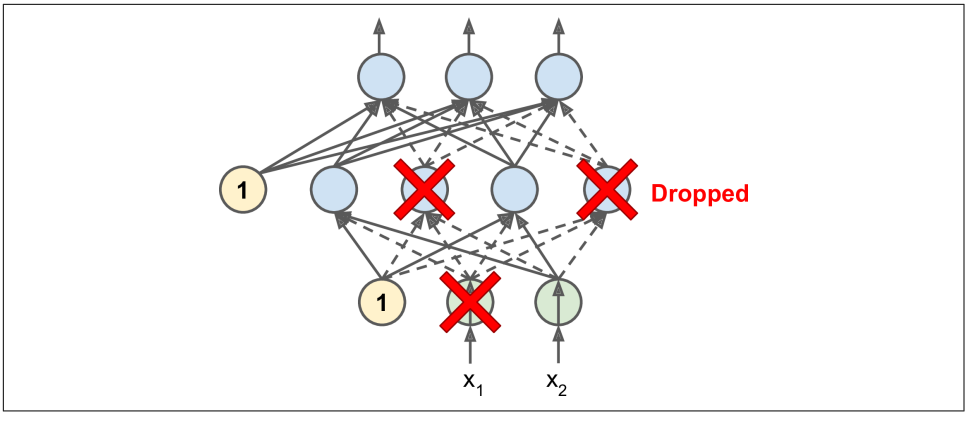

Figure 11-9. Dropout regularization

It is quite surprising at first that this rather brutal technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would obviously be forced to adapt its organization; it could not rely on any single person to fill in the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn't make much of a difference. It's unclear whether this idea would actually work for companies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end you get a more robust network that generalizes better.

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or absent, there is a total of 2^N possible networks (where N is the total number of droppable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run a 10,000 training steps, you have essentially trained 10,000 different neural networks (each with just one training instance). These neural networks are obviously not independent since they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.

There is one small but important technical detail. Suppose $p = 50$, in which case during testing a neuron will be connected to twice as many input neurons as it was (on average) during training. To compensate for this fact, we need to multiply each neu-

ron's input connection weights by 0.5 after training. If we don't, each neuron will get a total input signal roughly twice as large as what the network was trained on, and it is unlikely to perform well. More generally, we need to multiply each input connection weight by the *keep probability* (1 – *p*) after training. Alternatively, we can divide each neuron's output by the keep probability during training (these alternatives are not perfectly equivalent, but they work equally well).

To implement dropout using TensorFlow, you can simply apply the `dropout()` function to the input layer and to the output of every hidden layer. During training, this function randomly drops some items (setting them to 0) and divides the remaining items by the keep probability. After training, this function does nothing at all. The following code applies dropout regularization to our three-layer neural network:

```
from tensorflow.contrib.layers import dropout

[...]
is_training = tf.placeholder(tf.bool, shape=(), name='is_training')

keep_prob = 0.5
X_drop = dropout(X, keep_prob, is_training=is_training)

hidden1 = fully_connected(X_drop, n_hidden1, scope="hidden1")
hidden1_drop = dropout(hidden1, keep_prob, is_training=is_training)

hidden2 = fully_connected(hidden1_drop, n_hidden2, scope="hidden2")
hidden2_drop = dropout(hidden2, keep_prob, is_training=is_training)

logits = fully_connected(hidden2_drop, n_outputs, activation_fn=None,
                         scope="outputs")
```

You want to use the `dropout()` function in `tensorflow.contrib.layers`, not the one in `tensorflow.nn`. The first one turns off (no-op) when not training, which is what you want, while the second one does not.

Of course, just like you did earlier for Batch Normalization, you need to set `is_training` to `True` when training, and to `False` when testing.

If you observe that the model is overfitting, you can increase the dropout rate (i.e., reduce the `keep_prob` hyperparameter). Conversely, you should try decreasing the dropout rate (i.e., increasing `keep_prob`) if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones.

Dropout does tend to significantly slow down convergence, but it usually results in a much better model when tuned properly. So, it is generally well worth the extra time and effort.

Dropconnect is a variant of dropout where individual connections are dropped randomly rather than whole neurons. In general dropout performs better.

Max-Norm Regularization

Another regularization technique that is quite popular for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights $\mathbf{w}$ of the incoming connections such that $\| \mathbf{w} \|_2 \leq r$, where r is the max-norm hyperparameter and $\| \cdot \|_2$ is the ℓ_2 norm.

We typically implement this constraint by computing $\|\mathbf{w}\|_2$ after each training step and clipping $\mathbf{w}$ if needed ($\mathbf{w} \leftarrow \mathbf{w}\frac{r}{\| \mathbf{w} \|_2}$).

Reducing r increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the vanishing/exploding gradients problems (if you are not using Batch Normalization).

TensorFlow does not provide an off-the-shelf max-norm regularizer, but it is not too hard to implement. The following code creates a node `clip_weights` that will clip the `weights` variable along the second axis so that each row vector has a maximum norm of 1.0:

```
threshold = 1.0
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

You would then apply this operation after each training step, like so:

```
with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            clip_weights.eval()
```

You may wonder how to get access to the `weights` variable of each layer. For this you can simply use a variable scope like this:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")

with tf.variable_scope("hidden1", reuse=True):
    weights1 = tf.get_variable("weights")
```

Alternatively, you can use the root variable scope:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
```

```
[...]

with tf.variable_scope("", default_name="", reuse=True):  # root scope
    weights1 = tf.get_variable("hidden1/weights")
    weights2 = tf.get_variable("hidden2/weights")
```

If you don't know what the name of a variable is, you can either use TensorBoard to find out or simply use the `global_variables()` function and print out all the variable names:

```
for variable in tf.global_variables():
    print(variable.name)
```

Although the preceding solution should work fine, it is a bit messy. A cleaner solution is to create a `max_norm_regularizer()` function and use it just like the earlier `l1_regularizer()` function:

```
def max_norm_regularizer(threshold, axes=1, name="max_norm",
                         collection="max_norm"):
    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
        return None  # there is no regularization loss term
    return max_norm
```

This function returns a parametrized `max_norm()` function that you can use like any other regularizer:

```
max_norm_reg = max_norm_regularizer(threshold=1.0)
hidden1 = fully_connected(X, n_hidden1, scope="hidden1",
                          weights_regularizer=max_norm_reg)
```

Note that max-norm regularization does not require adding a regularization loss term to your overall loss function, so the `max_norm()` function returns `None`. But you still need to be able to run the `clip_weights` operation after each training step, so you need to be able to get a handle on it. This is why the `max_norm()` function adds the `clip_weights` node to a collection of max-norm clipping operations. You need to fetch these clipping operations and run them after each training step:

```
clip_all_weights = tf.get_collection("max_norm")

with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            sess.run(clip_all_weights)
```

Much cleaner code, isn't it?

Data Augmentation

One last regularization technique, data augmentation, consists of generating new training instances from existing ones, artificially boosting the size of the training set. This will reduce overfitting, making this a regularization technique. The trick is to generate realistic training instances; ideally, a human should not be able to tell which instances were generated and which ones were not. Moreover, simply adding white noise will not help; the modifications you apply should be learnable (white noise is not).

For example, if your model is meant to classify pictures of mushrooms, you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set (see Figure 11-10). This forces the model to be more tolerant to the position, orientation, and size of the mushrooms in the picture. If you want the model to be more tolerant to lighting conditions, you can similarly generate many images with various contrasts. Assuming the mushrooms are symmetrical, you can also flip the pictures horizontally. By combining these transformations you can greatly increase the size of your training set.

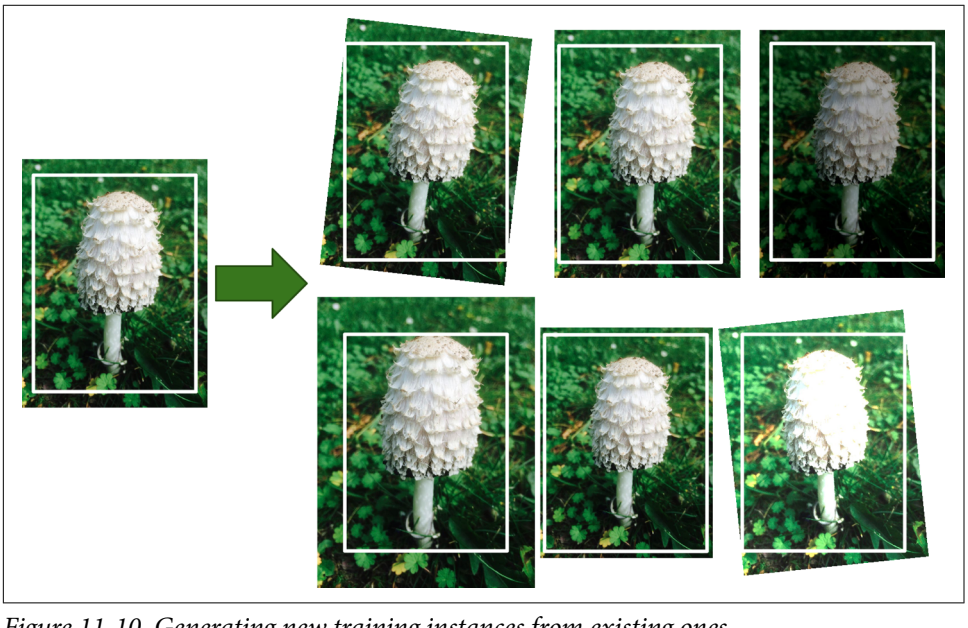

Figure 11-10. Generating new training instances from existing ones

It is often preferable to generate training instances on the fly during training rather than wasting storage space and network bandwidth. TensorFlow offers several image manipulation operations such as transposing (shifting), rotating, resizing, flipping, and cropping, as well as adjusting the brightness, contrast, saturation, and hue (see

the API documentation for more details). This makes it easy to implement data augmentation for image datasets.

Another powerful technique to train very deep neural networks is to add *skip connections* (a skip connection is when you add the input of a layer to the output of a higher layer). We will explore this idea in Chapter 13 when we talk about deep residual networks.

Practical Guidelines

In this chapter, we have covered a wide range of techniques and you may be wondering which ones you should use. The configuration in Table 11-2 will work fine in most cases.

Table 11-2. Default DNN configuration

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam
Learning rate schedule	None

Of course, you should try to reuse parts of a pretrained neural network if you can find one that solves a similar problem.

This default configuration may need to be tweaked:

- If you can't find a good learning rate (convergence was too slow, so you increased the training rate, and now convergence is fast but the network's accuracy is suboptimal), then you can try adding a learning schedule such as exponential decay.
- If your training set is a bit too small, you can implement data augmentation.
- If you need a sparse model, you can add some ℓ_1 regularization to the mix (and optionally zero out the tiny weights after training). If you need an even sparser model, you can try using FTRL instead of Adam optimization, along with ℓ_1 regularization.
- If you need a lightning-fast model at runtime, you may want to drop Batch Normalization, and possibly replace the ELU activation function with the leaky ReLU. Having a sparse model will also help.

With these guidelines, you are now ready to train very deep nets—well, if you are very patient, that is! If you use a single machine, you may have to wait for days or

even months for training to complete. In the next chapter we will discuss how to use distributed TensorFlow to train and run models across many servers and GPUs.

Exercises

1. Is it okay to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
2. Is it okay to initialize the bias terms to 0?
3. Name three advantages of the ELU activation function over ReLU.
4. In which cases would you want to use each of the following activation functions: ELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?
5. What may happen if you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using a `MomentumOptimizer`?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)?
8. Deep Learning.
   a. Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function.
   b. Using Adam optimization and early stopping, try training it on MNIST but only on digits 0 to 4, as we will use transfer learning for digits 5 to 9 in the next exercise. You will need a softmax output layer with five neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.
   c. Tune the hyperparameters using cross-validation and see what precision you can achieve.
   d. Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?
   e. Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?
9. Transfer learning.
   a. Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezes them, and replaces the softmax output layer with a fresh new one.
   b. Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?

c. Try caching the frozen layers, and train the model again: how much faster is it now?

d. Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?

e. Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?

10. Pretraining on an auxiliary task.

a. In this exercise you will build a DNN that compares two MNIST digit images and predicts whether they represent the same digit or not. Then you will reuse the lower layers of this network to train an MNIST classifier using very little training data. Start by building two DNNs (let's call them DNN A and B), both similar to the one you built earlier but without the output layer: each DNN should have five hidden layers of 100 neurons each, He initialization, and ELU activation. Next, add a single output layer on top of both DNNs. You should use TensorFlow's `concat()` function with `axis=1` to concatenate the outputs of both DNNs along the horizontal axis, then feed the result to the output layer. This output layer should contain a single neuron using the logistic activation function.

b. Split the MNIST training set in two sets: split #1 should containing 55,000 images, and split #2 should contain contain 5,000 images. Create a function that generates a training batch where each instance is a pair of MNIST images picked from split #1. Half of the training instances should be pairs of images that belong to the same class, while the other half should be images from different classes. For each pair, the training label should be 0 if the images are from the same class, or 1 if they are from different classes.

c. Train the DNN on this training set. For each image pair, you can simultaneously feed the first image to DNN A and the second image to DNN B. The whole network will gradually learn to tell whether two images belong to the same class or not.

d. Now create a new DNN by reusing and freezing the hidden layers of DNN A and adding a softmax output layer on with 10 neurons. Train this network on split #2 and see if you can achieve high performance despite having only 500 images per class.

Solutions to these exercises are available in Appendix A.

CHAPTER 12

Distributing TensorFlow Across Devices and Servers

In Chapter 11 we discussed several techniques that can considerably speed up training: better weight initialization, Batch Normalization, sophisticated optimizers, and so on. However, even with all of these techniques, training a large neural network on a single machine with a single CPU can take days or even weeks.

In this chapter we will see how to use TensorFlow to distribute computations across multiple devices (CPUs and GPUs) and run them in parallel (see Figure 12-1). First we will distribute computations across multiple devices on just one machine, then on multiple devices across multiple machines.

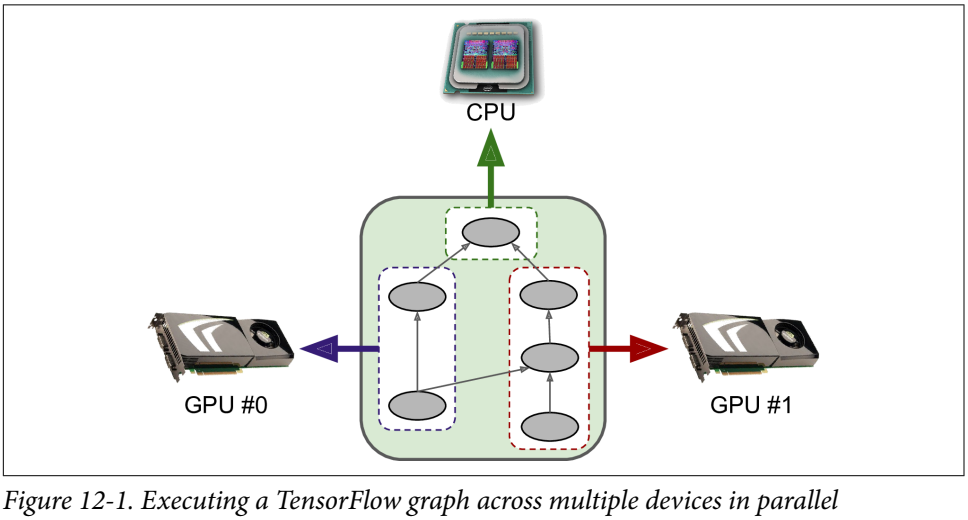

Figure 12-1. Executing a TensorFlow graph across multiple devices in parallel

TensorFlow's support of distributed computing is one of its main highlights compared to other neural network frameworks. It gives you full control over how to split (or replicate) your computation graph across devices and servers, and it lets you parallelize and synchronize operations in flexible ways so you can choose between all sorts of parallelization approaches.

We will look at some of the most popular approaches to parallelizing the execution and training of a neural network. Instead of waiting for weeks for a training algorithm to complete, you may end up waiting for just a few hours. Not only does this save an enormous amount of time, it also means that you can experiment with various models much more easily, and frequently retrain your models on fresh data.

Other great use cases of parallelization include exploring a much larger hyperparameter space when fine-tuning your model, and running large ensembles of neural networks efficiently.

But we must learn to walk before we can run. Let's start by parallelizing simple graphs across several GPUs on a single machine.

Multiple Devices on a Single Machine

You can often get a major performance boost simply by adding GPU cards to a single machine. In fact, in many cases this will suffice; you won't need to use multiple machines at all. For example, you can typically train a neural network just as fast using 8 GPUs on a single machine rather than 16 GPUs across multiple machines (due to the extra delay imposed by network communications in a multimachine setup).

In this section we will look at how to set up your environment so that TensorFlow can use multiple GPU cards on one machine. Then we will look at how you can distribute operations across available devices and execute them in parallel.

Installation

In order to run TensorFlow on multiple GPU cards, you first need to make sure your GPU cards have NVidia Compute Capability (greater or equal to 3.0). This includes Nvidia's Titan, Titan X, K20, and K40 cards (if you own another card, you can check its compatibility at *https://developer.nvidia.com/cuda-gpus*).

If you don't own any GPU cards, you can use a hosting service with GPU capability such as Amazon AWS. Detailed instructions to set up TensorFlow 0.9 with Python 3.5 on an Amazon AWS GPU instance are available in Žiga Avsec's helpful blog post. It should not be too hard to update it to the latest version of TensorFlow. Google also released a cloud service called *Cloud Machine Learning* to run TensorFlow graphs. In May 2016, they announced that their platform now includes servers equipped with *tensor processing units* (TPUs), processors specialized for Machine Learning that are much faster than GPUs for many ML tasks. Of course, another option is simply to buy your own GPU card. Tim Dettmers wrote a great blog post to help you choose, and he updates it fairly regularly.

You must then download and install the appropriate version of the CUDA and cuDNN libraries (CUDA 8.0 and cuDNN 5.1 if you are using the binary installation of TensorFlow 1.0.0), and set a few environment variables so TensorFlow knows where to find CUDA and cuDNN. The detailed installation instructions are likely to change fairly quickly, so it is best that you follow the instructions on TensorFlow's website.

Nvidia's *Compute Unified Device Architecture* library (CUDA) allows developers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration). Nvidia's *CUDA Deep Neural Network* library (cuDNN) is a GPU-accelerated library of primitives for DNNs. It provides optimized implementations of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling (see Chapter 13). It is part of Nvidia's Deep Learning SDK (note that it requires creating an Nvidia developer account in order to download it). TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see Figure 12-2).

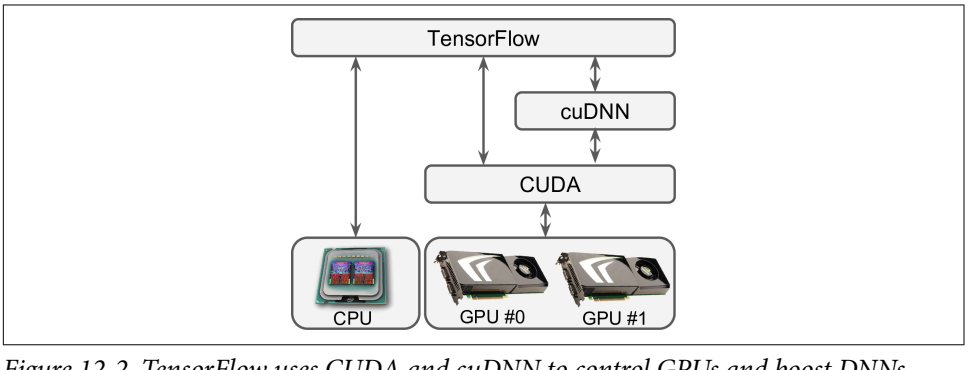

Figure 12-2. TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

You can use the `nvidia-smi` command to check that CUDA is properly installed. It lists the available GPU cards, as well as processes running on each card:

```
$ nvidia-smi
Wed Sep 16 09:50:03 2016
+-----------------------------------------------------+
| NVIDIA-SMI 352.63     Driver Version: 352.63        |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GRID K520           Off  | 0000:00:03.0     Off |                  N/A |
| N/A   27C    P8    17W / 125W |     11MiB /  4095MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Finally, you must install TensorFlow with GPU support. 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
```

Then install the appropriate GPU-enabled version of TensorFlow:

```
$ pip3 install --upgrade tensorflow-gpu
```

Now you can open up a Python shell and check that TensorFlow detects and uses CUDA and cuDNN properly by importing TensorFlow and creating a session:

```
>>> import tensorflow as tf
I [...]/dso_loader.cc:108] successfully opened CUDA library libcublas.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcudnn.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcufft.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcuda.so.1 locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcurand.so locally
>>> sess = tf.Session()
[...]
I [...]/gpu_init.cc:102] Found device 0 with properties:
name: GRID K520
major: 3 minor: 0 memoryClockRate (GHz) 0.797
pciBusID 0000:00:03.0
Total memory: 4.00GiB
Free memory: 3.95GiB
I [...]/gpu_init.cc:126] DMA: 0
I [...]/gpu_init.cc:136] 0:   Y
I [...]/gpu_device.cc:839] Creating TensorFlow device
(/gpu:0) -> (device: 0, name: GRID K520, pci bus id: 0000:00:03.0)
```

Looks good! TensorFlow detected the CUDA and cuDNN libraries, and it used the CUDA library to detect the GPU card (in this case an Nvidia Grid K520 card).

Managing the GPU RAM

By default TensorFlow automatically grabs all the RAM in all available GPUs the first time you run a graph, so you will not be able to start a second TensorFlow program while the first one is still running. If you try, you will get the following error:

```
E [...]/cuda_driver.cc:965] failed to allocate 3.66G (3928915968 bytes) from
device: CUDA_ERROR_OUT_OF_MEMORY
```

One solution is to run each process on different GPU cards. To do this, the simplest option is to set the `CUDA_VISIBLE_DEVICES` environment variable so that each process only sees the appropriate GPU cards. For example, you could start two programs like this:

```
$ CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# and in another terminal:
$ CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Program #1 will only see GPU cards 0 and 1 (numbered 0 and 1, respectively), and program #2 will only see GPU cards 2 and 3 (numbered 1 and 0, respectively). Everything will work fine (see Figure 12-3).

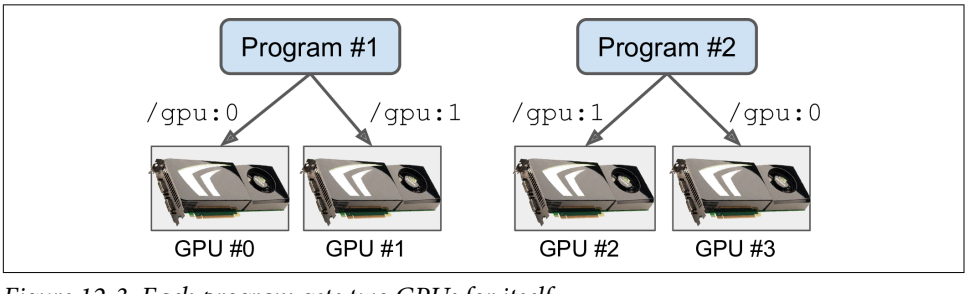

Figure 12-3. Each program gets two GPUs for itself

Another option is to tell TensorFlow to grab only a fraction of the memory. For example, to make TensorFlow grab only 40% of each GPU's memory, you must create a `ConfigProto` object, set its `gpu_options.per_process_gpu_memory_fraction` option to `0.4`, and create the session using this configuration:

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.Session(config=config)
```

Now two programs like this one can run in parallel using the same GPU cards (but not three, since $3 \times 0.4 > 1$). See Figure 12-4.

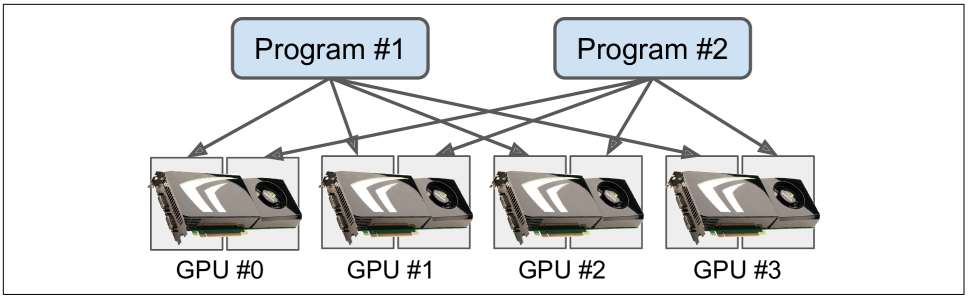

Figure 12-4. Each program gets all four GPUs, but with only 40% of the RAM each

If you run the `nvidia-smi` command while both programs are running, you should see that each process holds roughly 40% of the total RAM of each card:

```
$ nvidia-smi
[...]
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0      5231    C   python                                       1677MiB |
|    0      5262    C   python                                       1677MiB |
|    1      5231    C   python                                       1677MiB |
|    1      5262    C   python                                       1677MiB |
[...]
```

Yet another option is to tell TensorFlow to grab memory only when it needs it. To do this you must set `config.gpu_options.allow_growth` to `True`. However, TensorFlow never releases memory once it has grabbed it (to avoid memory fragmentation) so you may still run out of memory after a while. It may be harder to guarantee a deterministic behavior using this option, so in general you probably want to stick with one of the previous options.

Okay, now you have a working GPU-enabled TensorFlow installation. Let's see how to use it!

Placing Operations on Devices

The TensorFlow whitepaper[1] presents a friendly *dynamic placer* algorithm that automagically distributes operations across all available devices, taking into account things like the measured computation time in previous runs of the graph, estimations of the size of the input and output tensors to each operation, the amount of RAM available in each device, communication delay when transferring data in and out of

1 "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," Google Research (2015).

devices, hints and constraints from the user, and more. Unfortunately, this sophisticated algorithm is internal to Google; it was not released in the open source version of TensorFlow. The reason it was left out seems to be that in practice a small set of placement rules specified by the user actually results in more efficient placement than what the dynamic placer is capable of. However, the TensorFlow team is working on improving the dynamic placer, and perhaps it will eventually be good enough to be released.

Until then TensorFlow relies on the *simple placer*, which (as its name suggests) is very basic.

Simple placement

Whenever you run a graph, if TensorFlow needs to evaluate a node that is not placed on a device yet, it uses the simple placer to place it, along with all other nodes that are not placed yet. The simple placer respects the following rules:

- If a node was already placed on a device in a previous run of the graph, it is left on that device.
- Else, if the user *pinned* a node to a device (described next), the placer places it on that device.
- Else, it defaults to GPU #0, or the CPU if there is no GPU.

As you can see, placing operations on the appropriate device is mostly up to you. If you don't do anything, the whole graph will be placed on the default device. To pin nodes onto a device, you must create a device block using the `device()` function. For example, the following code pins the variable `a` and the constant `b` on the CPU, but the multiplication node `c` is not pinned on any device, so it will be placed on the default device:

```
with tf.device("/cpu:0"):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)

c = a * b
```

The `"/cpu:0"` device aggregates all CPUs on a multi-CPU system. There is currently no way to pin nodes on specific CPUs or to use just a subset of all CPUs.

Logging placements

Let's check that the simple placer respects the placement constraints we have just defined. For this you can set the `log_device_placement` option to `True`; this tells the placer to log a message whenever it places a node. For example:

```
>>> config = tf.ConfigProto()
>>> config.log_device_placement = True
>>> sess = tf.Session(config=config)
I [...] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GRID K520,
pci bus id: 0000:00:03.0)
[...]
>>> x.initializer.run(session=sess)
I [...] a: /job:localhost/replica:0/task:0/cpu:0
I [...] a/read: /job:localhost/replica:0/task:0/cpu:0
I [...] mul: /job:localhost/replica:0/task:0/gpu:0
I [...] a/Assign: /job:localhost/replica:0/task:0/cpu:0
I [...] b: /job:localhost/replica:0/task:0/cpu:0
I [...] a/initial_value: /job:localhost/replica:0/task:0/cpu:0
>>> sess.run(c)
12
```

The lines starting with `"I"` for Info are the log messages. When we create a session, TensorFlow logs a message to tell us that it has found a GPU card (in this case the Grid K520 card). Then the first time we run the graph (in this case when initializing the variable `a`), the simple placer is run and places each node on the device it was assigned to. As expected, the log messages show that all nodes are placed on `"/cpu:0"` except the multiplication node, which ends up on the default device `"/gpu:0"` (you can safely ignore the prefix `/job:localhost/replica:0/task:0` for now; we will talk about it in a moment). Notice that the second time we run the graph (to compute `c`), the placer is not used since all the nodes TensorFlow needs to compute `c` are already placed.

Dynamic placement function

When you create a device block, you can specify a function instead of a device name. TensorFlow will call this function for each operation it needs to place in the device block, and the function must return the name of the device to pin the operation on. For example, the following code pins all the variable nodes to `"/cpu:0"` (in this case just the variable `a`) and all other nodes to `"/gpu:0"`:

```
def variables_on_cpu(op):
    if op.type == "Variable":
        return "/cpu:0"
    else:
        return "/gpu:0"

with tf.device(variables_on_cpu):
    a = tf.Variable(3.0)
```

```
b = tf.constant(4.0)
c = a * b
```

You can easily implement more complex algorithms, such as pinning variables across GPUs in a round-robin fashion.

Operations and kernels

For a TensorFlow operation to run on a device, it needs to have an implementation for that device; this is called a *kernel*. Many operations have kernels for both CPUs and GPUs, but not all of them. For example, TensorFlow does not have a GPU kernel for integer variables, so the following code will fail when TensorFlow tries to place the variable `i` on GPU #0:

```
>>> with tf.device("/gpu:0"):
...     i = tf.Variable(3)
[...]
>>> sess.run(i.initializer)
Traceback (most recent call last):
[...]
tensorflow.python.framework.errors.InvalidArgumentError: Cannot assign a device
to node 'Variable': Could not satisfy explicit device specification
```

Note that TensorFlow infers that the variable must be of type `int32` since the initialization value is an integer. If you change the initialization value to `3.0` instead of `3`, or if you explicitly set `dtype=tf.float32` when creating the variable, everything will work fine.

Soft placement

By default, if you try to pin an operation on a device for which the operation has no kernel, you get the exception shown earlier when TensorFlow tries to place the operation on the device. If you prefer TensorFlow to fall back to the CPU instead, you can set the `allow_soft_placement` configuration option to `True`:

```
with tf.device("/gpu:0"):
    i = tf.Variable(3)

config = tf.ConfigProto()
config.allow_soft_placement = True
sess = tf.Session(config=config)
sess.run(i.initializer)  # the placer runs and falls back to /cpu:0
```

So far we have discussed how to place nodes on different devices. Now let's see how TensorFlow will run these nodes in parallel.

Parallel Execution

When TensorFlow runs a graph, it starts by finding out the list of nodes that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow

then starts evaluating the nodes with zero dependencies (i.e., source nodes). If these nodes are placed on separate devices, they obviously get evaluated in parallel. If they are placed on the same device, they get evaluated in different threads, so they may run in parallel too (in separate GPU threads or CPU cores).

TensorFlow manages a thread pool on each device to parallelize operations (see Figure 12-5). These are called the *inter-op thread pools*. Some operations have multithreaded kernels: they can use other thread pools (one per device) called the *intra-op thread pools*.

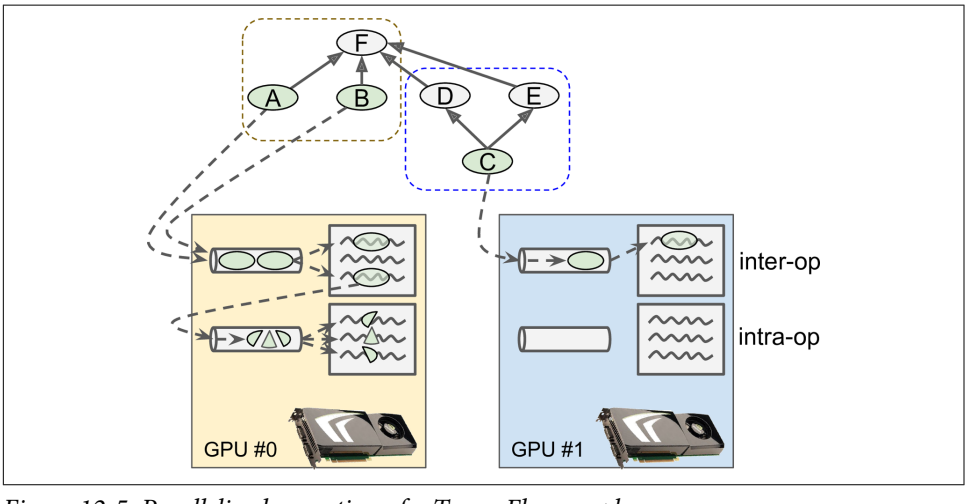

Figure 12-5. Parallelized execution of a TensorFlow graph

For example, in Figure 12-5, operations A, B, and C are source ops, so they can immediately be evaluated. Operations A and B are placed on GPU #0, so they are sent to this device's inter-op thread pool, and immediately evaluated in parallel. Operation A happens to have a multithreaded kernel; its computations are split in three parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU #1's inter-op thread pool.

As soon as operation C finishes, the dependency counters of operations D and E will be decremented and will both reach 0, so both operations will be sent to the inter-op thread pool to be executed.

You can control the number of threads per inter-op pool by setting the `inter_op_parallelism_threads` option. Note that the first session you start creates the inter-op thread pools. All other sessions will just reuse them unless you set the `use_per_session_threads` option to `True`. You can control the number of threads per intra-op pool by setting the `intra_op_parallelism_threads` option.

Control Dependencies

In some cases, it may be wise to postpone the evaluation of an operation even though all the operations it depends on have been executed. For example, if it uses a lot of memory but its value is needed only much further in the graph, it would be best to evaluate it at the last moment to avoid needlessly occupying RAM that other operations may need. Another example is a set of operations that depend on data located outside of the device. If they all run at the same time, they may saturate the device's communication bandwidth, and they will end up all waiting on I/O. Other operations that need to communicate data will also be blocked. It would be preferable to execute these communication-heavy operations sequentially, allowing the device to perform other operations in parallel.

To postpone evaluation of some nodes, a simple solution is to add *control dependencies*. For example, the following code tells TensorFlow to evaluate `x` and `y` only after `a` and `b` have been evaluated:

```
a = tf.constant(1.0)
b = a + 2.0

with tf.control_dependencies([a, b]):
    x = tf.constant(3.0)
    y = tf.constant(4.0)

z = x + y
```

Obviously, since `z` depends on `x` and `y`, evaluating `z` also implies waiting for `a` and `b` to be evaluated, even though it is not explicitly in the `control_dependencies()` block. Also, since `b` depends on `a`, we could simplify the preceding code by just creating a control dependency on `[b]` instead of `[a, b]`, but in some cases "explicit is better than implicit."

Great! Now you know:

- How to place operations on multiple devices in any way you please
- How these operations get executed in parallel
- How to create control dependencies to optimize parallel execution

It's time to distribute computations across multiple servers!

Multiple Devices Across Multiple Servers

To run a graph across multiple servers, you first need to define a *cluster*. A cluster is composed of one or more TensorFlow servers, called *tasks*, typically spread across several machines (see Figure 12-6). Each task belongs to a *job*. A job is just a named group of tasks that typically have a common role, such as keeping track of the model

parameters (such a job is usually named `"ps"` for *parameter server*), or performing computations (such a job is usually named `"worker"`).

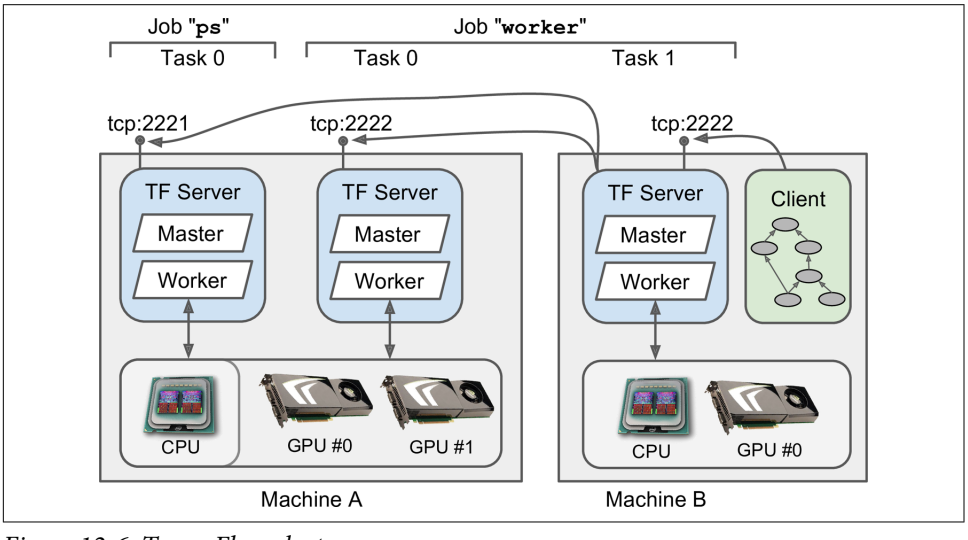

Figure 12-6. TensorFlow cluster

The following *cluster specification* defines two jobs, `"ps"` and `"worker"`, containing one task and two tasks, respectively. In this example, machine A hosts two TensorFlow servers (i.e., tasks), listening on different ports: one is part of the `"ps"` job, and the other is part of the `"worker"` job. Machine B just hosts one TensorFlow server, part of the `"worker"` job.

```
cluster_spec = tf.train.ClusterSpec({
    "ps": [
        "machine-a.example.com:2221",  # /job:ps/task:0
    ],
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222",  # /job:worker/task:1
    ]})
```

To start a TensorFlow server, you must create a `Server` object, passing it the cluster specification (so it can communicate with other servers) and its own job name and task number. For example, to start the first worker task, you would run the following code on machine A:

```
server = tf.train.Server(cluster_spec, job_name="worker", task_index=0)
```

It is usually simpler to just run one task per machine, but the previous example demonstrates that TensorFlow allows you to run multiple tasks on the same machine if

you want.[2] If you have several servers on one machine, you will need to ensure that they don't all try to grab all the RAM of every GPU, as explained earlier. For example, in Figure 12-6 the `"ps"` task does not see the GPU devices, since presumably its process was launched with `CUDA_VISIBLE_DEVICES=""`. Note that the CPU is shared by all tasks located on the same machine.

If you want the process to do nothing other than run the TensorFlow server, you can block the main thread by telling it to wait for the server to finish using the `join()` method (otherwise the server will be killed as soon as your main thread exits). Since there is currently no way to stop the server, this will actually block forever:

```
server.join()  # blocks until the server stops (i.e., never)
```

Opening a Session

Once all the tasks are up and running (doing nothing yet), you can open a session on any of the servers, from a client located in any process on any machine (even from a process running one of the tasks), and use that session like a regular local session. For example:

```
a = tf.constant(1.0)
b = a + 2
c = a * 3

with tf.Session("grpc://machine-b.example.com:2222") as sess:
    print(c.eval())  # 9.0
```

This client code first creates a simple graph, then opens a session on the TensorFlow server located on machine B (which we will call the *master*), and instructs it to evaluate `c`. The master starts by placing the operations on the appropriate devices. In this example, since we did not pin any operation on any device, the master simply places them all on its own default device—in this case, machine B's GPU device. Then it just evaluates `c` as instructed by the client, and it returns the result.

The Master and Worker Services

The client uses the *gRPC* protocol (*Google Remote Procedure Call*) to communicate with the server. This is an efficient open source framework to call remote functions and get their outputs across a variety of platforms and languages.[3] It is based on HTTP2, which opens a connection and leaves it open during the whole session, allowing efficient bidirectional communication once the connection is established.

2 You can even start multiple tasks in the same process. It may be useful for tests, but it is not recommended in production.

3 It is the next version of Google's internal *Stubby* service, which Google has used successfully for over a decade. See *http://grpc.io/* for more details.

Data is transmitted in the form of *protocol buffers*, another open source Google technology. This is a lightweight binary data interchange format.

All servers in a TensorFlow cluster may communicate with any other server in the cluster, so make sure to open the appropriate ports on your firewall.

Every TensorFlow server provides two services: the *master service* and the *worker service*. The master service allows clients to open sessions and use them to run graphs. It coordinates the computations across tasks, relying on the worker service to actually execute computations on other tasks and get their results.

This architecture gives you a lot of flexibility. One client can connect to multiple servers by opening multiple sessions in different threads. One server can handle multiple sessions simultaneously from one or more clients. You can run one client per task (typically within the same process), or just one client to control all tasks. All options are open.

Pinning Operations Across Tasks

You can use device blocks to pin operations on any device managed by any task, by specifying the job name, task index, device type, and device index. For example, the following code pins `a` to the CPU of the first task in the `"ps"` job (that's the CPU on machine A), and it pins `b` to the second GPU managed by the first task of the `"worker"` job (that's GPU #1 on machine A). Finally, `c` is not pinned to any device, so the master places it on its own default device (machine B's GPU #0 device).

```
with tf.device("/job:ps/task:0/cpu:0")
    a = tf.constant(1.0)

with tf.device("/job:worker/task:0/gpu:1")
    b = a + 2

c = a + b
```

As earlier, if you omit the device type and index, TensorFlow will default to the task's default device; for example, pinning an operation to `"/job:ps/task:0"` will place it on the default device of the first task of the `"ps"` job (machine A's CPU). If you also omit the task index (e.g., `"/job:ps"`), TensorFlow defaults to `"/task:0"`. If you omit the job name and the task index, TensorFlow defaults to the session's master task.

Sharding Variables Across Multiple Parameter Servers

As we will see shortly, a common pattern when training a neural network on a distributed setup is to store the model parameters on a set of parameter servers (i.e., the tasks in the `"ps"` job) while other tasks focus on computations (i.e., the tasks in the `"worker"` job). For large models with millions of parameters, it is useful to shard these parameters across multiple parameter servers, to reduce the risk of saturating a single parameter server's network card. If you were to manually pin every variable to a different parameter server, it would be quite tedious. Fortunately, TensorFlow provides the `replica_device_setter()` function, which distributes variables across all the `"ps"` tasks in a round-robin fashion. For example, the following code pins five variables to two parameter servers:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2):
    v1 = tf.Variable(1.0)  # pinned to /job:ps/task:0
    v2 = tf.Variable(2.0)  # pinned to /job:ps/task:1
    v3 = tf.Variable(3.0)  # pinned to /job:ps/task:0
    v4 = tf.Variable(4.0)  # pinned to /job:ps/task:1
    v5 = tf.Variable(5.0)  # pinned to /job:ps/task:0
```

Instead of passing the number of `ps_tasks`, you can pass the cluster spec `cluster=cluster_spec` and TensorFlow will simply count the number of tasks in the `"ps"` job.

If you create other operations in the block, beyond just variables, TensorFlow automatically pins them to `"/job:worker"`, which will default to the first device managed by the first task in the `"worker"` job. You can pin them to another device by setting the `worker_device` parameter, but a better approach is to use embedded device blocks. An inner device block can override the job, task, or device defined in an outer block. For example:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
    v1 = tf.Variable(1.0)  # pinned to /job:ps/task:0 (+ defaults to /cpu:0)
    v2 = tf.Variable(2.0)  # pinned to /job:ps/task:1 (+ defaults to /cpu:0)
    v3 = tf.Variable(3.0)  # pinned to /job:ps/task:0 (+ defaults to /cpu:0)
    [...]
    s = v1 + v2            # pinned to /job:worker (+ defaults to task:0/gpu:0)
    with tf.device("/gpu:1"):
        p1 = 2 * s         # pinned to /job:worker/gpu:1 (+ defaults to /task:0)
        with tf.device("/task:1"):
            p2 = 3 * s     # pinned to /job:worker/task:1/gpu:1
```

This example assumes that the parameter servers are CPU-only, which is typically the case since they only need to store and communicate parameters, not perform intensive computations.

Sharing State Across Sessions Using Resource Containers

When you are using a plain *local session* (not the distributed kind), each variable's state is managed by the session itself; as soon as it ends, all variable values are lost. Moreover, multiple local sessions cannot share any state, even if they both run the same graph; each session has its own copy of every variable (as we discussed in Chapter 9). In contrast, when you are using *distributed sessions*, variable state is managed by *resource containers* located on the cluster itself, not by the sessions. So if you create a variable named `x` using one client session, it will automatically be available to any other session on the same cluster (even if both sessions are connected to a different server). For example, consider the following client code:

```
# simple_client.py
import tensorflow as tf
import sys

x = tf.Variable(0.0, name="x")
increment_x = tf.assign(x, x + 1)

with tf.Session(sys.argv[1]) as sess:
    if sys.argv[2:]==["init"]:
        sess.run(x.initializer)
    sess.run(increment_x)
    print(x.eval())
```

Let's suppose you have a TensorFlow cluster up and running on machines A and B, port 2222. You could launch the client, have it open a session with the server on machine A, and tell it to initialize the variable, increment it, and print its value by launching the following command:

```
$ python3 simple_client.py grpc://machine-a.example.com:2222 init
1.0
```

Now if you launch the client with the following command, it will connect to the server on machine B and magically reuse the same variable `x` (this time we don't ask the server to initialize the variable):

```
$ python3 simple_client.py grpc://machine-b.example.com:2222
2.0
```

This feature cuts both ways: it's great if you want to share variables across multiple sessions, but if you want to run completely independent computations on the same cluster you will have to be careful not to use the same variable names by accident. One way to ensure that you won't have name clashes is to wrap all of your construction phase inside a variable scope with a unique name for each computation, for example:

```
with tf.variable_scope("my_problem_1"):
    [...] # Construction phase of problem 1
```

A better option is to use a container block:

```
with tf.container("my_problem_1"):
    [...] # Construction phase of problem 1
```

This will use a container dedicated to problem #1, instead of the default one (whose name is an empty string `""`). One advantage is that variable names remain nice and short. Another advantage is that you can easily reset a named container. For example, the following command will connect to the server on machine A and ask it to reset the container named `"my_problem_1"`, which will free all the resources this container used (and also close all sessions open on the server). Any variable managed by this container must be initialized before you can use it again:

```
tf.Session.reset("grpc://machine-a.example.com:2222", ["my_problem_1"])
```

Resource containers make it easy to share variables across sessions in flexible ways. For example, Figure 12-7 shows four clients running different graphs on the same cluster, but sharing some variables. Clients A and B share the same variable `x` managed by the default container, while clients C and D share another variable named `x` managed by the container named `"my_problem_1"`. Note that client C even uses variables from both containers.

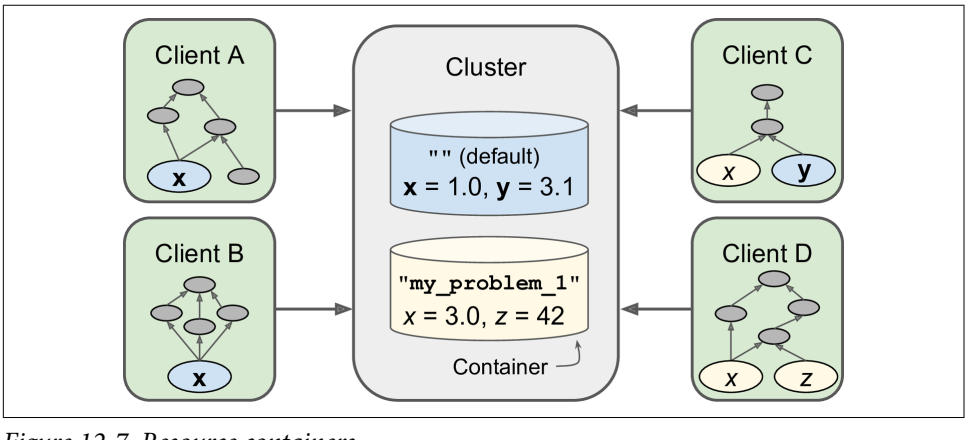

Figure 12-7. Resource containers

Resource containers also take care of preserving the state of other stateful operations, namely queues and readers. Let's take a look at queues first.

Asynchronous Communication Using TensorFlow Queues

Queues are another great way to exchange data between multiple sessions; for example, one common use case is to have a client create a graph that loads the training data and pushes it into a queue, while another client creates a graph that pulls the data from the queue and trains a model (see Figure 12-8). This can speed up training con-

siderably because the training operations don't have to wait for the next mini-batch at every step.

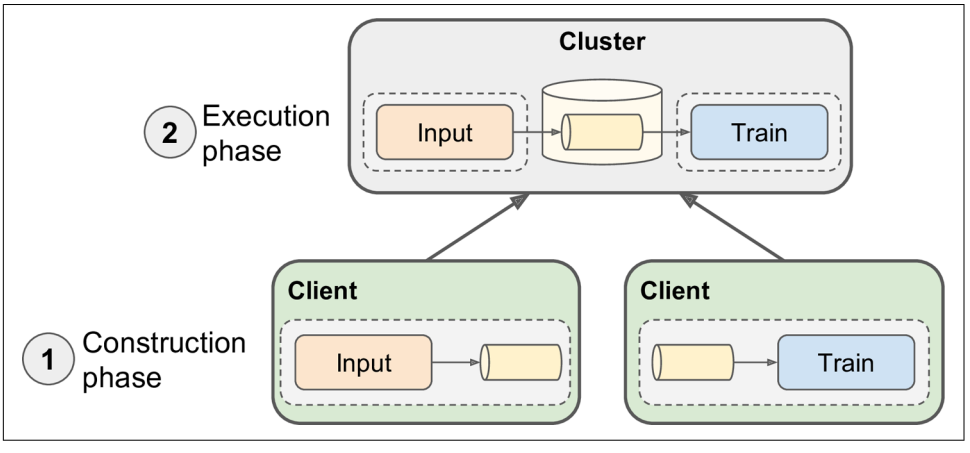

Figure 12-8. Using queues to load the training data asynchronously

TensorFlow provides various kinds of queues. The simplest kind is the *first-in first-out* (*FIFO*) queue. For example, the following code creates a FIFO queue that can store up to 10 tensors containing two float values each:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.float32], shapes=[[2]],
                 name="q", shared_name="shared_q")
```

To share variables across sessions, all you had to do was to specify the same name and container on both ends. With queues TensorFlow does not use the `name` attribute but instead uses `shared_name`, so it is important to specify it (even if it is the same as the `name`). And, of course, use the same container.

Enqueuing data

To push data to a queue, you must create an `enqueue` operation. For example, the following code pushes three training instances to the queue:

```
# training_data_loader.py
import tensorflow as tf

q = [...]
training_instance = tf.placeholder(tf.float32, shape=(2))
enqueue = q.enqueue([training_instance])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
   sess.run(enqueue, feed_dict={training_instance: [1., 2.]})
   sess.run(enqueue, feed_dict={training_instance: [3., 4.]})
   sess.run(enqueue, feed_dict={training_instance: [5., 6.]})
```

Instead of enqueuing instances one by one, you can enqueue several at a time using an `enqueue_many` operation:

```
[...]
training_instances = tf.placeholder(tf.float32, shape=(None, 2))
enqueue_many = q.enqueue([training_instances])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enqueue_many,
             feed_dict={training_instances: [[1., 2.], [3., 4.], [5., 6.]]})
```

Both examples enqueue the same three tensors to the queue.

Dequeuing data

To pull the instances out of the queue, on the other end, you need to use a `dequeue` operation:

```
# trainer.py
import tensorflow as tf

q = [...]
dequeue = q.dequeue()

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue))  # [1., 2.]
    print(sess.run(dequeue))  # [3., 4.]
    print(sess.run(dequeue))  # [5., 6.]
```

In general you will want to pull a whole mini-batch at once, instead of pulling just one instance at a time. To do so, you must use a `dequeue_many` operation, specifying the mini-batch size:

```
[...]
batch_size = 2
dequeue_mini_batch= q.dequeue_many(batch_size)

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue_mini_batch))  # [[1., 2.], [4., 5.]]
    print(sess.run(dequeue_mini_batch))  # blocked waiting for another instance
```

When a queue is full, the enqueue operation will block until items are pulled out by a dequeue operation. Similarly, when a queue is empty (or you are using `dequeue_many()` and there are fewer items than the mini-batch size), the dequeue operation will block until enough items are pushed into the queue using an enqueue operation.

Queues of tuples

Each item in a queue can be a tuple of tensors (of various types and shapes) instead of just a single tensor. For example, the following queue stores pairs of tensors, one of type `int32` and shape `()`, and the other of type `float32` and shape `[3,2]`:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.int32, tf.float32], shapes=[[],[3,2]],
                 name="q", shared_name="shared_q")
```

The enqueue operation must be given pairs of tensors (note that each pair represents only one item in the queue):

```
a = tf.placeholder(tf.int32, shape=())
b = tf.placeholder(tf.float32, shape=(3, 2))
enqueue = q.enqueue((a, b))

with tf.Session([...]) as sess:
    sess.run(enqueue, feed_dict={a: 10, b:[[1., 2.], [3., 4.], [5., 6.]]})
    sess.run(enqueue, feed_dict={a: 11, b:[[2., 4.], [6., 8.], [0., 2.]]})
    sess.run(enqueue, feed_dict={a: 12, b:[[3., 6.], [9., 2.], [5., 8.]]})
```

On the other end, the `dequeue()` function now creates a pair of dequeue operations:

```
dequeue_a, dequeue_b = q.dequeue()
```

In general, you should run these operations together:

```
with tf.Session([...]) as sess:
    a_val, b_val = sess.run([dequeue_a, dequeue_b])
    print(a_val) # 10
    print(b_val) # [[1., 2.], [3., 4.], [5., 6.]]
```

If you run `dequeue_a` on its own, it will dequeue a pair and return only the first element; the second element will be lost (and similarly, if you run `dequeue_b` on its own, the first element will be lost).

The `dequeue_many()` function also returns a pair of operations:

```
batch_size = 2
dequeue_as, dequeue_bs = q.dequeue_many(batch_size)
```

You can use it as you would expect:

```
with tf.Session([...]) as sess:
    a, b = sess.run([dequeue_a, dequeue_b])
    print(a) # [10, 11]
    print(b) # [[[1., 2.], [3., 4.], [5., 6.]], [[2., 4.], [6., 8.], [0., 2.]]]
    a, b = sess.run([dequeue_a, dequeue_b])  # blocked waiting for another pair
```

Closing a queue

It is possible to close a queue to signal to the other sessions that no more data will be enqueued:

```
close_q = q.close()

with tf.Session([...]) as sess:
    [...]
    sess.run(close_q)
```

Subsequent executions of `enqueue` or `enqueue_many` operations will raise an exception. By default, any pending enqueue request will be honored, unless you call `q.close(cancel_pending_enqueues=True)`.

Subsequent executions of `dequeue` or `dequeue_many` operations will continue to succeed as long as there are items in the queue, but they will fail when there are not enough items left in the queue. If you are using a `dequeue_many` operation and there are a few instances left in the queue, but fewer than the mini-batch size, they will be lost. You may prefer to use a `dequeue_up_to` operation instead; it behaves exactly like `dequeue_many` except when a queue is closed and there are fewer than `batch_size` instances left in the queue, in which case it just returns them.

RandomShuffleQueue

TensorFlow also supports a couple more types of queues, including `RandomShuffleQueue`, which can be used just like a `FIFOQueue` except that items are dequeued in a random order. This can be useful to shuffle training instances at each epoch during training. First, let's create the queue:

```
q = tf.RandomShuffleQueue(capacity=50, min_after_dequeue=10,
                          dtypes=[tf.float32], shapes=[()],
                          name="q", shared_name="shared_q")
```

The `min_after_dequeue` specifies the minimum number of items that must remain in the queue after a dequeue operation. This ensures that there will be enough instances in the queue to have enough randomness (once the queue is closed, the `min_after_dequeue` limit is ignored). Now suppose that you enqueued 22 items in this queue (floats `1.` to `22.`). Here is how you could dequeue them:

```
dequeue = q.dequeue_many(5)

with tf.Session([...]) as sess:
   print(sess.run(dequeue)) # [ 20.  15.  11.  12.   4.]   (17 items left)
   print(sess.run(dequeue)) # [  5.  13.   6.   0.  17.]   (12 items left)
   print(sess.run(dequeue)) # 12 - 5 < 10: blocked waiting for 3 more instances
```

PaddingFifoQueue

A `PaddingFIFOQueue` can also be used just like a `FIFOQueue` except that it accepts tensors of variable sizes along any dimension (but with a fixed rank). When you are dequeuing them with a `dequeue_many` or `dequeue_up_to` operation, each tensor is padded with zeros along every variable dimension to make it the same size as the largest tensor in the mini-batch. For example, you could enqueue 2D tensors (matrices) of arbitrary sizes:

```
q = tf.PaddingFIFOQueue(capacity=50, dtypes=[tf.float32], shapes=[(None, None)]
                        name="q", shared_name="shared_q")
v = tf.placeholder(tf.float32, shape=(None, None))
enqueue = q.enqueue([v])

with tf.Session([...]) as sess:
   sess.run(enqueue, feed_dict={v: [[1., 2.], [3., 4.], [5., 6.]]})       # 3x2
   sess.run(enqueue, feed_dict={v: [[1.]]})                               # 1x1
   sess.run(enqueue, feed_dict={v: [[7., 8., 9., 5.], [6., 7., 8., 9.]]}) # 2x4
```

If we just dequeue one item at a time, we get the exact same tensors that were enqueued. But if we dequeue several items at a time (using `dequeue_many()` or `dequeue_up_to()`), the queue automatically pads the tensors appropriately. For example, if we dequeue all three items at once, all tensors will be padded with zeros to become 3 × 4 tensors, since the maximum size for the first dimension is 3 (first item) and the maximum size for the second dimension is 4 (third item):

```
>>> q = [...]
>>> dequeue = q.dequeue_many(3)
>>> with tf.Session([...]) as sess:
...     print(sess.run(dequeue))
[[[ 1.  2.  0.  0.]
  [ 3.  4.  0.  0.]
  [ 5.  6.  0.  0.]]

 [[ 1.  0.  0.  0.]
  [ 0.  0.  0.  0.]
  [ 0.  0.  0.  0.]]

 [[ 7.  8.  9.  5.]
  [ 6.  7.  8.  9.]
  [ 0.  0.  0.  0.]]]
```

This type of queue can be useful when you are dealing with variable length inputs, such as sequences of words (see Chapter 14).

Okay, now let's pause for a second: so far you have learned to distribute computations across multiple devices and servers, share variables across sessions, and communicate asynchronously using queues. Before you start training neural networks, though, there's one last topic we need to discuss: how to efficiently load training data.

Loading Data Directly from the Graph

So far we have assumed that the clients would load the training data and feed it to the cluster using placeholders. This is simple and works quite well for simple setups, but it is rather inefficient since it transfers the training data several times:

1. From the filesystem to the client
2. From the client to the master task
3. Possibly from the master task to other tasks where the data is needed

It gets worse if you have several clients training various neural networks using the same training data (for example, for hyperparameter tuning): if every client loads the data simultaneously, you may end up even saturating your file server or the network's bandwidth.

Preload the data into a variable

For datasets that can fit in memory, a better option is to load the training data once and assign it to a variable, then just use that variable in your graph. This is called *preloading* the training set. This way the data will be transferred only once from the client to the cluster (but it may still need to be moved around from task to task depending on which operations need it). The following code shows how to load the full training set into a variable:

```
training_set_init = tf.placeholder(tf.float32, shape=(None, n_features))
training_set = tf.Variable(training_set_init, trainable=False, collections=[],
                           name="training_set")

with tf.Session([...]) as sess:
    data = [...]  # load the training data from the datastore
    sess.run(training_set.initializer, feed_dict={training_set_init: data})
```

You must set `trainable=False` so the optimizers don't try to tweak this variable. You should also set `collections=[]` to ensure that this variable won't get added to the `GraphKeys.GLOBAL_VARIABLES` collection, which is used for saving and restoring checkpoints.

This example assumes that all of your training set (including the labels) consists only of `float32` values. If that's not the case, you will need one variable per type.

Reading the training data directly from the graph

If the training set does not fit in memory, a good solution is to use *reader operations*: these are operations capable of reading data directly from the filesystem. This way the

training data never needs to flow through the clients at all. TensorFlow provides readers for various file formats:

- CSV
- Fixed-length binary records
- TensorFlow's own `TFRecords` format, based on protocol buffers

Let's look at a simple example reading from a CSV file (for other formats, please check out the API documentation). Suppose you have file named *my_test.csv* that contains training instances, and you want to create operations to read it. Suppose it has the following content, with two float features `x1` and `x2` and one integer `target` representing a binary class:

```
x1,  x2,  target
1. , 2. , 0
4. , 5  , 1
7. ,    , 0
```

First, let's create a `TextLineReader` to read this file. A `TextLineReader` opens a file (once we tell it which one to open) and reads lines one by one. It is a stateful operation, like variables and queues: it preserves its state across multiple runs of the graph, keeping track of which file it is currently reading and what its current position is in this file.

```
reader = tf.TextLineReader(skip_header_lines=1)
```

Next, we create a queue that the reader will pull from to know which file to read next. We also create an enqueue operation and a placeholder to push any filename we want to the queue, and we create an operation to close the queue once we have no more files to read:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()
```

Now we are ready to create a `read` operation that will read one record (i.e., a line) at a time and return a key/value pair. The key is the record's unique identifier—a string composed of the filename, a colon (`:`), and the line number—and the value is simply a string containing the content of the line:

```
key, value = reader.read(filename_queue)
```

We have all we need to read the file line by line! But we are not quite done yet—we need to parse this string to get the features and target:

```
x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1]])
features = tf.stack([x1, x2])
```

The first line uses TensorFlow's CSV parser to extract the values from the current line. The default values are used when a field is missing (in this example the third training instance's `x2` feature), and they are also used to determine the type of each field (in this case two floats and one integer).

Finally, we can push this training instance and its target to a `RandomShuffleQueue` that we will share with the training graph (so it can pull mini-batches from it), and we create an operation to close that queue when we are done pushing instances to it:

```
instance_queue = tf.RandomShuffleQueue(
    capacity=10, min_after_dequeue=2,
    dtypes=[tf.float32, tf.int32], shapes=[[2],[]],
    name="instance_q", shared_name="shared_instance_q")
enqueue_instance = instance_queue.enqueue([features, target])
close_instance_queue = instance_queue.close()
```

Wow! That was a lot of work just to read a file. Plus we only created the graph, so now we need to run it:

```
with tf.Session([...]) as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    try:
        while True:
            sess.run(enqueue_instance)
    except tf.errors.OutOfRangeError as ex:
        pass # no more records in the current file and no more files to read
    sess.run(close_instance_queue)
```

First we open the session, and then we enqueue the filename `"my_test.csv"` and immediately close that queue since we will not enqueue any more filenames. Then we run an infinite loop to enqueue instances one by one. The `enqueue_instance` depends on the reader reading the next line, so at every iteration a new record is read until it reaches the end of the file. At that point it tries to read the filename queue to know which file to read next, and since the queue is closed it throws an `OutOfRangeError` exception (if we did not close the queue, it would just remain blocked until we pushed another filename or closed the queue). Lastly, we close the instance queue so that the training operations pulling from it won't get blocked forever. Figure 12-9 summarizes what we have learned; it represents a typical graph for reading training instances from a set of CSV files.

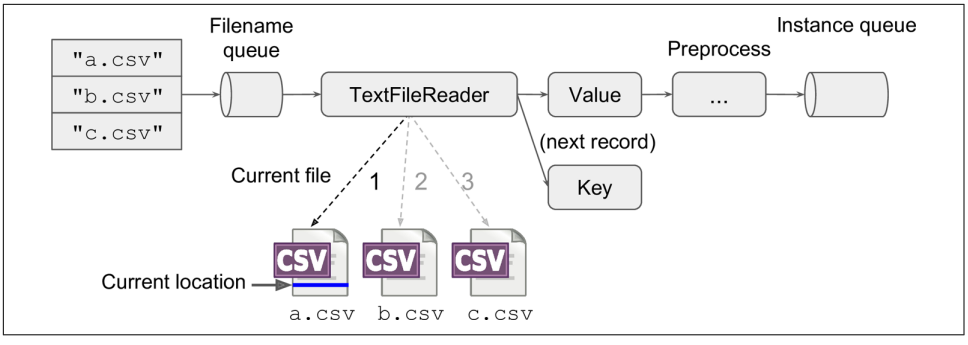

Figure 12-9. A graph dedicated to reading training instances from CSV files

In the training graph, you need to create the shared instance queue and simply dequeue mini-batches from it:

```
instance_queue = tf.RandomShuffleQueue([...], shared_name="shared_instance_q")
mini_batch_instances, mini_batch_targets = instance_queue.dequeue_up_to(2)
[...] # use the mini_batch instances and targets to build the training graph
training_op = [...]

with tf.Session([...]) as sess:
    try:
        for step in range(max_steps):
            sess.run(training_op)
    except tf.errors.OutOfRangeError as ex:
        pass # no more training instances
```

In this example, the first mini-batch will contain the first two instances of the CSV file, and the second mini-batch will contain the last instance.

TensorFlow queues don't handle sparse tensors well, so if your training instances are sparse you should parse the records after the instance queue.

This architecture will only use one thread to read records and push them to the instance queue. You can get a much higher throughput by having multiple threads read simultaneously from multiple files using multiple readers. Let's see how.

Multithreaded readers using a Coordinator and a QueueRunner

To have multiple threads read instances simultaneously, you could create Python threads (using the `threading` module) and manage them yourself. However, TensorFlow provides some tools to make this simpler: the `Coordinator` class and the `QueueRunner` class.

A `Coordinator` is a very simple object whose sole purpose is to coordinate stopping multiple threads. First you create a `Coordinator`:

```
coord = tf.train.Coordinator()
```

Then you give it to all threads that need to stop jointly, and their main loop looks like this:

```
while not coord.should_stop():
    [...] # do something
```

Any thread can request that every thread stop by calling the `Coordinator`'s `request_stop()` method:

```
coord.request_stop()
```

Every thread will stop as soon as it finishes its current iteration. You can wait for all of the threads to finish by calling the `Coordinator`'s `join()` method, passing it the list of threads:

```
coord.join(list_of_threads)
```

A `QueueRunner` runs multiple threads that each run an enqueue operation repeatedly, filling up a queue as fast as possible. As soon as the queue is closed, the next thread that tries to push an item to the queue will get an `OutOfRangeError`; this thread catches the error and immediately tells other threads to stop using a `Coordinator`. The following code shows how you can use a `QueueRunner` to have five threads reading instances simultaneously and pushing them to an instance queue:

```
[...] # same construction phase as earlier
queue_runner = tf.train.QueueRunner(instance_queue, [enqueue_instance] * 5)

with tf.Session() as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    coord = tf.train.Coordinator()
    enqueue_threads = queue_runner.create_threads(sess, coord=coord, start=True)
```

The first line creates the `QueueRunner` and tells it to run five threads, all running the same `enqueue_instance` operation repeatedly. Then we start a session and we enqueue the name of the files to read (in this case just `"my_test.csv"`). Next we create a `Coordinator` that the `QueueRunner` will use to stop gracefully, as just explained. Finally, we tell the `QueueRunner` to create the threads and start them. The threads will read all training instances and push them to the instance queue, and then they will all stop gracefully.

This will be a bit more efficient than earlier, but we can do better. Currently all threads are reading from the same file. We can make them read simultaneously from separate files instead (assuming the training data is sharded across multiple CSV files) by creating multiple readers (see Figure 12-10).

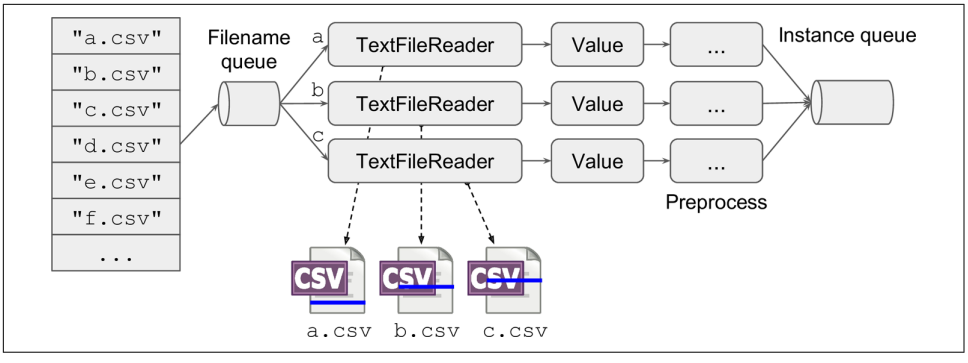

Figure 12-10. Reading simultaneously from multiple files

For this we need to write a small function to create a reader and the nodes that will read and push one instance to the instance queue:

```
def read_and_push_instance(filename_queue, instance_queue):
    reader = tf.TextLineReader(skip_header_lines=1)
    key, value = reader.read(filename_queue)
    x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1]])
    features = tf.stack([x1, x2])
    enqueue_instance = instance_queue.enqueue([features, target])
    return enqueue_instance
```

Next we define the queues:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()

instance_queue = tf.RandomShuffleQueue([...])
```

And finally we create the `QueueRunner`, but this time we give it a list of different enqueue operations. Each operation will use a different reader, so the threads will simultaneously read from different files:

```
read_and_enqueue_ops = [
    read_and_push_instance(filename_queue, instance_queue)
    for i in range(5)]
queue_runner = tf.train.QueueRunner(instance_queue, read_and_enqueue_ops)
```

The execution phase is then the same as before: first push the names of the files to read, then create a `Coordinator` and create and start the `QueueRunner` threads. This time all threads will read from different files simultaneously until all files are read entirely, and then the `QueueRunner` will close the instance queue so that other ops pulling from it don't get blocked.

Other convenience functions

TensorFlow also offers a few convenience functions to simplify some common tasks when reading training instances. We will go over just a few (see the API documentation for the full list).

The `string_input_producer()` takes a 1D tensor containing a list of filenames, creates a thread that pushes one filename at a time to the filename queue, and then closes the queue. If you specify a number of epochs, it will cycle through the filenames once per epoch before closing the queue. By default, it shuffles the filenames at each epoch. It creates a `QueueRunner` to manage its thread, and adds it to the `GraphKeys.QUEUE_RUNNERS` collection. To start every `QueueRunner` in that collection, you can call the `tf.train.start_queue_runners()` function. Note that if you forget to start the `QueueRunner`, the filename queue will be open and empty, and your readers will be blocked forever.

There are a few other *producer* functions that similarly create a queue and a corresponding `QueueRunner` for running an enqueue operation (e.g., `input_producer()`, `range_input_producer()`, and `slice_input_producer()`).

The `shuffle_batch()` function takes a list of tensors (e.g., `[features, target]`) and creates:

- A `RandomShuffleQueue`
- A `QueueRunner` to enqueue the tensors to the queue (added to the `GraphKeys.QUEUE_RUNNERS` collection)
- A `dequeue_many` operation to extract a mini-batch from the queue

This makes it easy to manage in a single process a multithreaded input pipeline feeding a queue and a training pipeline reading mini-batches from that queue. Also check out the `batch()`, `batch_join()`, and `shuffle_batch_join()` functions that provide similar functionality.

Okay! You now have all the tools you need to start training and running neural networks efficiently across multiple devices and servers on a TensorFlow cluster. Let's review what you have learned:

- Using multiple GPU devices
- Setting up and starting a TensorFlow cluster
- Distributing computations across multiple devices and servers
- Sharing variables (and other stateful ops such as queues and readers) across sessions using containers
- Coordinating multiple graphs working asynchronously using queues

- Reading inputs efficiently using readers, queue runners, and coordinators

Now let's use all of this to parallelize neural networks!

Parallelizing Neural Networks on a TensorFlow Cluster

In this section, first we will look at how to parallelize several neural networks by simply placing each one on a different device. Then we will look at the much trickier problem of training a single neural network across multiple devices and servers.

One Neural Network per Device

The most trivial way to train and run neural networks on a TensorFlow cluster is to take the exact same code you would use for a single device on a single machine, and specify the master server's address when creating the session. That's it—you're done! Your code will be running on the server's default device. You can change the device that will run your graph simply by putting your code's construction phase within a device block.

By running several client sessions in parallel (in different threads or different processes), connecting them to different servers, and configuring them to use different devices, you can quite easily train or run many neural networks in parallel, across all devices and all machines in your cluster (see Figure 12-11). The speedup is almost linear.[4] Training 100 neural networks across 50 servers with 2 GPUs each will not take much longer than training just 1 neural network on 1 GPU.

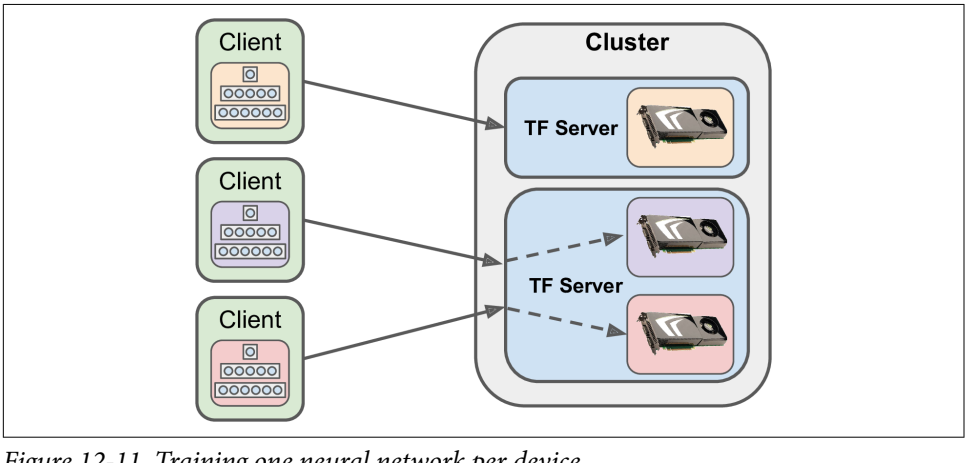

Figure 12-11. Training one neural network per device

4 Not 100% linear if you wait for all devices to finish, since the total time will be the time taken by the slowest device.

This solution is perfect for hyperparameter tuning: each device in the cluster will train a different model with its own set of hyperparameters. The more computing power you have, the larger the hyperparameter space you can explore.

It also works perfectly if you host a web service that receives a large number of *queries per second* (QPS) and you need your neural network to make a prediction for each query. Simply replicate the neural network across all devices on the cluster and dispatch queries across all devices. By adding more servers you can handle an unlimited number of QPS (however, this will not reduce the time it takes to process a single request since it will still have to wait for a neural network to make a prediction).

Another option is to serve your neural networks using *TensorFlow Serving*. It is an open source system, released by Google in February 2016, designed to serve a high volume of queries to Machine Learning models (typically built with TensorFlow). It handles model versioning, so you can easily deploy a new version of your network to production, or experiment with various algorithms without interrupting your service, and it can sustain a heavy load by adding more servers. For more details, check out *https://tensorflow.github.io/serving/*.

In-Graph Versus Between-Graph Replication

You can also parallelize the training of a large ensemble of neural networks by simply placing every neural network on a different device (ensembles were introduced in Chapter 7). However, once you want to *run* the ensemble, you will need to aggregate the individual predictions made by each neural network to produce the ensemble's prediction, and this requires a bit of coordination.

There are two major approaches to handling a neural network ensemble (or any other graph that contains large chunks of independent computations):

- You can create one big graph, containing every neural network, each pinned to a different device, plus the computations needed to aggregate the individual predictions from all the neural networks (see Figure 12-12). Then you just create one session to any server in the cluster and let it take care of everything (including waiting for all individual predictions to be available before aggregating them). This approach is called *in-graph replication*.

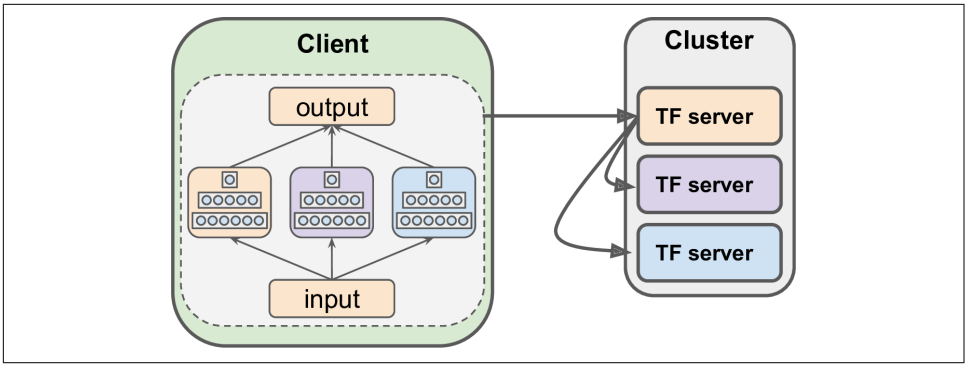

Figure 12-12. In-graph replication

- Alternatively, you can create one separate graph for each neural network and handle synchronization between these graphs yourself. This approach is called *between-graph replication*. One typical implementation is to coordinate the execution of these graphs using queues (see Figure 12-13). A set of clients handles one neural network each, reading from its dedicated input queue, and writing to its dedicated prediction queue. Another client is in charge of reading the inputs and pushing them to all the input queues (copying all inputs to every queue). Finally, one last client is in charge of reading one prediction from each prediction queue and aggregating them to produce the ensemble's prediction.

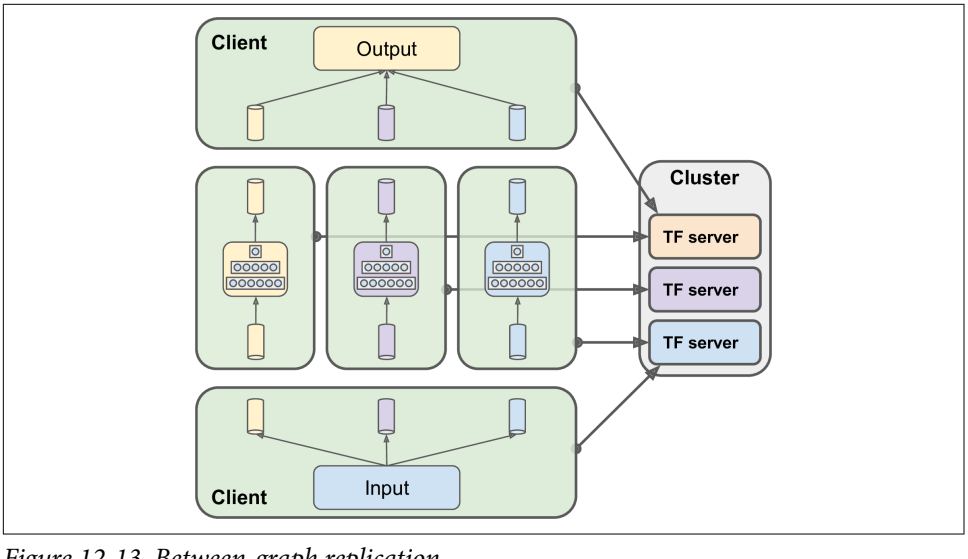

Figure 12-13. Between-graph replication

These solutions have their pros and cons. In-graph replication is somewhat simpler to implement since you don't have to manage multiple clients and multiple queues. However, between-graph replication is a bit easier to organize into well-bounded and easy-to-test modules. Moreover, it gives you more flexibility. For example, you could add a dequeue timeout in the aggregator client so that the ensemble would not fail even if one of the neural network clients crashes or if one neural network takes too long to produce its prediction. TensorFlow lets you specify a timeout when calling the `run()` function by passing a `RunOptions` with `timeout_in_ms`:

```
with tf.Session([...]) as sess:
    [...]
    run_options = tf.RunOptions()
    run_options.timeout_in_ms = 1000  # 1s timeout
    try:
        pred = sess.run(dequeue_prediction, options=run_options)
    except tf.errors.DeadlineExceededError as ex:
        [...] # the dequeue operation timed out after 1s
```

Another way you can specify a timeout is to set the session's `operation_timeout_in_ms` configuration option, but in this case the `run()` function times out if *any* operation takes longer than the timeout delay:

```
config = tf.ConfigProto()
config.operation_timeout_in_ms = 1000  # 1s timeout for every operation

with tf.Session([...], config=config) as sess:
    [...]
    try:
        pred = sess.run(dequeue_prediction)
    except tf.errors.DeadlineExceededError as ex:
        [...]  # the dequeue operation timed out after 1s
```

Model Parallelism

So far we have run each neural network on a single device. What if we want to run a single neural network across multiple devices? This requires chopping your model into separate chunks and running each chunk on a different device. This is called *model parallelism*. Unfortunately, model parallelism turns out to be pretty tricky, and it really depends on the architecture of your neural network. For fully connected networks, there is generally not much to be gained from this approach (see Figure 12-14). Intuitively, it may seem that an easy way to split the model is to place each layer on a different device, but this does not work since each layer needs to wait for the output of the previous layer before it can do anything. So perhaps you can slice it vertically—for example, with the left half of each layer on one device, and the right part on another device? This is slightly better, since both halves of each layer can indeed work in parallel, but the problem is that each half of the next layer requires the output of both halves, so there will be a lot of cross-device communication (repre-

sented by the dashed arrows). This is likely to completely cancel out the benefit of the parallel computation, since cross-device communication is slow (especially if it is across separate machines).

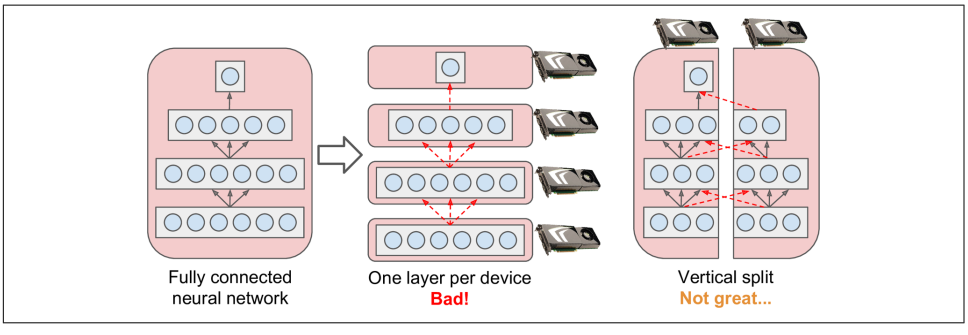

Figure 12-14. Splitting a fully connected neural network

However, as we will see in Chapter 13, some neural network architectures, such as convolutional neural networks, contain layers that are only partially connected to the lower layers, so it is much easier to distribute chunks across devices in an efficient way.

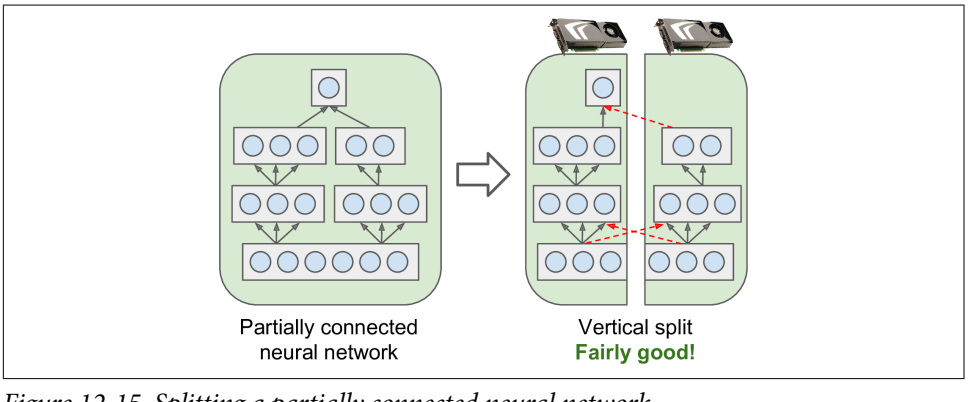

Figure 12-15. Splitting a partially connected neural network

Moreover, as we will see in Chapter 14, some deep recurrent neural networks are composed of several layers of *memory cells* (see the left side of Figure 12-16). A cell's output at time t is fed back to its input at time $t + 1$ (as you can see more clearly on the right side of Figure 12-16). If you split such a network horizontally, placing each layer on a different device, then at the first step only one device will be active, at the second step two will be active, and by the time the signal propagates to the output layer all devices will be active simultaneously. There is still a lot of cross-device communication going on, but since each cell may be fairly complex, the benefit of running multiple cells in parallel often outweighs the communication penalty.

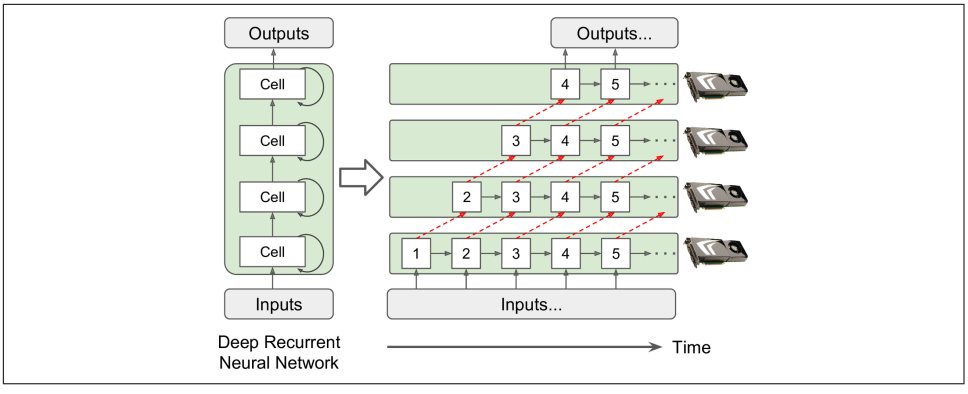

Figure 12-16. Splitting a deep recurrent neural network

In short, model parallelism can speed up running or training some types of neural networks, but not all, and it requires special care and tuning, such as making sure that devices that need to communicate the most run on the same machine.

Data Parallelism

Another way to parallelize the training of a neural network is to replicate it on each device, run a training step simultaneously on all replicas using a different mini-batch for each, and then aggregate the gradients to update the model parameters. This is called *data parallelism* (see Figure 12-17).

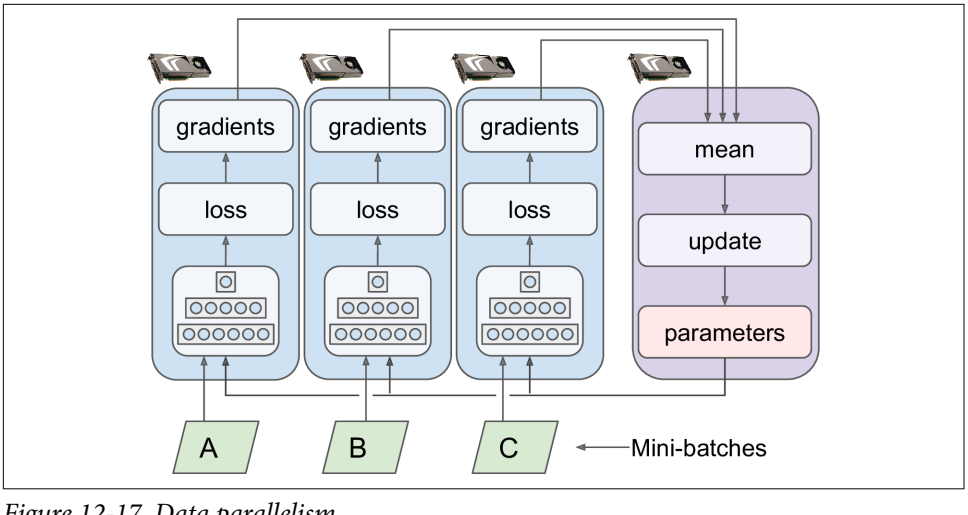

Figure 12-17. Data parallelism

There are two variants of this approach: *synchronous updates* and *asynchronous updates*.

Synchronous updates

With *synchronous updates*, the aggregator waits for all gradients to be available before computing the average and applying the result (i.e., using the aggregated gradients to update the model parameters). Once a replica has finished computing its gradients, it must wait for the parameters to be updated before it can proceed to the next mini-batch. The downside is that some devices may be slower than others, so all other devices will have to wait for them at every step. Moreover, the parameters will be copied to every device almost at the same time (immediately after the gradients are applied), which may saturate the parameter servers' bandwidth.

To reduce the waiting time at each step, you could ignore the gradients from the slowest few replicas (typically ~10%). For example, you could run 20 replicas, but only aggregate the gradients from the fastest 18 replicas at each step, and just ignore the gradients from the last 2. As soon as the parameters are updated, the first 18 replicas can start working again immediately, without having to wait for the 2 slowest replicas. This setup is generally described as having 18 replicas plus 2 *spare replicas*.[5]

Asynchronous updates

With asynchronous updates, whenever a replica has finished computing the gradients, it immediately uses them to update the model parameters. There is no aggregation (remove the "mean" step in Figure 12-17), and no synchronization. Replicas just work independently of the other replicas. Since there is no waiting for the other replicas, this approach runs more training steps per minute. Moreover, although the parameters still need to be copied to every device at every step, this happens at different times for each replica so the risk of bandwidth saturation is reduced.

Data parallelism with asynchronous updates is an attractive choice, because of its simplicity, the absence of synchronization delay, and a better use of the bandwidth. However, although it works reasonably well in practice, it is almost surprising that it works at all! Indeed, by the time a replica has finished computing the gradients based on some parameter values, these parameters will have been updated several times by other replicas (on average $N - 1$ times if there are N replicas) and there is no guarantee that the computed gradients will still be pointing in the right direction (see Figure 12-18). When gradients are severely out-of-date, they are called *stale gradients*: they can slow down convergence, introducing noise and wobble effects (the learning

5 This name is slightly confusing since it sounds like some replicas are special, doing nothing. In reality, all replicas are equivalent: they all work hard to be among the fastest at each training step, and the losers vary at every step (unless some devices are really slower than others).

curve may contain temporary oscillations), or they can even make the training algorithm diverge.

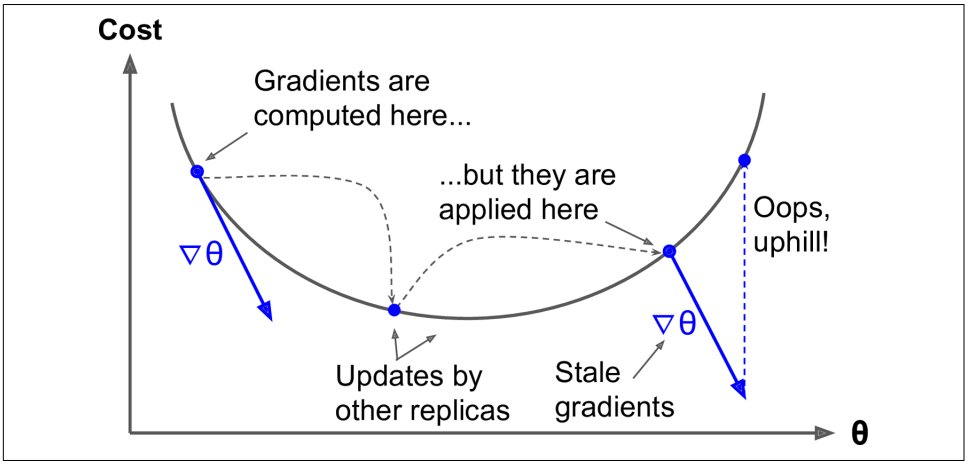

Figure 12-18. Stale gradients when using asynchronous updates

There are a few ways to reduce the effect of stale gradients:

- Reduce the learning rate.
- Drop stale gradients or scale them down.
- Adjust the mini-batch size.
- Start the first few epochs using just one replica (this is called the *warmup phase*). Stale gradients tend to be more damaging at the beginning of training, when gradients are typically large and the parameters have not settled into a valley of the cost function yet, so different replicas may push the parameters in quite different directions.

A paper published by the Google Brain team in April 2016 benchmarked various approaches and found that data parallelism with synchronous updates using a few spare replicas was the most efficient, not only converging faster but also producing a better model. However, this is still an active area of research, so you should not rule out asynchronous updates quite yet.

Bandwidth saturation

Whether you use synchronous or asynchronous updates, data parallelism still requires communicating the model parameters from the parameter servers to every replica at the beginning of every training step, and the gradients in the other direction at the end of each training step. Unfortunately, this means that there always comes a point where adding an extra GPU will not improve performance at all because the

time spent moving the data in and out of GPU RAM (and possibly across the network) will outweigh the speedup obtained by splitting the computation load. At that point, adding more GPUs will just increase saturation and slow down training.

For some models, typically relatively small and trained on a very large training set, you are often better off training the model on a single machine with a single GPU.

Saturation is more severe for large dense models, since they have a lot of parameters and gradients to transfer. It is less severe for small models (but the parallelization gain is small) and also for large sparse models since the gradients are typically mostly zeros, so they can be communicated efficiently. Jeff Dean, initiator and lead of the Google Brain project, reported typical speedups of 25–40x when distributing computations across 50 GPUs for dense models, and 300x speedup for sparser models trained across 500 GPUs. As you can see, sparse models really do scale better. Here are a few concrete examples:

- Neural Machine Translation: 6x speedup on 8 GPUs
- Inception/ImageNet: 32x speedup on 50 GPUs
- RankBrain: 300x speedup on 500 GPUs

These numbers represent the state of the art in Q1 2016. Beyond a few dozen GPUs for a dense model or few hundred GPUs for a sparse model, saturation kicks in and performance degrades. There is plenty of research going on to solve this problem (exploring peer-to-peer architectures rather than centralized parameter servers, using lossy model compression, optimizing when and what the replicas need to communicate, and so on), so there will likely be a lot of progress in parallelizing neural networks in the next few years.

In the meantime, here are a few simple steps you can take to reduce the saturation problem:

- Group your GPUs on a few servers rather than scattering them across many servers. This will avoid unnecessary network hops.
- Shard the parameters across multiple parameter servers (as discussed earlier).
- Drop the model parameters' float precision from 32 bits (`tf.float32`) to 16 bits (`tf.bfloat16`). This will cut in half the amount of data to transfer, without much impact on the convergence rate or the model's performance.

Although 16-bit precision is the minimum for training neural network, you can actually drop down to 8-bit precision after training to reduce the size of the model and speed up computations. This is called *quantizing* the neural network. It is particularly useful for deploying and running pretrained models on mobile phones. See Pete Warden's great post on the subject.

TensorFlow implementation

To implement data parallelism using TensorFlow, you first need to choose whether you want in-graph replication or between-graph replication, and whether you want synchronous updates or asynchronous updates. Let's look at how you would implement each combination (see the exercises and the Jupyter notebooks for complete code examples).

With in-graph replication + synchronous updates, you build one big graph containing all the model replicas (placed on different devices), and a few nodes to aggregate all their gradients and feed them to an optimizer. Your code opens a session to the cluster and simply runs the training operation repeatedly.

With in-graph replication + asynchronous updates, you also create one big graph, but with one optimizer per replica, and you run one thread per replica, repeatedly running the replica's optimizer.

With between-graph replication + asynchronous updates, you run multiple independent clients (typically in separate processes), each training the model replica as if it were alone in the world, but the parameters are actually shared with other replicas (using a resource container).

With between-graph replication + synchronous updates, once again you run multiple clients, each training a model replica based on shared parameters, but this time you wrap the optimizer (e.g., a `MomentumOptimizer`) within a `SyncReplicasOptimizer`. Each replica uses this optimizer as it would use any other optimizer, but under the hood this optimizer sends the gradients to a set of queues (one per variable), which is read by one of the replica's `SyncReplicasOptimizer`, called the *chief*. The chief aggregates the gradients and applies them, then writes a token to a *token queue* for each replica, signaling it that it can go ahead and compute the next gradients. This approach supports having *spare replicas*.

If you go through the exercises, you will implement each of these four solutions. You will easily be able to apply what you have learned to train large deep neural networks across dozens of servers and GPUs! In the following chapters we will go through a few more important neural network architectures before we tackle Reinforcement Learning.

Exercises

1. If you get a `CUDA_ERROR_OUT_OF_MEMORY` when starting your TensorFlow program, what is probably going on? What can you do about it?
2. What is the difference between pinning an operation on a device and placing an operation on a device?
3. If you are running on a GPU-enabled TensorFlow installation, and you just use the default placement, will all operations be placed on the first GPU?
4. If you pin a variable to `"/gpu:0"`, can it be used by operations placed on `/gpu:1`? Or by operations placed on `"/cpu:0"`? Or by operations pinned to devices located on other servers?
5. Can two operations placed on the same device run in parallel?
6. What is a control dependency and when would you want to use one?
7. Suppose you train a DNN for days on a TensorFlow cluster, and immediately after your training program ends you realize that you forgot to save the model using a `Saver`. Is your trained model lost?
8. Train several DNNs in parallel on a TensorFlow cluster, using different hyperparameter values. This could be DNNs for MNIST classification or any other task you are interested in. The simplest option is to write a single client program that trains only one DNN, then run this program in multiple processes in parallel, with different hyperparameter values for each client. The program should have command-line options to control what server and device the DNN should be placed on, and what resource container and hyperparameter values to use (make sure to use a different resource container for each DNN). Use a validation set or cross-validation to select the top three models.
9. Create an ensemble using the top three models from the previous exercise. Define it in a single graph, ensuring that each DNN runs on a different device. Evaluate it on the validation set: does the ensemble perform better than the individual DNNs?
10. Train a DNN using between-graph replication and data parallelism with asynchronous updates, timing how long it takes to reach a satisfying performance. Next, try again using synchronous updates. Do synchronous updates produce a better model? Is training faster? Split the DNN vertically and place each vertical slice on a different device, and train the model again. Is training any faster? Is the performance any different?

Solutions to these exercises are available in Appendix A.

CHAPTER 13
Convolutional Neural Networks

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

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

In this chapter we will present where CNNs came from, what their building blocks look like, and how to implement them using TensorFlow. Then we will present some of the best CNN architectures.

The Architecture of the Visual Cortex

David H. Hubel and Torsten Wiesel performed a series of experiments on cats in 1958[1] and 1959[2] (and a few years later on monkeys[3]), giving crucial insights on the structure of the visual cortex (the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work). In particular, they showed that many neurons in the visual cortex have a small *local receptive field*, meaning they react only to visual stimuli located in a limited region of the visual field (see Figure 13-1, in which the local receptive fields of five neurons are represented by dashed circles). The receptive fields of different neurons may overlap, and together they tile the whole visual field. Moreover, the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). They also noticed that some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons (in Figure 13-1, notice that each neuron is connected only to a few neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

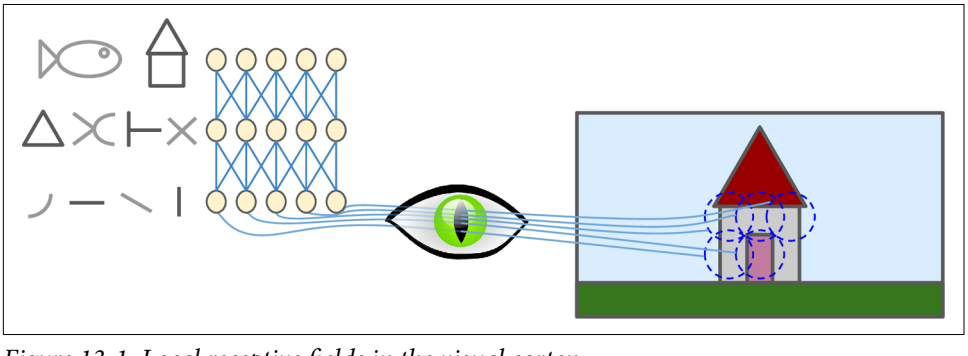

Figure 13-1. Local receptive fields in the visual cortex

These studies of the visual cortex inspired the neocognitron, introduced in 1980,[4] which gradually evolved into what we now call *convolutional neural networks*. An important milestone was a 1998 paper[5] by Yann LeCun, Léon Bottou, Yoshua Bengio,

1 "Single Unit Activity in Striate Cortex of Unrestrained Cats," D. Hubel and T. Wiesel (1958).

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

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

4 "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position," K. Fukushima (1980).

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

and Patrick Haffner, which introduced the famous *LeNet-5* architecture, widely used to recognize handwritten check numbers. This architecture has some building blocks that you already know, such as fully connected layers and sigmoid activation functions, but it also introduces two new building blocks: *convolutional layers* and *pooling layers*. Let's look at them now.

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

Convolutional Layer

The most important building block of a CNN is the *convolutional layer*:[6] neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in previous chapters), but only to pixels in their receptive fields (see Figure 13-2). In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on low-level features in the first hidden layer, then assemble them into higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

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

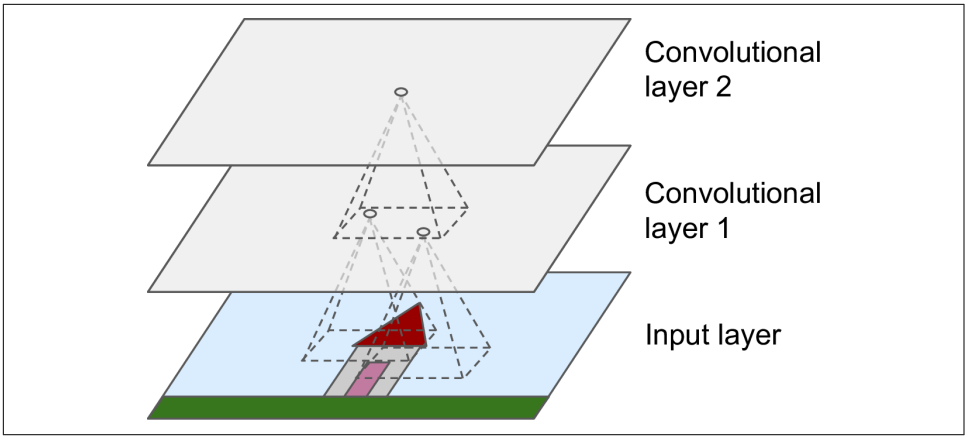

Figure 13-2. CNN layers with rectangular local receptive fields

Until now, all multilayer neural networks we looked at had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network. Now each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field (see Figure 13-3). In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram. This is called *zero padding*.

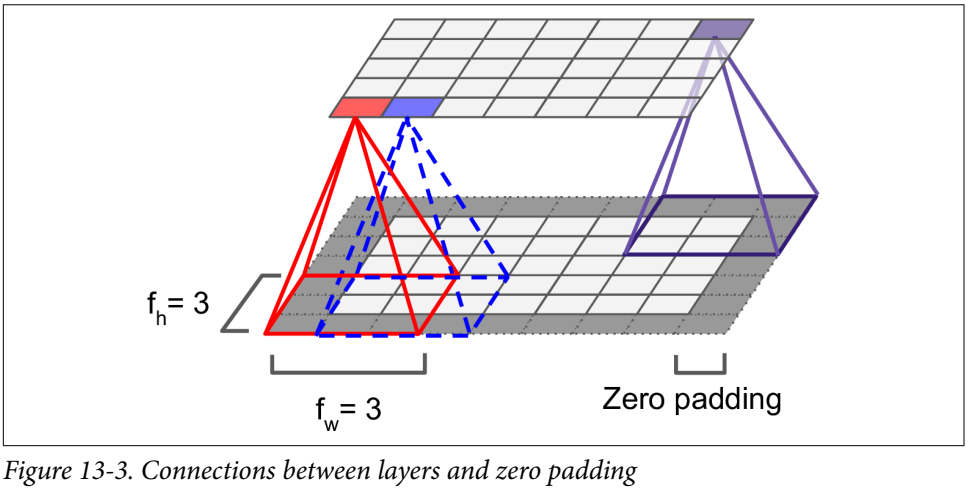

Figure 13-3. Connections between layers and zero padding

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in Figure 13-4. The distance between two consecutive receptive fields is called the *stride*. In the diagram, a 5 × 7 input layer (plus zero padding) is connected to a 3 × 4 layer, using 3 × 3 receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). A neuron located in row i, column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

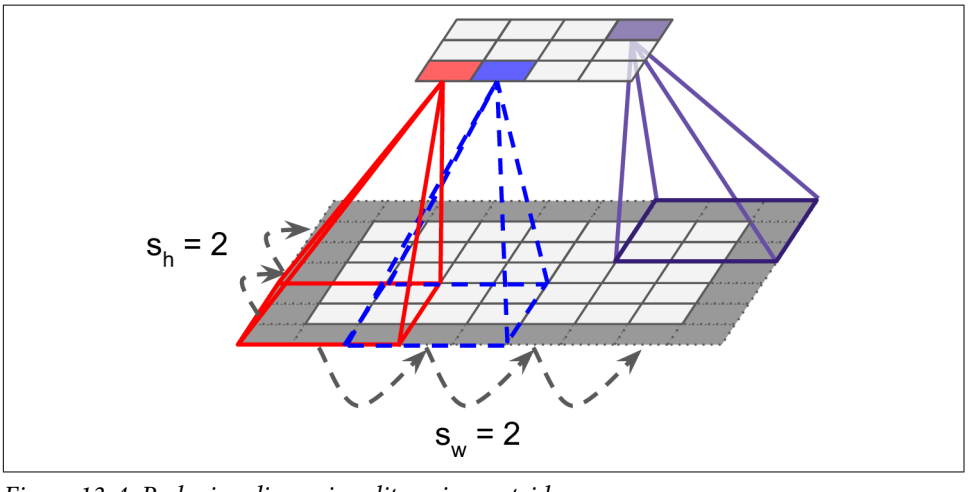

Figure 13-4. Reducing dimensionality using a stride

Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, Figure 13-5 shows two possible sets of weights, called *filters* (or *convolution kernels*). The first one is represented as a black square with a vertical white line in the middle (it is a 7 × 7 matrix full of 0s except for the central column, which is full of 1s); neurons using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will get multiplied by 0, except for the ones located in the central vertical line). The second filter is a black square with a horizontal white line in the middle. Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

Now if all neurons in a layer use the same vertical line filter (and the same bias term), and you feed the network the input image shown in Figure 13-5 (bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what you get if all neurons use the horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out. Thus, a layer full of neurons using the same filter gives

you a *feature map*, which highlights the areas in an image that are most similar to the filter. During training, a CNN finds the most useful filters for its task, and it learns to combine them into more complex patterns (e.g., a cross is an area in an image where both the vertical filter and the horizontal filter are active).

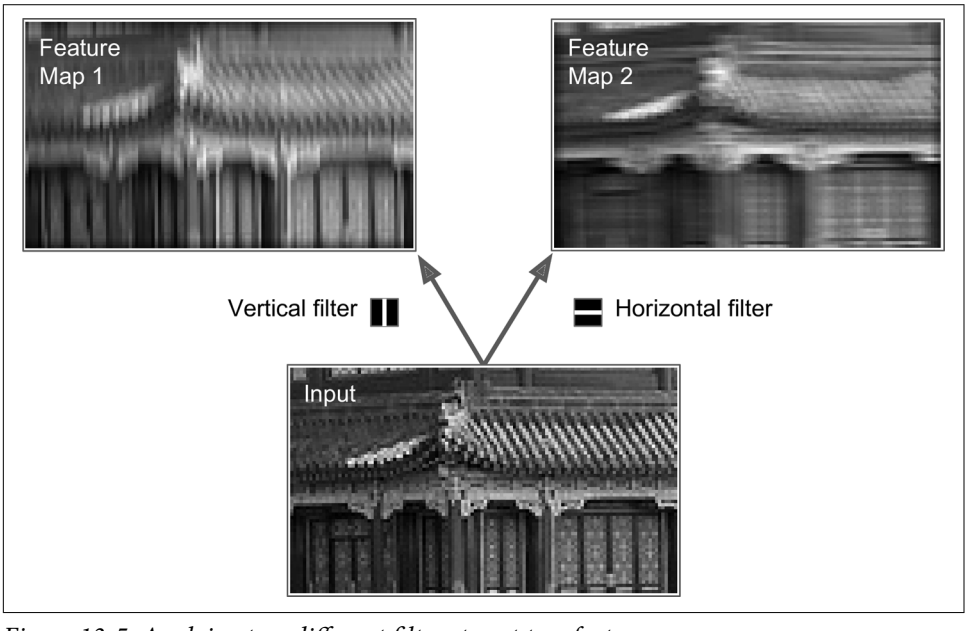

Figure 13-5. Applying two different filters to get two feature maps

Stacking Multiple Feature Maps

Up to now, for simplicity, we have represented each convolutional layer as a thin 2D layer, but in reality it is composed of several feature maps of equal sizes, so it is more accurately represented in 3D (see Figure 13-6). Within one feature map, all neurons share the same parameters (weights and bias term), but different feature maps may have different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the previous layers' feature maps. In short, a convolutional layer simultaneously applies multiple filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

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

Moreover, input images are also composed of multiple sublayers: one per *color channel*. There are typically three: red, green, and blue (RGB). Grayscale images have just one channel, but some images may have much more—for example, satellite images that capture extra light frequencies (such as infrared).

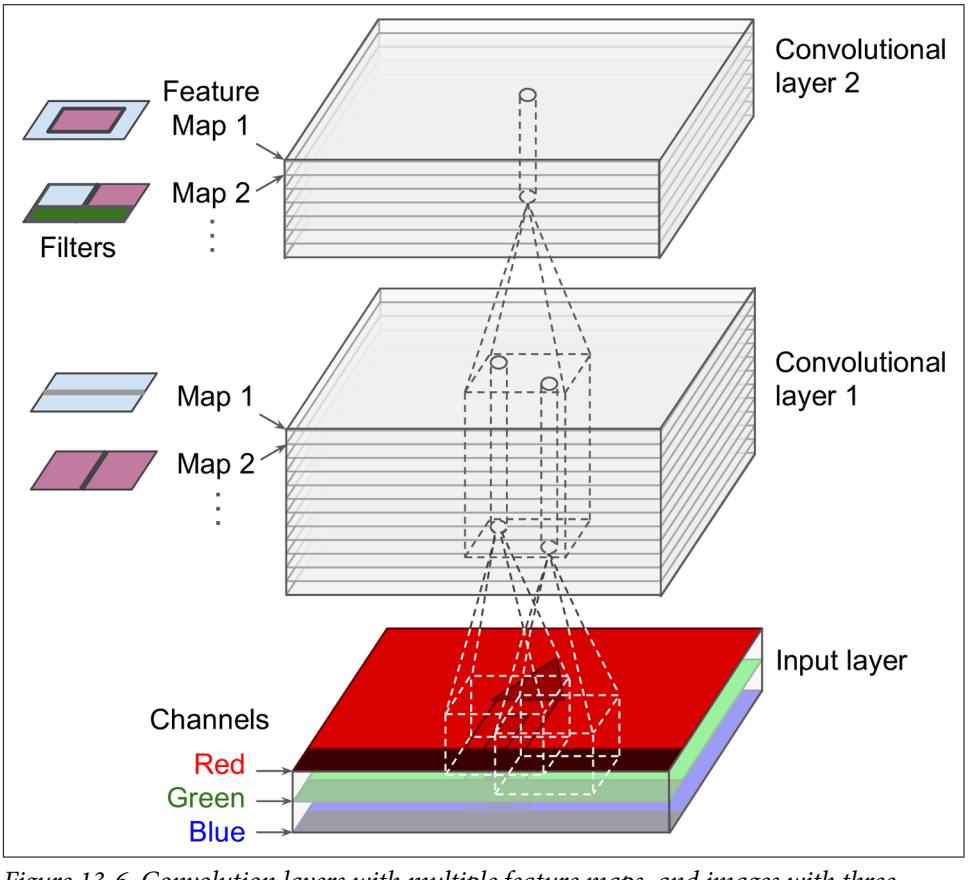

Figure 13-6. Convolution layers with multiple feature maps, and images with three channels

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

Equation 13-1 summarizes the preceding explanations in one big mathematical equation: it shows how to compute the output of a given neuron in a convolutional layer.

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

Equation 13-1. Computing the output of a neuron in a convolutional layer

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

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

TensorFlow Implementation

In TensorFlow, each input image is typically represented as a 3D tensor of `shape [height, width, channels]`. A mini-batch is represented as a 4D tensor of `shape [mini-batch size, height, width, channels]`. The weights of a convolutional layer are represented as a 4D tensor of shape $[f_h, f_w, f_n, f_{n'}]$. The bias terms of a convolutional layer are simply represented as a 1D tensor of `shape [`f_n`]`.

Let's look at a simple example. The following code loads two sample images, using Scikit-Learn's `load_sample_images()` (which loads two color images, one of a Chinese temple, and the other of a flower). Then it creates two 7×7 filters (one with a vertical white line in the middle, and the other with a horizontal white line), and applies them to both images using a convolutional layer built using TensorFlow's `conv2d()` function (with zero padding and a stride of 2). Finally, it plots one of the resulting feature maps (similar to the top-right image in Figure 13-5).

```
import numpy as np
from sklearn.datasets import load_sample_images

# Load sample images
dataset = np.array(load_sample_images().images, dtype=np.float32)
batch_size, height, width, channels = dataset.shape

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

# Create a graph with input X plus a convolutional layer applying the 2 filters
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1], padding="SAME")

with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={X: dataset})

plt.imshow(output[0, :, :, 1])  # plot 1st image's 2nd feature map
plt.show()
```

Most of this code is self-explanatory, but the `conv2d()` line deserves a bit of explanation:

- `X` is the input mini-batch (a 4D tensor, as explained earlier).
- `filters` is the set of filters to apply (also a 4D tensor, as explained earlier).
- `strides` is a four-element 1D array, where the two central elements are the vertical and horizontal strides (s_h and s_w). The first and last elements must currently be equal to 1. They may one day be used to specify a batch stride (to skip some instances) and a channel stride (to skip some of the previous layer's feature maps or channels).
- `padding` must be either `"VALID"` or `"SAME"`:
  - If set to `"VALID"`, the convolutional layer does *not* use zero padding, and may ignore some rows and columns at the bottom and right of the input image, depending on the stride, as shown in Figure 13-7 (for simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension).
  - If set to `"SAME"`, the convolutional layer uses zero padding if necessary. In this case, the number of output neurons is equal to the number of input neurons divided by the stride, rounded up (in this example, ceil (13 / 5) = 3). Then zeros are added as evenly as possible around the inputs.

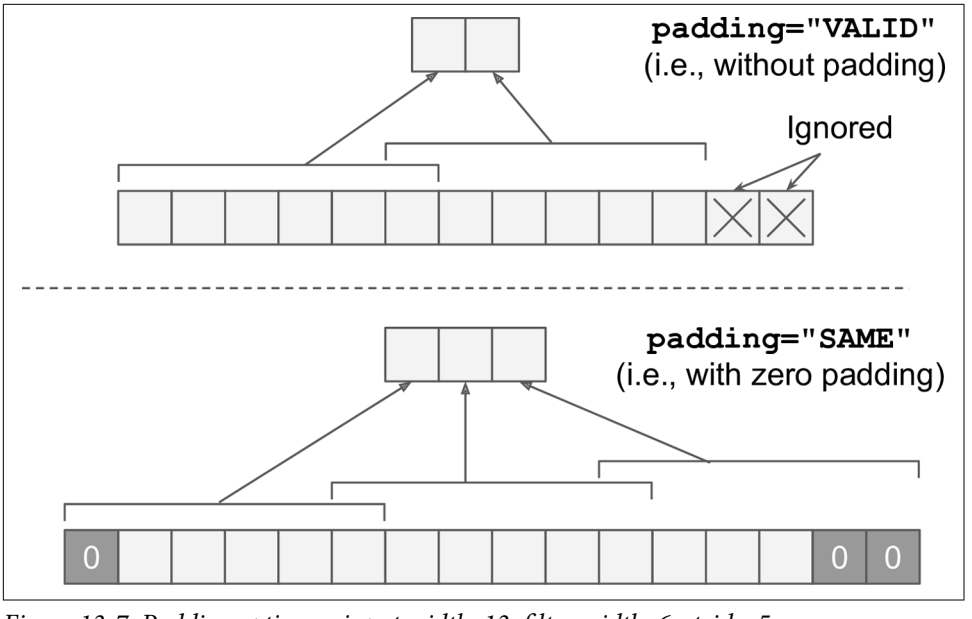

Figure 13-7. Padding options—input width: 13, filter width: 6, stride: 5

Unfortunately, convolutional layers have quite a few hyperparameters: you must choose the number of filters, their height and width, the strides, and the padding type. As always, you can use cross-validation to find the right hyperparameter values, but this is very time-consuming. We will discuss common CNN architectures later, to give you some idea of what hyperparameter values work best in practice.

Memory Requirements

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

For example, consider a convolutional layer with 5×5 filters, outputting 200 feature maps of size 150×100, with stride 1 and SAME padding. If the input is a 150×100 RGB image (three channels), then the number of parameters is $(5 \times 5 \times 3 + 1) \times 200 = 15{,}200$ (the +1 corresponds to the bias terms), which is fairly small compared to a fully connected layer.[7] However, each of the 200 feature maps contains 150×100 neurons, and each of these neurons needs to compute a weighted sum of its $5 \times 5 \times 3 = 75$ inputs: that's a total of 225 million float multiplications. Not as bad as a fully con-

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

nected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy $200 \times 150 \times 100 \times 32 = 96$ million bits (about 11.4 MB) of RAM.[8] And that's just for one instance! If a training batch contains 100 instances, then this layer will use up over 1 GB of RAM!

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

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

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

Pooling Layer

Once you understand how convolutional layers work, the pooling layers are quite easy to grasp. Their goal is to *subsample* (i.e., shrink) the input image in order to reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting). Reducing the input image size also makes the neural network tolerate a little bit of image shift (*location invariance*).

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean. Figure 13-8 shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2×2 *pooling kernel*, a stride of 2, and no padding. Note that only the max input value in each kernel makes it to the next layer. The other inputs are dropped.

8 1 MB = 1,024 kB = 1,024 × 1,024 bytes = 1,024 × 1,024 × 8 bits.

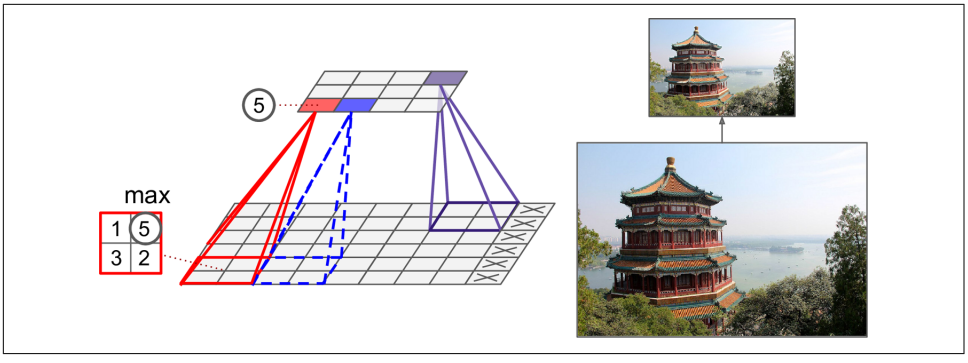

Figure 13-8. Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

This is obviously a very destructive kind of layer: even with a tiny 2 × 2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75% of the input values.

A pooling layer typically works on every input channel independently, so the output depth is the same as the input depth. You may alternatively pool over the depth dimension, as we will see next, in which case the image's spatial dimensions (height and width) remain unchanged, but the number of channels is reduced.

Implementing a max pooling layer in TensorFlow is quite easy. The following code creates a max pooling layer using a 2 × 2 kernel, stride 2, and no padding, then applies it to all the images in the dataset:

```
[...] # load the image dataset, just like above

# Create a graph with input X plus a max pooling layer
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8))  # plot the output for the 1st image
plt.show()
```

The `ksize` argument contains the kernel shape along all four dimensions of the input tensor: `[batch size, height, width, channels]`. TensorFlow currently does not support pooling over multiple instances, so the first element of `ksize` must be equal to 1. Moreover, it does not support pooling over both the spatial dimensions (height and width) and the depth dimension, so either `ksize[1]` and `ksize[2]` must both be equal to 1, or `ksize[3]` must be equal to 1.

To create an *average pooling layer*, just use the `avg_pool()` function instead of `max_pool()`.

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

CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps) thanks to the convolutional layers (see Figure 13-9). At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

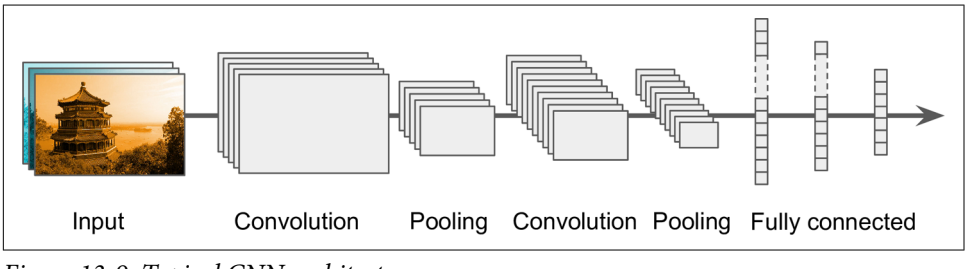

Figure 13-9. Typical CNN architecture

A common mistake is to use convolution kernels that are too large. You can often get the same effect as a 9 × 9 kernel by stacking two 3 × 3 kernels on top of each other, for a lot less compute.

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the ILSVRC ImageNet challenge. In this competition the top-5 error rate for image classification fell from over 26% to barely over 3% in just five years. The top-five error rate is the number of test images for which the system's top 5 predictions did not include the correct answer. The images are large (256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds). Looking at the evolution of the winning entries is a good way to understand how CNNs work.

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

Other Visual Tasks

There was stunning progress as well in other visual tasks such as object detection and localization, and image segmentation. In object detection and localization, the neural network typically outputs a sequence of bounding boxes around various objects in the image. For example, see Maxine Oquab et al.'s 2015 paper that outputs a heat map for each object class, or Russell Stewart et al.'s 2015 paper that uses a combination of a CNN to detect faces and a recurrent neural network to output a sequence of bounding boxes around them. In image segmentation, the net outputs an image (usually of the same size as the input) where each pixel indicates the class of the object to which the corresponding input pixel belongs. For example, check out Evan Shelhamer et al.'s 2016 paper.

LeNet-5

The LeNet-5 architecture is perhaps the most widely known CNN architecture. As mentioned earlier, it was created by Yann LeCun in 1998 and widely used for handwritten digit recognition (MNIST). It is composed of the layers shown in Table 13-1.

Table 13-1. LeNet-5 architecture

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

There are a few extra details to be noted:

- MNIST images are 28×28 pixels, but they are zero-padded to 32×32 pixels and normalized before being fed to the network. The rest of the network does not use any padding, which is why the size keeps shrinking as the image progresses through the network.
- The average pooling layers are slightly more complex than usual: each neuron computes the mean of its inputs, then multiplies the result by a learnable coefficient (one per map) and adds a learnable bias term (again, one per map), then finally applies the activation function.

- Most neurons in C3 maps are connected to neurons in only three or four S2 maps (instead of all six S2 maps). See table 1 in the original paper for details.
- The output layer is a bit special: instead of computing the dot product of the inputs and the weight vector, each neuron outputs the square of the Euclidian distance between its input vector and its weight vector. Each output measures how much the image belongs to a particular digit class. The cross entropy cost function is now preferred, as it penalizes bad predictions much more, producing larger gradients and thus converging faster.

Yann LeCun's website ("LENET" section) features great demos of LeNet-5 classifying digits.

AlexNet

The *AlexNet* CNN architecture[9] won the 2012 ImageNet ILSVRC challenge by a large margin: it achieved 17% top-5 error rate while the second best achieved only 26%! It was developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton. It is quite similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of each other, instead of stacking a pooling layer on top of each convolutional layer. Table 13-2 presents this architecture.

Table 13-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13×13	3×3	1	SAME	ReLU
C6	Convolution	384	13×13	3×3	1	SAME	ReLU
C5	Convolution	384	13×13	3×3	1	SAME	ReLU
S4	Max Pooling	256	13×13	3×3	2	VALID	–
C3	Convolution	256	27×27	5×5	1	SAME	ReLU
S2	Max Pooling	96	27×27	3×3	2	VALID	–
C1	Convolution	96	55×55	11×11	4	SAME	ReLU
In	Input	3 (RGB)	224×224	–	–	–	–

To reduce overfitting, the authors used two regularization techniques we discussed in previous chapters: first they applied dropout (with a 50% dropout rate) during training to the outputs of layers F8 and F9. Second, they performed data augmentation by

9 "ImageNet Classification with Deep Convolutional Neural Networks," A. Krizhevsky et al. (2012).

randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, called *local response normalization*. This form of normalization makes the neurons that most strongly activate inhibit neurons at the same location but in neighboring feature maps (such competitive activation has been observed in biological neurons). This encourages different feature maps to specialize, pushing them apart and forcing them to explore a wider range of features, ultimately improving generalization. Equation 13-2 shows how to apply LRN.

Equation 13-2. Local response normalization

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

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

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

In AlexNet, the hyperparameters are set as follows: $r = 2$, $\alpha = 0.00002$, $\beta = 0.75$, and $k = 1$. This step can be implemented using TensorFlow's `local_response_normalization()` operation.

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

GoogLeNet

The GoogLeNet architecture was developed by Christian Szegedy et al. from Google Research,[10] and it won the ILSVRC 2014 challenge by pushing the top-5 error rate

10 "Going Deeper with Convolutions," C. Szegedy et al. (2015).

below 7%. This great performance came in large part from the fact that the network was much deeper than previous CNNs (see Figure 13-11). This was made possible by sub-networks called *inception modules*,[11] which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

Figure 13-10 shows the architecture of an inception module. The notation "3 × 3 + 2(S)" means that the layer uses a 3 × 3 kernel, stride 2, and SAME padding. The input signal is first copied and fed to four different layers. All convolutional layers use the ReLU activation function. Note that the second set of convolutional layers uses different kernel sizes (1 × 1, 3 × 3, and 5 × 5), allowing them to capture patterns at different scales. Also note that every single layer uses a stride of 1 and SAME padding (even the max pooling layer), so their outputs all have the same height and width as their inputs. This makes it possible to concatenate all the outputs along the depth dimension in the final *depth concat layer* (i.e., stack the feature maps from all four top convolutional layers). This concatenation layer can be implemented in TensorFlow using the `concat()` operation, with `axis=3` (axis 3 is the depth).

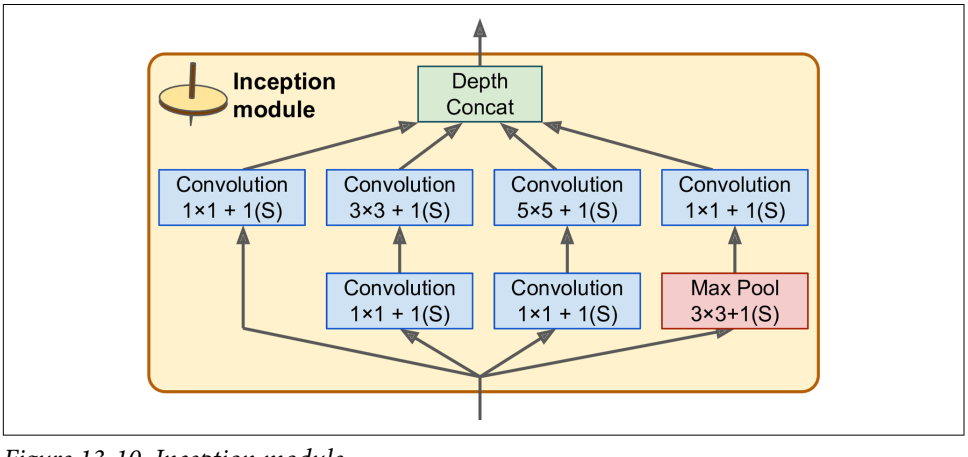

Figure 13-10. Inception module

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

- First, they are configured to output many fewer feature maps than their inputs, so they serve as *bottleneck layers*, meaning they reduce dimensionality. This is par-

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

ticularly useful before the 3 × 3 and 5 × 5 convolutions, since these are very computationally expensive layers.

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

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

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

Now let's look at the architecture of the GoogLeNet CNN (see Figure 13-11). It is so deep that we had to represent it in three columns, but GoogLeNet is actually one tall stack, including nine inception modules (the boxes with the spinning tops) that actually contain three layers each. The number of feature maps output by each convolutional layer and each pooling layer is shown before the kernel size. The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module (in the same order as in Figure 13-10). Note that all the convolutional layers use the ReLU activation function.

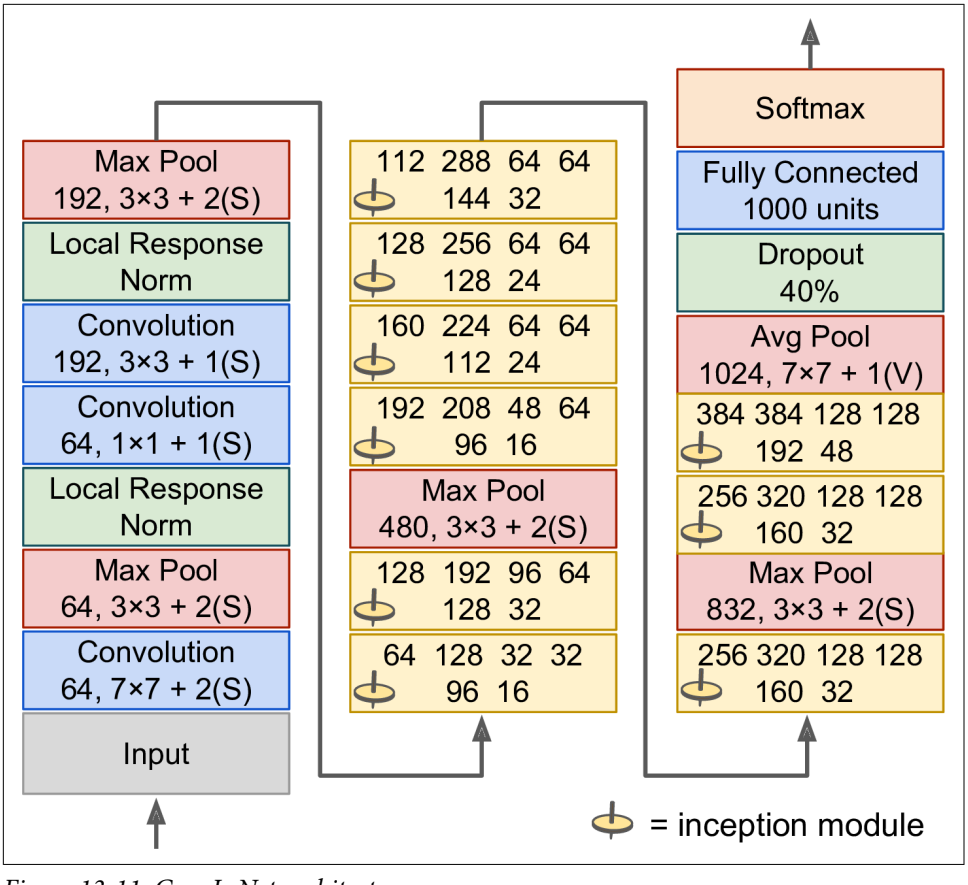

Figure 13-11. GoogLeNet architecture

Let's go through this network:

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features (as discussed earlier).
- Two convolutional layers follow, where the first acts like a *bottleneck layer*. As explained earlier, you can think of this pair as a single smarter convolutional layer.
- Again, a local response normalization layer ensures that the previous layers capture a wide variety of patterns.
- Next a max pooling layer reduces the image height and width by 2, again to speed up computations.

- Then comes the tall stack of nine inception modules, interleaved with a couple max pooling layers to reduce dimensionality and speed up the net.
- Next, the average pooling layer uses a kernel the size of the feature maps with VALID padding, outputting 1×1 feature maps: this surprising strategy is called *global average pooling*. It effectively forces the previous layers to produce feature maps that are actually confidence maps for each target class (since other kinds of features would be destroyed by the averaging step). This makes it unnecessary to have several fully connected layers at the top of the CNN (like in AlexNet), considerably reducing the number of parameters in the network and limiting the risk of overfitting.
- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with a softmax activation function to output estimated class probabilities.

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

ResNet

Last but not least, the winner of the ILSVRC 2015 challenge was the *Residual Network* (or *ResNet*), developed by Kaiming He et al.,[12] which delivered an astounding top-5 error rate under 3.6%, using an extremely deep CNN composed of 152 layers. The key to being able to train such a deep network is to use *skip connections* (also called *shortcut connections*): the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack. Let's see why this is useful.

When training a neural network, the goal is to make it model a target function $h(\mathbf{x})$. If you add the input $\mathbf{x}$ to the output of the network (i.e., you add a skip connection), then the network will be forced to model $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ rather than $h(\mathbf{x})$. This is called *residual learning* (see Figure 13-12).

12 "Deep Residual Learning for Image Recognition," K. He (2015).

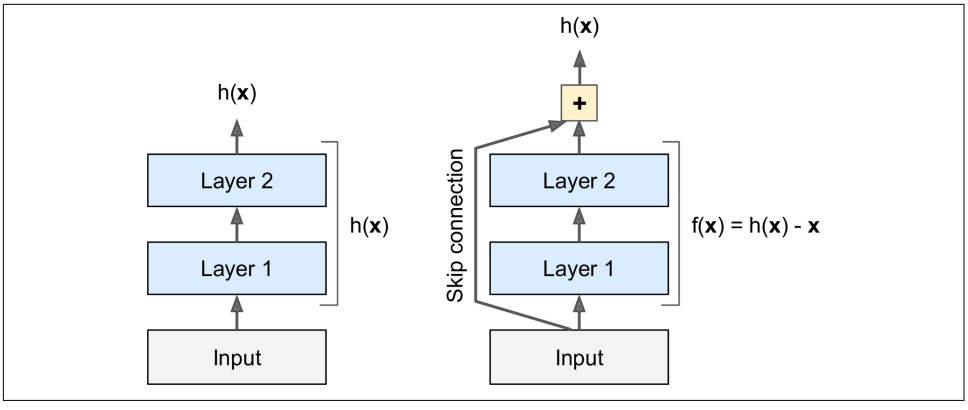

Figure 13-12. Residual learning

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

Moreover, if you add many skip connections, the network can start making progress even if several layers have not started learning yet (see Figure 13-13). Thanks to skip connections, the signal can easily make its way across the whole network. The deep residual network can be seen as a stack of *residual units*, where each residual unit is a small neural network with a skip connection.

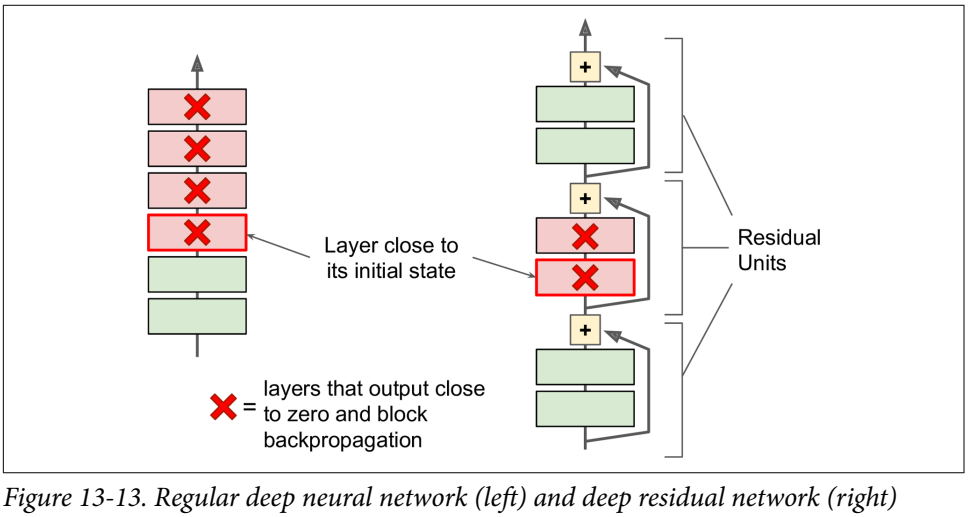

Figure 13-13. Regular deep neural network (left) and deep residual network (right)

Now let's look at ResNet's architecture (see Figure 13-14). It is actually surprisingly simple. It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of simple residual units. Each residual unit is composed of two convolutional layers, with Batch Normalization (BN) and ReLU activation, using 3 × 3 kernels and preserving spatial dimensions (stride 1, SAME padding).

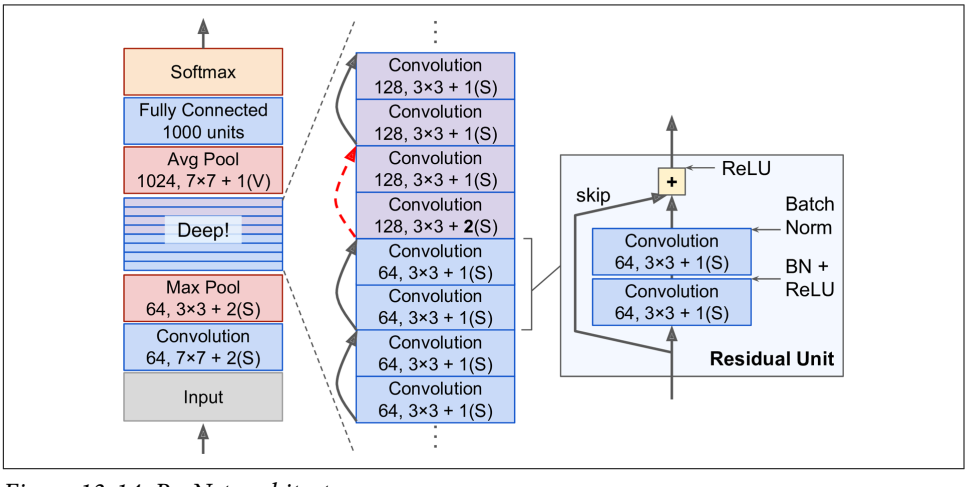

Figure 13-14. ResNet architecture

Note that the number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2). When this happens the inputs cannot be added directly to the outputs of the residual unit since they don't have the same shape (for example, this problem affects the skip connection represented by the dashed arrow in Figure 13-14). To solve this problem, the inputs are passed through a 1 × 1 convolutional layer with stride 2 and the right number of output feature maps (see Figure 13-15).

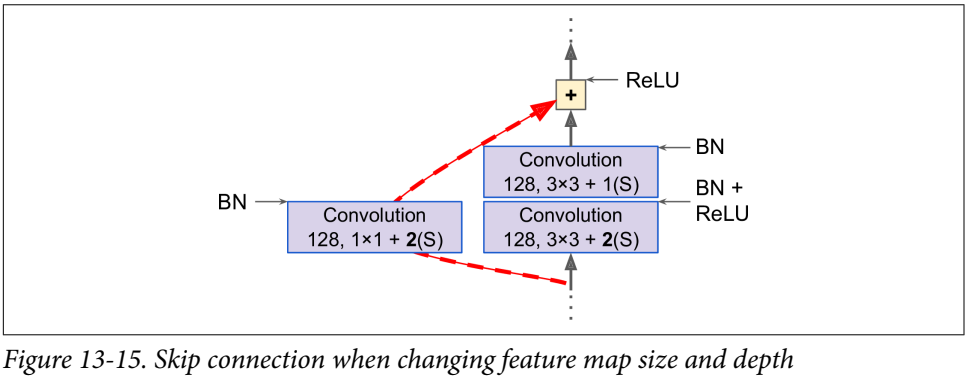

Figure 13-15. Skip connection when changing feature map size and depth

ResNet-34 is the ResNet with 34 layers (only counting the convolutional layers and the fully connected layer) containing three residual units that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps.

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

As you can see, the field is moving rapidly, with all sorts of architectures popping out every year. One clear trend is that CNNs keep getting deeper and deeper. They are also getting lighter, requiring fewer and fewer parameters. At present, the ResNet architecture is both the most powerful and arguably the simplest, so it is really the one you should probably use for now, but keep looking at the ILSVRC challenge every year. The 2016 winners were the Trimps-Soushen team from China with an astounding 2.99% error rate. To achieve this they trained combinations of the previous models and joined them into an ensemble. Depending on the task, the reduced error rate may or may not be worth the extra complexity.

There are a few other architectures that you may want to look at, in particular VGGNet[13] (runner-up of the ILSVRC 2014 challenge) and Inception-v4[14] (which merges the ideas of GoogLeNet and ResNet and achieves close to 3% top-5 error rate on ImageNet classification).

There is really nothing special about implementing the various CNN architectures we just discussed. We saw earlier how to build all the individual building blocks, so now all you need is to assemble them to create the desired architecture. We will build ResNet-34 in the upcoming exercises and you will find full working code in the Jupyter notebooks.

13 "Very Deep Convolutional Networks for Large-Scale Image Recognition," K. Simonyan and A. Zisserman (2015).

14 "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," C. Szegedy et al. (2016).

TensorFlow Convolution Operations

TensorFlow also offers a few other kinds of convolutional layers:

- `conv1d()` creates a convolutional layer for 1D inputs. This is useful, for example, in natural language processing, where a sentence may be represented as a 1D array of words, and the receptive field covers a few neighboring words.
- `conv3d()` creates a convolutional layer for 3D inputs, such as 3D PET scan.
- `atrous_conv2d()` creates an *atrous convolutional layer* ("à trous" is French for "with holes"). This is equivalent to using a regular convolutional layer with a filter dilated by inserting rows and columns of zeros (i.e., holes). For example, a 1×3 filter equal to `[[1,2,3]]` may be dilated with a *dilation rate* of 4, resulting in a *dilated filter* `[[1, 0, 0, 0, 2, 0, 0, 0, 3]]`. This allows the convolutional layer to have a larger receptive field at no computational price and using no extra parameters.
- `conv2d_transpose()` creates a *transpose convolutional layer*, sometimes called a *deconvolutional layer*,[15] which *upsamples* an image. It does so by inserting zeros between the inputs, so you can think of this as a regular convolutional layer using a fractional stride. Upsampling is useful, for example, in image segmentation: in a typical CNN, feature maps get smaller and smaller as you progress through the network, so if you want to output an image of the same size as the input, you need an upsampling layer.
- `depthwise_conv2d()` creates a *depthwise convolutional layer* that applies every filter to every individual input channel independently. Thus, if there are f_n filters and $f_{n'}$ input channels, then this will output $f_n \times f_{n'}$ feature maps.
- `separable_conv2d()` creates a *separable convolutional layer* that first acts like a depthwise convolutional layer, then applies a 1×1 convolutional layer to the resulting feature maps. This makes it possible to apply filters to arbitrary sets of inputs channels.

Exercises

1. What are the advantages of a CNN over a fully connected DNN for image classification?
2. Consider a CNN composed of three convolutional layers, each with 3×3 kernels, a stride of 2, and SAME padding. The lowest layer outputs 100 feature maps, the

15 This name is quite misleading since this layer does *not* perform a deconvolution, which is a well-defined mathematical operation (the inverse of a convolution).

middle one outputs 200, and the top one outputs 400. The input images are RGB images of 200 × 300 pixels. What is the total number of parameters in the CNN? If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance? What about when training on a mini-batch of 50 images?

3. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?
4. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?
5. When would you want to add a *local response normalization* layer?
6. Can you name the main innovations in AlexNet, compared to LeNet-5? What about the main innovations in GoogLeNet and ResNet?
7. Build your own CNN and try to achieve the highest possible accuracy on MNIST.
8. Classifying large images using Inception v3.
   a. Download some images of various animals. Load them in Python, for example using the `matplotlib.image.mpimg.imread()` function. Resize and/or crop them to 299 × 299 pixels, and ensure that they have just three channels (RGB), with no transparency channel.
   b. Download the latest pretrained Inception v3 model: the checkpoint is available at *https://goo.gl/nxSQvl*.
   c. Create the Inception v3 model by calling the `inception_v3()` function, as shown below. This must be done within an argument scope created by the `inception_v3_arg_scope()` function. Also, you must set `is_training=False` and `num_classes=1001` like so:

```
from tensorflow.contrib.slim.nets import inception
import tensorflow.contrib.slim as slim

X = tf.placeholder(tf.float32, shape=[None, 299, 299, 3])
with slim.arg_scope(inception.inception_v3_arg_scope()):
    logits, end_points = inception.inception_v3(
                             X, num_classes=1001, is_training=False)
predictions = end_points["Predictions"]
saver = tf.train.Saver()
```

   d. Open a session and use the `Saver` to restore the pretrained model checkpoint you downloaded earlier.
   e. Run the model to classify the images you prepared. Display the top five predictions for each image, along with the estimated probability (the list of class names is available at *https://goo.gl/brXRtZ*). How accurate is the model?
9. Transfer learning for large image classification.

a. Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.), or alternatively you can just use an existing dataset, such as the flowers dataset or MIT's places dataset (requires registration, and it is huge).

b. Write a preprocessing step that will resize and crop the image to 299 × 299, with some randomness for data augmentation.

c. Using the pretrained Inception v3 model from the previous exercise, freeze all layers up to the bottleneck layer (i.e., the last layer before the output layer), and replace the output layer with the appropriate number of outputs for your new classification task (e.g., the flowers dataset has five mutually exclusive classes so the output layer must have five neurons and use the softmax activation function).

d. Split your dataset into a training set and a test set. Train the model on the training set and evaluate it on the test set.

10. Go through TensorFlow's DeepDream tutorial. It is a fun way to familiarize yourself with various ways of visualizing the patterns learned by a CNN, and to generate art using Deep Learning.

Solutions to these exercises are available in Appendix A.