hyperparameter values. We will use the Tox21 dataset from Chapter 4 as a case study to demonstrate these black-box optimization methods. The Tox21 dataset is small enough to make experimentation easy, but complex enough that hyperparameter optimization isn't trivial.

We note before setting off that none of these black-box algorithms works perfectly. As you will soon see, in practice, much human input is required to optimize hyperparameters.

Can't Hyperparameter Optimization Be Automated?

One of the long-running dreams of machine learning has been to automate the process of selecting model hyperparameters. Projects such as the "automated statistician" and others have sought to remove some of the drudgery from the hyperparameter selection process and make model construction more easily available to non-experts. However, in practice, there has typically been a steep cost in performance for the added convenience.

In recent years, there has been a surge of work focused on improving the algorithmic foundations of model tuning. Gaussian processes, evolutionary algorithms, and reinforcement learning have all been used to learn model hyperparameters and architectures with very limited human input. Recent work has shown that with large amounts of computing power, these algorithms can exceed expert performance in model tuning! But the overhead is severe, with dozens to hundreds of times greater computational power required.

For now, automatic model tuning is still not practical. All algorithms we cover in this section require significant manual tuning However, as hardware quality improves, we anticipate that hyperparameter learning will become increasingly automated. In the near term, we recommend strongly that all practitioners master the intricacies of hyperparameter tuning. A strong ability to hyperparameter tune is the skill that separates the expert from the novice.

Setting Up a Baseline

The first step in hyperparameter tuning is finding a *baseline*. A baseline is performance achievable by a robust (non–deep learning usually) algorithm. In general, random forests are a superb choice for setting baselines. As shown in Figure 5-3, random forests are an ensemble method that train many decision tree models on subsets of the input data and input features. These individual trees then vote on the outcome.

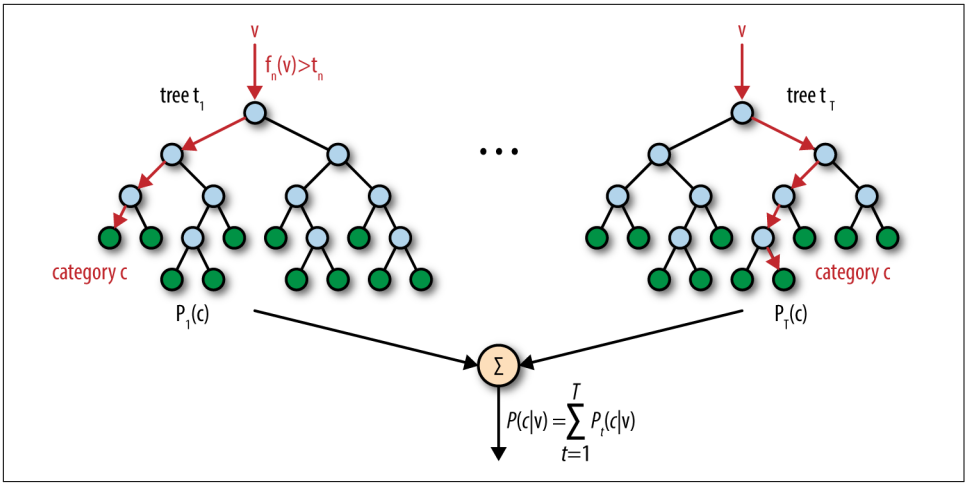

Figure 5-3. An illustration of a random forest. Here v is the input feature vector.

Random forests tend to be quite robust models. They are noise tolerant, and don't worry about the scale of their input features. (Although we don't have to worry about this for Tox21 since all our features are binary, in general deep networks are quite sensitive to their input range. It's healthy to normalize or otherwise scale the input range for good performance. We will return to this point in later chapters.) They also tend to have strong generalization and don't require much hyperparameter tuning to boot. For certain datasets, beating the performance of a random forest with a deep network can require considerable sophistication.

How can we create and train a random forest? Luckily for us, in Python, the scikit-learn library provides a high-quality implementation of a random forest. There are many tutorials and introductions to scikit-learn available, so we'll just display the training and prediction code needed to build a Tox21 random forest model here (Example 5-1).

Example 5-1. Defining and training a random forest on the Tox21 dataset

```
from sklearn.ensemble import RandomForestClassifier

# Generate tensorflow graph
sklearn_model = RandomForestClassifier(
    class_weight="balanced", n_estimators=50)
print("About to fit model on training set.")
sklearn_model.fit(train_X, train_y)

train_y_pred = sklearn_model.predict(train_X)
valid_y_pred = sklearn_model.predict(valid_X)
test_y_pred = sklearn_model.predict(test_X)
```

```
weighted_score = accuracy_score(train_y, train_y_pred, sample_weight=train_w)
print("Weighted train Classification Accuracy: %f" % weighted_score)
weighted_score = accuracy_score(valid_y, valid_y_pred, sample_weight=valid_w)
print("Weighted valid Classification Accuracy: %f" % weighted_score)
weighted_score = accuracy_score(test_y, test_y_pred, sample_weight=test_w)
print("Weighted test Classification Accuracy: %f" % weighted_score)
```

Here `train_X`, `train_y`, and so on are the Tox21 datasets defined in the previous chapter. Recall that all these quantities are NumPy arrays. `n_estimators` refers to the number of decision trees in our forest. Setting 50 or 100 trees often provides decent performance. Scikit-learn offers a simple object-oriented API with `fit(X, y)` and `predict(X)` methods. This model achieves the following accuracy with respect to our weighted accuracy metric:

```
Weighted train Classification Accuracy: 0.989845
Weighted valid Classification Accuracy: 0.681413
```

Recall that the fully connected network from Chapter 4 achieved performance:

```
Train Weighted Classification Accuracy: 0.742045
Valid Weighted Classification Accuracy: 0.648828
```

It looks like our baseline gets greater accuracy than our deep learning model! Time to roll up our sleeves and get to work.

Graduate Student Descent

The simplest method to try good hyperparameters is to simply try a number of different hyperparameter variants manually to see what works. This strategy can be surprisingly effective and educational. A deep learning practitioner needs to build up intuition about the structure of deep networks. Given the very weak state of theory, empirical work is the best way to learn how to build deep learning models. We highly recommend trying many variants of the fully connected model yourself. Be systematic; record your choices and results in a spreadsheet and systematically explore the space. Try to understand the effects of various hyperparameters. Which make network training proceed faster and which slower? What ranges of settings completely break learning? (These are quite easy to find, unfortunately.)

There are a few software engineering tricks that can make this search easier. Make a function whose arguments are the hyperparameter you wish to explore and have it print out the accuracy. Then trying new hyperparameter combinations requires only a single function call. Example 5-2 shows what this function signature would look like for our fully connected network from the Tox21 case study.