There is one subtlety in the process, however. Deep networks can be fairly sensitive to the choice of random seed used to initialize the network. For this reason, it's worth repeating each choice of hyperparameter settings multiple times and averaging the results to damp the variance.

The code to do this is straightforward, as Example 5-4 shows.

Example 5-4. Performing grid search on Tox21 fully connected network hyperparameters

```
scores = {}
n_reps = 3
hidden_sizes = [50]
epochs = [10]
dropouts = [.5, 1.0]
num_layers = [1, 2]

for rep in range(n_reps):
  for n_epochs in epochs:
    for hidden_size in hidden_sizes:
      for dropout in dropouts:
        for n_layers in num_layers:
          score = eval_tox21_hyperparams(n_hidden=hidden_size, n_epochs=n_epochs,
                                         dropout_prob=dropout, n_layers=n_layers)
          if (hidden_size, n_epochs, dropout, n_layers) not in scores:
            scores[(hidden_size, n_epochs, dropout, n_layers)] = []
          scores[(hidden_size, n_epochs, dropout, n_layers)].append(score)
print("All Scores")
print(scores)

avg_scores = {}
for params, param_scores in scores.iteritems():
  avg_scores[params] = np.mean(np.array(param_scores))
print("Scores Averaged over %d repetitions" % n_reps)
```

Random Hyperparameter Search

For experienced practitioners, it will often be very tempting to reuse magical hyperparameter settings or search grids that worked in previous applications. These settings can be valuable, but they can also lead us astray. Each machine learning problem is slightly different, and the optimal settings might lie in a region of parameter space we haven't previously considered. For that reason, it's often worthwhile to try random settings for hyperparameters (where the random values are chosen from a reasonable range).

There's also a deeper reason to try random searches. In higher-dimensional spaces, regular grids can miss a lot of information, especially if the spacing between grid

points isn't great. Selecting random choices for grid points can help us from falling into the trap of loose grids. Figure 5-4 illustrates this fact.

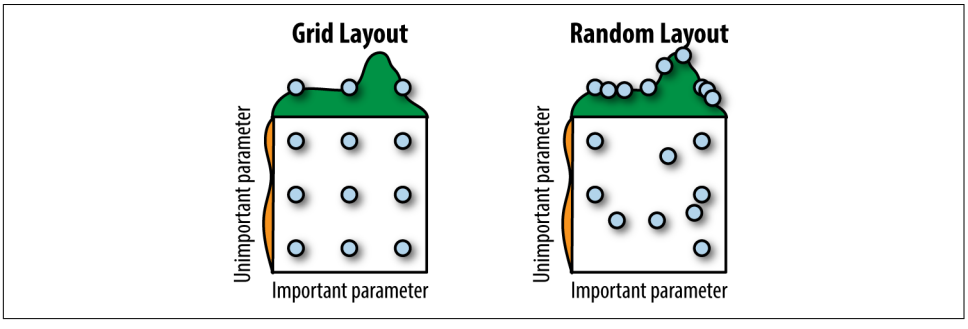

Figure 5-4. An illustration of why random hyperparameter search can be superior to grid search.

How can we implement random hyperparameter search in software? A neat software trick is to sample the random values desired up front and store them in a list. Then, random hyperparameter search simply turns into grid search over these randomly sampled lists. Here's an example. For learning rates, it's often useful to try a wide range from .1 to .000001 or so. Example 5-5 uses NumPy to sample some random learning rates.

Example 5-5. Sampling random learning rates

```
n_rates = 5
learning_rates = 10**(-np.random.uniform(low=1, high=6, size=n_rates))
```

We use a mathematical trick here. Note that $.1 = 10^{-1}$ and $.000001 = 10^{-6}$. Sampling real-valued numbers between ranges like 1 and 6 is easy with `np.random.uniform`. We can raise these sampled values to a power to recover our learning rates. Then `learning_rates` holds a list of values that we can feed into our grid search code from the previous section.

Challenge for the Reader

In this chapter, we've only covered the basics of hyperparameter tuning, but the tools covered are quite powerful. As a challenge, try tuning the fully connected deep network to achieve validation performance higher than that of the random forest. This might require a bit of work, but it's well worth the experience.