Hyperparameter Tuning

In the realm of machine learning, hyperparameter tuning is a “meta” learning task. It happens to be one of my favorite subjects because it can appear like black magic, yet its secrets are not impenetrable. In this chapter, we’ll talk about hyperparameter tuning in detail: why it’s hard, and what kind of smart tuning methods are being developed to do something about it.

Model Parameters Versus Hyperparameters

First, let’s define what a hyperparameter is, and how it is different from a normal nonhyper model parameter.

Machine learning models are basically mathematical functions that represent the relationship between different aspects of data. For instance, a linear regression model uses a line to represent the relationship between “features” and “target.” The formula looks like this:

$w^T x = y$

where x is a vector that represents features of the data and y is a scalar variable that represents the target (some numeric quantity that we wish to learn to predict).

This model assumes that the relationship between x and y is linear. The variable w is a weight vector that represents the normal vector for the line; it specifies the slope of the line. This is what’s known as a *model parameter*, which is learned during the training phase. “Training a model” involves using an optimization procedure to determine the best model parameter that “fits” the data.

There is another set of parameters known as *hyperparameters*, sometimes also knowns as "nuisance parameters." These are values that must be specified outside of the training procedure. Vanilla linear regression doesn't have any hyperparameters. But variants of linear regression do. Ridge regression and lasso both add a regularization term to linear regression; the weight for the regularization term is called the *regularization parameter*. Decision trees have hyperparameters such as the desired depth and number of leaves in the tree. Support vector machines (SVMs) require setting a misclassification penalty term. Kernelized SVMs require setting kernel parameters like the width for radial basis function (RBF) kernels. The list goes on.

What Do Hyperparameters Do?

A regularization hyperparameter controls the *capacity* of the model, i.e., how flexible the model is, how many degrees of freedom it has in fitting the data. Proper control of model capacity can prevent overfitting, which happens when the model is too flexible, and the training process adapts too much to the training data, thereby losing predictive accuracy on new test data. So a proper setting of the hyperparameters is important.

Another type of hyperparameter comes from the training process itself. Training a machine learning model often involves optimizing a loss function (the training metric). A number of mathematical optimization techniques may be employed, some of them having parameters of their own. For instance, stochastic gradient descent optimization requires a learning rate or a learning schedule. Some optimization methods require a convergence threshold. Random forests and boosted decision trees require knowing the number of total trees (though this could also be classified as a type of regularization hyperparameter). These also need to be set to reasonable values in order for the training process to find a good model.

Hyperparameter Tuning Mechanism

Hyperparameter settings could have a big impact on the prediction accuracy of the trained model. Optimal hyperparameter settings often differ for different datasets. Therefore they should be tuned for each dataset. Since the training process doesn't set the hyperparameters, there needs to be a meta process that tunes the hyperparameters. This is what we mean by hyperparameter tuning.

Hyperparameter tuning is a meta-optimization task. As Figure 4-1 shows, each trial of a particular hyperparameter setting involves training a model—an inner optimization process. The outcome of hyperparameter tuning is the best hyperparameter setting, and the outcome of model training is the best model parameter setting.

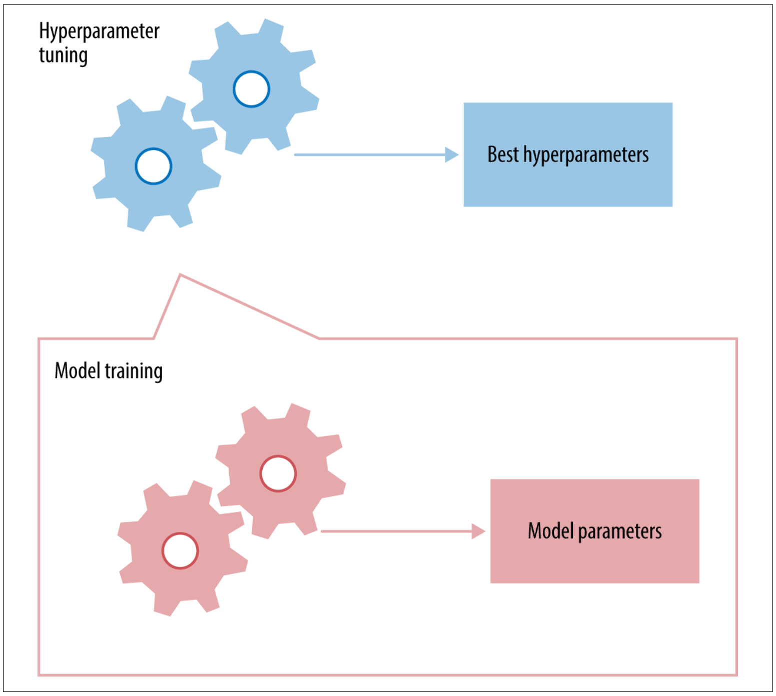

Figure 4-1. The relationship between hyperparameter tuning and model training

For each proposed hyperparameter setting, the inner model training process comes up with a model for the dataset and outputs evaluation results on hold-out or cross-validation datasets. After evaluating a number of hyperparameter settings, the hyperparameter tuner outputs the setting that yields the best performing model. The last step is to train a new model on the entire dataset (training and validation) under the best hyperparameter setting. Example 4-1 is a Pythonic version of the pseudocode. (The training and validation step can be conceptually replaced with a cross-validation step.)

Example 4-1. Pseudo-Python code for a very simple hyperparameter tuner

```
func hyperparameter_tuner (training_data,
                           validation_data,
                           hp_list):
    hp_perf = []

    # train and evaluate on all hyperparameter settings
    foreach hp_setting in hp_list:
        m = train_model(training_data, hp_setting)
        validation_results = eval_model(m, validation_data)
        hp_perf.append(validation_results)

    # find the best hyperparameter setting
    best_hp_setting = hp_list[max_index(hp_perf)]

    # IMPORTANT:
    # train a model on *all* available data using the best
    # hyperparameters
    best_m = train_model(training_data.append(validation_data),
                         best_hp_setting)

    return (best_hp_setting, best_m)
```

This pseudocode is correct for grid search and random search. But the smart search methods do not require a list of candidate settings as input. Rather it does something smarter than a for-loop through a static set of candidates. We'll see how later.

Hyperparameter Tuning Algorithms

Conceptually, hyperparameter tuning is an optimization task, just like model training.

However, these two tasks are quite different in practice. When training a model, the quality of a proposed set of model parameters can be written as a mathematical formula (usually called the loss function). When tuning hyperparameters, however, the quality of those hyperparameters cannot be written down in a closed-form formula, because it depends on the outcome of a black box (the model training process).

This is why hyperparameter tuning is much harder. Up until a few years ago, the only available methods were grid search and random search. In the last few years, there's been increased interest in autotuning. Several research groups have worked on the problem, published papers, and released new tools.

Grid Search

Grid search, true to its name, picks out a grid of hyperparameter values, evaluates every one of them, and returns the winner. For example, if the hyperparameter is the number of leaves in a decision tree, then the grid could be 10, 20, 30, …, 100. For regularization parameters, it's common to use exponential scale: 1e-5, 1e-4, 1e-3, …, 1. Some guesswork is necessary to specify the minimum and maximum values. So sometimes people run a small grid, see if the optimum lies at either endpoint, and then expand the grid in that direction. This is called manual grid search.

Grid search is dead simple to set up and trivial to parallelize. It is the most expensive method in terms of total computation time. However, if run in parallel, it is fast in terms of wall clock time.

Random Search

I love movies where the underdog wins, and I love machine learning papers where simple solutions are shown to be surprisingly effective. This is the storyline of "Random Search for Hyper Parameter Optimization" by Bergstra and Bengio. Random search is a slight variation on grid search. Instead of searching over the entire grid, random search only evaluates a random sample of points on the grid. This makes random search a lot cheaper than grid search. Random search wasn't taken very seriously before. This is because it doesn't search over all the grid points, so it cannot possibly beat the optimum found by grid search. But then along came Bergstra and Bengio. They showed that, in surprisingly many instances, random search performs about as well as grid search. All in all, trying 60 random points sampled from the grid seems to be good enough.

In hindsight, there is a simple probabilistic explanation for the result: for any distribution over a sample space with a finite maximum, the maximum of 60 random observations lies within the top 5% of the true maximum, with 95% probability. That may sound complicated, but it's not. Imagine the 5% interval around the true maximum. Now imagine that we sample points from this space and see if any of them land within that maximum. Each random draw has a 5% chance of landing in that interval; if we draw *n* points independently, then the probability that all of them miss the desired interval is $(1 - 0.05)^n$. So the probability that at least one of them succeeds in hitting the interval is 1 minus that quantity. We want at least a 0.95 probability of success. To figure out the number of draws we need, just solve for n in the following equation:

$$1 - (1 - 0.05)^n > 0.95$$

We get $n >= 60$. Ta-da!

The moral of the story is: *if at least 5% of the points on the grid yield a close-to-optimal solution, then random search with 60 trials will find that region with high probability*. The condition of the if-statement is very important. It can be satisfied if either the close-to-optimal region is large, or if somehow there is a high concentration of grid points in that region. The former is more likely, because a good machine learning model should not be overly sensitive to the hyperparameters, i.e., the close-to-optimal region is large.

With its utter simplicity and surprisingly reasonable performance, random search is my go-to method for hyperparameter tuning. It's trivially parallelizable, just like grid search, but it takes much fewer tries and performs almost as well most of the time.

Smart Hyperparameter Tuning

Smarter tuning methods are available. Unlike the "dumb" alternatives of grid search and random search, smart hyperparameter tuning is much less parallelizable. Instead of generating all the candidate points up front and evaluating the batch in parallel, smart tuning techniques pick a few hyperparameter settings, evaluate their quality, then decide where to sample next. This is an inherently iterative and sequential process. It is not very parallelizable. The goal is to make fewer evaluations overall and save on the overall computation time. If wall clock time is your goal, and you can afford multiple machines, then I suggest sticking to random search.

Buyer beware: smart search algorithms require computation time to figure out where to place the next set of samples. Some algorithms require much more time than others. Hence it only makes sense if the evaluation procedure—the inner optimization box—takes much longer than the process of evaluating where to sample next. Smart search algorithms also contain parameters of their own that need to be tuned. (Hyper-hyperparameters?) Sometimes tuning the hyper-

hyperparameters is crucial to make the smart search algorithm faster than random search.

Recall that hyperparameter tuning is difficult because we cannot write down the actual mathematical formula for the function we're optimizing. (The technical term for the function that is being optimized is *response surface*.) Consequently, we don't have the derivative of that function, and therefore most of the mathematical optimization tools that we know and love, such as the Newton method or stochastic gradient descent (SGD), cannot be applied.

I will highlight three smart tuning methods proposed in recent years: derivative-free optimization, Bayesian optimization, and random forest smart tuning. Derivative-free methods employ heuristics to determine where to sample next. Bayesian optimization and random forest smart tuning both model the response surface with another function, then sample more points based on what the model says.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams used Gaussian processes to model the response function and something called Expected Improvement to determine the next proposals. Gaussian processes are trippy; they specify distributions over *functions*. When one samples from a Gaussian process, one generates an entire function. Training a Gaussian process adapts this distribution over the data at hand, so that it generates functions that are more likely to model all of the data at once. Given the current estimate of the function, one can compute the amount of expected improvement of any point over the current optimum. They showed that this procedure of modeling the hyperparameter response surface and generating the next set of proposed hyperparameter settings can beat the evaluation cost of manual tuning.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown suggested training a random forest of regression trees to approximate the response surface. New points are sampled based on where the random forest considers to be the optimal regions. They call this SMAC (Sequential Model-based Algorithm Configuration). Word on the street is that this method works better than Gaussian processes for categorical hyperparameters.

Derivative-free optimization, as the name suggests, is a branch of mathematical optimization for situations where there is no derivative information. Notable derivative-free methods include genetic algorithms and the Nelder-Mead method. Essentially, the algorithms boil down to the following: try a bunch of random points, approximate the gradient, find the most likely search direction, and go there. A few years ago, Misha Bilenko and I tried Nelder-Mead for hyperparameter tuning. We found the algorithm delightfully easy to implement and no less efficient that Bayesian optimization.

The Case for Nested Cross-Validation

Before concluding this chapter, we need to go up one more level and talk about nested cross-validation, or nested hyperparameter tuning. (I suppose this makes it a meta-meta-learning task.)

There is a subtle difference between model selection and hyperparameter tuning. Model selection can include not just tuning the hyperparameters for a particular family of models (e.g., the depth of a decision tree); it can also include choosing between different model families (e.g., should I use decision tree or linear SVM?). Some advanced hyperparameter tuning methods claim to be able to choose between different model families. But most of the time this is not advisable. The hyperparameters for different kinds of models have nothing to do with each other, so it's best not to lump them together.

Choosing between different model families adds one more layer to our cake of prototyping models. Remember our discussion about why one must never mix training data and evaluation data? This means that we now must set aside validation data (or do cross-validation) for the hyperparameter tuner.

To make this precise, Example 4-2 shows the pseudocode in Python form. I use hold-out validation because it's simpler to code. You can do cross-validation or bootstrap validation, too. Note that at the end of each for loop, you should train the best model on *all* the available data at this stage.

Example 4-2. Pseudo-Python code for nested hyperparameter tuning

```
func nested_hp_tuning(data, model_family_list):
    perf_list = []
    hp_list = []

    for mf in model_family_list:
        # split data into 80% and 20% subsets
        # give subset A to the inner hyperparameter tuner,
        # save subset B for meta-evaluation
        A, B = train_test_split(data, 0.8)

        # further split A into training and validation sets
        C, D = train_test_split(A, 0.8)

        # generate_hp_candidates should be a function that knows
        # how to generate candidate hyperparameter settings
        # for any given model family
        hp_settings_list = generate_hp_candidates(mf)

        # run hyperparameter tuner to find best hyperparameters
        best_hp, best_m = hyperparameter_tuner(C, D,
                                               hp_settings_list)

        result = evaluate(best_m, B)
        perf_list.append(result)
        hp_list.append(best_hp)
        # end of inner hyperparameter tuning loop for a single
        # model family


    # find best model family (max_index is a helper function
    # that finds the index of the maximum element in a list)
    best_mf = model_family_list[max_index(perf_list)]
    best_hp = hp_list[max_index(perf_list)]

    # train a model from the best model family using all of
    # the data
    model = train_mf_model(best_mf, best_hp, data)
    return (best_mf, best_hp, model)
```

Hyperparameters can make a big difference in the performance of a machine learning model. Many Kaggle competitions come down to hyperparameter tuning. But after all, it is just another optimization task, albeit a difficult one. With all the smart tuning methods being invented, there is hope that manual hyperparameter tuning will soon be a thing of the past. Machine learning is about algorithms that make themselves smarter over time. (It's not a sinister Skynet;

it's just mathematics.) There's no reason that a machine learning model can't eventually learn to tune itself. We just need better optimization methods that can deal with complex response surfaces. We're almost there!

Related Reading

- "Random Search for Hyper-Parameter Optimization." James Bergstra and Yoshua Bengio. *Journal of Machine Learning Research*, 2012.
- "Algorithms for Hyper-Parameter Optimization." James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl." *Neural Information Processing Systems*, 2011. See also a SciPy 2013 talk by the authors.
- "Practical Bayesian Optimization of Machine Learning Algorithms." Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. *Neural Information Processing Systems*, 2012.
- "Sequential Model-Based Optimization for General Algorithm Configuration." Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. *Learning and Intelligent Optimization*, 2011.
- "Lazy Paired Hyper-Parameter Tuning." Alice Zheng and Mikhail Bilenko. *International Joint Conference on Artificial Intelligence*, 2013.
- *Introduction to Derivative-Free Optimization (MPS-SIAM Series on Optimization)*. Andrew R. Conn, Katya Scheinberg, and Luis N. Vincente, 2009.
- Gradient-Based Hyperparameter Optimization Through Reversible Learning. Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. *ArXiv*, 2015.

Software Packages

- Grid search and random search: GraphLab Create, scikit-learn.
- Bayesian optimization using Gaussian processes: Spearmint (from Jasper et al.)
- Bayesian optimization using Tree-based Parzen Estimators: Hyperopt (from Bergstra et al.)
- Random forest tuning: SMAC (from Hutter et al.)
- Hyper gradient: hypergrad (from Maclaurin et al.)

The Pitfalls of A/B Testing

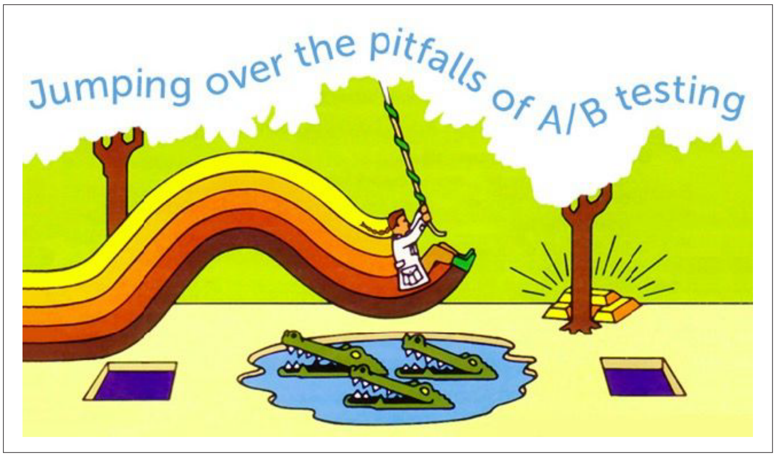

Figure 5-1. (Source: Eric Wolfe | Dato Design)

Thus far in this report, I've mainly focused on introducing the basic concepts in evaluating machine learning, with an occasional cautionary note here and there. This chapter is just the opposite. I'll give a cursory overview of the basics of A/B testing, and focus mostly on best practice tips. This is because there are many books and articles that teach statistical hypothesis testing, but relatively few articles about what can go wrong.

A/B testing is a widespread practice today. But a lot can go wrong in setting it up and interpreting the results. We'll discuss important questions to consider when doing A/B testing, followed by an overview of a promising alternative: multiarmed bandits.