Chapter 11

Practical Methodology

Successfully applying deep learning techniques requires more than just a good knowledge of what algorithms exist and the principles that explain how they work. A good machine learning practitioner also needs to know how to choose an algorithm for a particular application and how to monitor and respond to feedback obtained from experiments in order to improve a machine learning system. During day-to-day development of machine learning systems, practitioners need to decide whether to gather more data, increase or decrease model capacity, add or remove regularizing features, improve the optimization of a model, improve approximate inference in a model, or debug the software implementation of the model. All these operations are at the very least time consuming to try out, so it is important to be able to determine the right course of action rather than blindly guessing.

Most of this book is about different machine learning models, training algorithms, and objective functions. This may give the impression that the most important ingredient to being a machine learning expert is knowing a wide variety of machine learning techniques and being good at different kinds of math. In practice, one can usually do much better with a correct application of a commonplace algorithm than by sloppily applying an obscure algorithm. Correct application of an algorithm depends on mastering some fairly simple methodology. Many of the recommendations in this chapter are adapted from Ng (2015).

We recommend the following practical design process:

- Determine your goals—what error metric to use, and your target value for this error metric. These goals and error metrics should be driven by the problem that the application is intended to solve.
- Establish a working end-to-end pipeline as soon as possible, including the

estimation of the appropriate performance metrics.

- Instrument the system well to determine bottlenecks in performance. Diagnose which components are performing worse than expected and whether poor performance is due to overfitting, underfitting, or a defect in the data or software.

- Repeatedly make incremental changes such as gathering new data, adjusting hyperparameters, or changing algorithms, based on specific findings from your instrumentation.

As a running example, we will use the Street View address number transcription system (Goodfellow *et al.*, 2014d). The purpose of this application is to add buildings to Google Maps. Street View cars photograph the buildings and record the GPS coordinates associated with each photograph. A convolutional network recognizes the address number in each photograph, allowing the Google Maps database to add that address in the correct location. The story of how this commercial application was developed gives an example of how to follow the design methodology we advocate.

We now describe each of the steps in this process.

11.1 Performance Metrics

Determining your goals, in terms of which error metric to use, is a necessary first step because your error metric will guide all your future actions. You should also have an idea of what level of performance you desire.

Keep in mind that for most applications, it is impossible to achieve absolute zero error. The Bayes error defines the minimum error rate that you can hope to achieve, even if you have infinite training data and can recover the true probability distribution. This is because your input features may not contain complete information about the output variable, or because the system might be intrinsically stochastic. You will also be limited by having a finite amount of training data.

The amount of training data can be limited for a variety of reasons. When your goal is to build the best possible real-world product or service, you can typically collect more data but must determine the value of reducing error further and weigh this against the cost of collecting more data. Data collection can require time, money, or human suffering (for example, if your data collection process involves performing invasive medical tests). When your goal is to answer a scientific question about which algorithm performs better on a fixed benchmark, the benchmark

specification usually determines the training set, and you are not allowed to collect more data.

How can one determine a reasonable level of performance to expect? Typically, in the academic setting, we have some estimate of the error rate that is attainable based on previously published benchmark results. In the real-word setting, we have some idea of the error rate that is necessary for an application to be safe, cost-effective, or appealing to consumers. Once you have determined your realistic desired error rate, your design decisions will be guided by reaching this error rate.

Another important consideration besides the target value of the performance metric is the choice of which metric to use. Several different performance metrics may be used to measure the effectiveness of a complete application that includes machine learning components. These performance metrics are usually different from the cost function used to train the model. As described in section 5.1.2, it is common to measure the accuracy, or equivalently, the error rate, of a system.

However, many applications require more advanced metrics.

Sometimes it is much more costly to make one kind of a mistake than another. For example, an e-mail spam detection system can make two kinds of mistakes: incorrectly classifying a legitimate message as spam, and incorrectly allowing a spam message to appear in the inbox. It is much worse to block a legitimate message than to allow a questionable message to pass through. Rather than measuring the error rate of a spam classifier, we may wish to measure some form of total cost, where the cost of blocking legitimate messages is higher than the cost of allowing spam messages.

Sometimes we wish to train a binary classifier that is intended to detect some rare event. For example, we might design a medical test for a rare disease. Suppose that only one in every million people has this disease. We can easily achieve 99.9999 percent accuracy on the detection task, by simply hard coding the classifier to always report that the disease is absent. Clearly, accuracy is a poor way to characterize the performance of such a system. One way to solve this problem is to instead measure **precision** and **recall**. Precision is the fraction of detections reported by the model that were correct, while recall is the fraction of true events that were detected. A detector that says no one has the disease would achieve perfect precision, but zero recall. A detector that says everyone has the disease would achieve perfect recall, but precision equal to the percentage of people who have the disease (0.0001 percent in our example of a disease that only one people in a million have). When using precision and recall, it is common to plot a **PR curve**, with precision on the y-axis and recall on the x-axis. The classifier generates a score that is higher if the event to be detected occurred. For example, a feedforward

network designed to detect a disease outputs $\hat{y} = P(y = 1 \mid \boldsymbol{x})$, estimating the probability that a person whose medical results are described by features $\boldsymbol{x}$ has the disease. We choose to report a detection whenever this score exceeds some threshold. By varying the threshold, we can trade precision for recall. In many cases, we wish to summarize the performance of the classifier with a single number rather than a curve. To do so, we can convert precision p and recall r into an **F-score** given by

$$F = \frac{2pr}{p + r}. \tag{11.1}$$

Another option is to report the total area lying beneath the PR curve.

In some applications, it is possible for the machine learning system to refuse to make a decision. This is useful when the machine learning algorithm can estimate how confident it should be about a decision, especially if a wrong decision can be harmful and if a human operator is able to occasionally take over. The Street View transcription system provides an example of this situation. The task is to transcribe the address number from a photograph to associate the location where the photo was taken with the correct address in a map. Because the value of the map degrades considerably if the map is inaccurate, it is important to add an address only if the transcription is correct. If the machine learning system thinks that it is less likely than a human being to obtain the correct transcription, then the best course of action is to allow a human to transcribe the photo instead. Of course, the machine learning system is only useful if it is able to dramatically reduce the amount of photos that the human operators must process. A natural performance metric to use in this situation is **coverage**. Coverage is the fraction of examples for which the machine learning system is able to produce a response. It is possible to trade coverage for accuracy. One can always obtain 100 percent accuracy by refusing to process any example, but this reduces the coverage to 0 percent. For the Street View task, the goal for the project was to reach human-level transcription accuracy while maintaining 95 percent coverage. Human-level performance on this task is 98 percent accuracy.

Many other metrics are possible. We can, for example, measure click-through rates, collect user satisfaction surveys, and so on. Many specialized application areas have application-specific criteria as well.

What is important is to determine which performance metric to improve ahead of time, then concentrate on improving this metric. Without clearly defined goals, it can be difficult to tell whether changes to a machine learning system make progress or not.

11.2 Default Baseline Models

After choosing performance metrics and goals, the next step in any practical application is to establish a reasonable end-to-end system as soon as possible. In this section, we provide recommendations for which algorithms to use as the first baseline approach in various situations. Keep in mind that deep learning research progresses quickly, so better default algorithms are likely to become available soon after this writing.

Depending on the complexity of your problem, you may even want to begin without using deep learning. If your problem has a chance of being solved by just choosing a few linear weights correctly, you may want to begin with a simple statistical model like logistic regression.

If you know that your problem falls into an "AI-complete" category like object recognition, speech recognition, machine translation, and so on, then you are likely to do well by beginning with an appropriate deep learning model.

First, choose the general category of model based on the structure of your data. If you want to perform supervised learning with fixed-size vectors as input, use a feedforward network with fully connected layers. If the input has known topological structure (for example, if the input is an image), use a convolutional network. In these cases, you should begin by using some kind of piecewise linear unit (ReLUs or their generalizations, such as Leaky ReLUs, PreLus, or maxout). If your input or output is a sequence, use a gated recurrent net (LSTM or GRU).

A reasonable choice of optimization algorithm is SGD with momentum with a decaying learning rate (popular decay schemes that perform better or worse on different problems include decaying linearly until reaching a fixed minimum learning rate, decaying exponentially, or decreasing the learning rate by a factor of 2–10 each time validation error plateaus). Another reasonable alternative is Adam. Batch normalization can have a dramatic effect on optimization performance, especially for convolutional networks and networks with sigmoidal nonlinearities. While it is reasonable to omit batch normalization from the very first baseline, it should be introduced quickly if optimization appears to be problematic.

Unless your training set contains tens of millions of examples or more, you should include some mild forms of regularization from the start. Early stopping should be used almost universally. Dropout is an excellent regularizer that is easy to implement and compatible with many models and training algorithms. Batch normalization also sometimes reduces generalization error and allows dropout to be omitted, because of the noise in the estimate of the statistics used to normalize each variable.

If your task is similar to another task that has been studied extensively, you will probably do well by first copying the model and algorithm that is already known to perform best on the previously studied task. You may even want to copy a trained model from that task. For example, it is common to use the features from a convolutional network trained on ImageNet to solve other computer vision tasks (Girshick *et al.*, 2015).

A common question is whether to begin by using unsupervised learning, described further in part III. This is somewhat domain specific. Some domains, such as natural language processing, are known to benefit tremendously from unsupervised learning techniques, such as learning unsupervised word embeddings. In other domains, such as computer vision, current unsupervised learning techniques do not bring a benefit, except in the semi-supervised setting, when the number of labeled examples is very small (Kingma *et al.*, 2014; Rasmus *et al.*, 2015). If your application is in a context where unsupervised learning is known to be important, then include it in your first end-to-end baseline. Otherwise, only use unsupervised learning in your first attempt if the task you want to solve is unsupervised. You can always try adding unsupervised learning later if you observe that your initial baseline overfits.

11.3 Determining Whether to Gather More Data

After the first end-to-end system is established, it is time to measure the performance of the algorithm and determine how to improve it. Many machine learning novices are tempted to make improvements by trying out many different algorithms. Yet, it is often much better to gather more data than to improve the learning algorithm.

How does one decide whether to gather more data? First, determine whether the performance on the training set is acceptable. If performance on the training set is poor, the learning algorithm is not using the training data that is already available, so there is no reason to gather more data. Instead, try increasing the size of the model by adding more layers or adding more hidden units to each layer. Also, try improving the learning algorithm, for example by tuning the learning rate hyperparameter. If large models and carefully tuned optimization algorithms do not work well, then the problem might be the *quality* of the training data. The data may be too noisy or may not include the right inputs needed to predict the desired outputs. This suggests starting over, collecting cleaner data, or collecting a richer set of features.

If the performance on the training set is acceptable, then measure the per-

formance on a test set. If the performance on the test set is also acceptable, then there is nothing left to be done. If test set performance is much worse than training set performance, then gathering more data is one of the most effective solutions. The key considerations are the cost and feasibility of gathering more data, the cost and feasibility of reducing the test error by other means, and the amount of data that is expected to be necessary to improve test set performance significantly. At large internet companies with millions or billions of users, it is feasible to gather large datasets, and the expense of doing so can be considerably less than that of the alternatives, so the answer is almost always to gather more training data. For example, the development of large labeled datasets was one of the most important factors in solving object recognition. In other contexts, such as medical applications, it may be costly or infeasible to gather more data. A simple alternative to gathering more data is to reduce the size of the model or improve regularization, by adjusting hyperparameters such as weight decay coefficients, or by adding regularization strategies such as dropout. If you find that the gap between train and test performance is still unacceptable even after tuning the regularization hyperparameters, then gathering more data is advisable.

When deciding whether to gather more data, it is also necessary to decide how much to gather. It is helpful to plot curves showing the relationship between training set size and generalization error, as in figure 5.4. By extrapolating such curves, one can predict how much additional training data would be needed to achieve a certain level of performance. Usually, adding a small fraction of the total number of examples will not have a noticeable effect on generalization error. It is therefore recommended to experiment with training set sizes on a logarithmic scale, for example, doubling the number of examples between consecutive experiments.

If gathering much more data is not feasible, the only other way to improve generalization error is to improve the learning algorithm itself. This becomes the domain of research and not the domain of advice for applied practitioners.

11.4 Selecting Hyperparameters

Most deep learning algorithms come with several hyperparameters that control many aspects of the algorithm's behavior. Some of these hyperparameters affect the time and memory cost of running the algorithm. Some of these hyperparameters affect the quality of the model recovered by the training process and its ability to infer correct results when deployed on new inputs.

There are two basic approaches to choosing these hyperparameters: choosing them manually and choosing them automatically. Choosing the hyperparameters

manually requires understanding what the hyperparameters do and how machine learning models achieve good generalization. Automatic hyperparameter selection algorithms greatly reduce the need to understand these ideas, but they are often much more computationally costly.

11.4.1 Manual Hyperparameter Tuning

To set hyperparameters manually, one must understand the relationship between hyperparameters, training error, generalization error and computational resources (memory and runtime). This means establishing a solid foundation on the fundamental ideas concerning the effective capacity of a learning algorithm, as described in chapter 5.

The goal of manual hyperparameter search is usually to find the lowest generalization error subject to some runtime and memory budget. We do not discuss how to determine the runtime and memory impact of various hyperparameters here because this is highly platform dependent.

The primary goal of manual hyperparameter search is to adjust the effective capacity of the model to match the complexity of the task. Effective capacity is constrained by three factors: the representational capacity of the model, the ability of the learning algorithm to successfully minimize the cost function used to train the model, and the degree to which the cost function and training procedure regularize the model. A model with more layers and more hidden units per layer has higher representational capacity—it is capable of representing more complicated functions. It cannot necessarily learn all these functions though, if the training algorithm cannot discover that certain functions do a good job of minimizing the training cost, or if regularization terms such as weight decay forbid some of these functions.

The generalization error typically follows a U-shaped curve when plotted as a function of one of the hyperparameters, as in figure 5.3. At one extreme, the hyperparameter value corresponds to low capacity, and generalization error is high because training error is high. This is the underfitting regime. At the other extreme, the hyperparameter value corresponds to high capacity, and the generalization error is high because the gap between training and test error is high. Somewhere in the middle lies the optimal model capacity, which achieves the lowest possible generalization error, by adding a medium generalization gap to a medium amount of training error.

For some hyperparameters, overfitting occurs when the value of the hyperparameter is large. The number of hidden units in a layer is one such example,

because increasing the number of hidden units increases the capacity of the model. For some hyperparameters, overfitting occurs when the value of the hyperparameter is small. For example, the smallest allowable weight decay coefficient of zero corresponds to the greatest effective capacity of the learning algorithm.

Not every hyperparameter will be able to explore the entire U-shaped curve. Many hyperparameters are discrete, such as the number of units in a layer or the number of linear pieces in a maxout unit, so it is only possible to visit a few points along the curve. Some hyperparameters are binary. Usually these hyperparameters are switches that specify whether or not to use some optional component of the learning algorithm, such as a preprocessing step that normalizes the input features by subtracting their mean and dividing by their standard deviation. These hyperparameters can explore only two points on the curve. Other hyperparameters have some minimum or maximum value that prevents them from exploring some part of the curve. For example, the minimum weight decay coefficient is zero. This means that if the model is underfitting when weight decay is zero, we cannot enter the overfitting region by modifying the weight decay coefficient. In other words, some hyperparameters can only subtract capacity.

The learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate. It controls the effective capacity of the model in a more complicated way than other hyperparameters—the effective capacity of the model is highest when the learning rate is *correct* for the optimization problem, not when the learning rate is especially large or especially small. The learning rate has a U-shaped curve for *training* error, illustrated in figure 11.1. When the learning rate is too large, gradient descent can inadvertently increase rather than decrease the training error. In the idealized quadratic case, this occurs if the learning rate is at least twice as large as its optimal value (LeCun *et al.*, 1998a). When the learning rate is too small, training is not only slower but may become permanently stuck with a high training error. This effect is poorly understood (it would not happen for a convex loss function).

Tuning the parameters other than the learning rate requires monitoring both training and test error to diagnose whether your model is overfitting or underfitting, then adjusting its capacity appropriately.

If your error on the training set is higher than your target error rate, you have no choice but to increase capacity. If you are not using regularization and you are confident that your optimization algorithm is performing correctly, then you must add more layers to your network or add more hidden units. Unfortunately, this increases the computational costs associated with the model.

If your error on the test set is higher than your target error rate, you can now

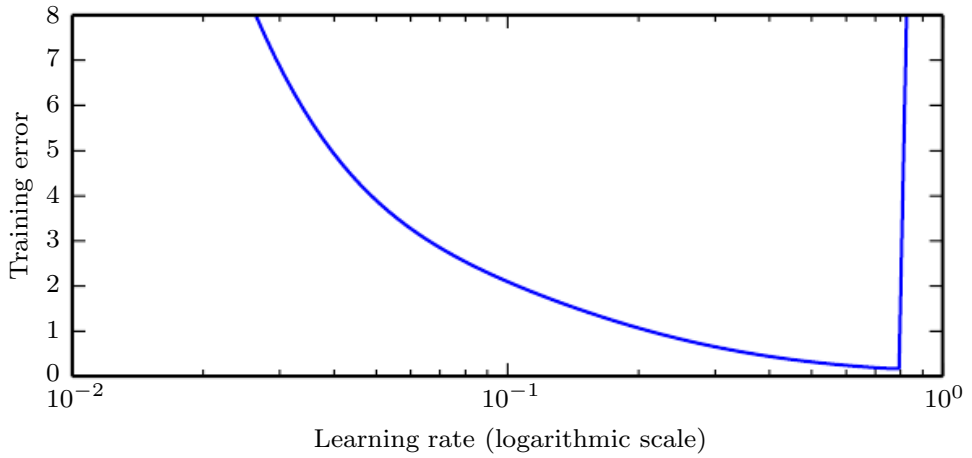

Figure 11.1: Typical relationship between the learning rate and the training error. Notice the sharp rise in error when the learning is above an optimal value. This is for a fixed training time, as a smaller learning rate may sometimes only slow down training by a factor proportional to the learning rate reduction. Generalization error can follow this curve or be complicated by regularization effects arising out of having too large or too small learning rates, since poor optimization can, to some degree, reduce or prevent overfitting, and even points with equivalent training error can have different generalization error.

take two kinds of actions. The test error is the sum of the training error and the gap between training and test error. The optimal test error is found by trading off these quantities. Neural networks typically perform best when the training error is very low (and thus, when capacity is high) and the test error is primarily driven by the gap between training and test error. Your goal is to reduce this gap without increasing training error faster than the gap decreases. To reduce the gap, change regularization hyperparameters to reduce effective model capacity, such as by adding dropout or weight decay. Usually the best performance comes from a large model that is regularized well, for example, by using dropout.

Most hyperparameters can be set by reasoning about whether they increase or decrease model capacity. Some examples are included in table 11.1.

While manually tuning hyperparameters, do not lose sight of your end goal: good performance on the test set. Adding regularization is only one way to achieve this goal. As long as you have low training error, you can always reduce generalization error by collecting more training data. The brute force way to practically guarantee success is to continually increase model capacity and training set size until the task is solved. This approach does of course increase the computational cost of training and inference, so it is only feasible given appropriate resources. In principle, this approach could fail due to optimization difficulties, but for many

Hyperparameter	Increases capacity when...	Reason	Caveats
Number of hidden units	increased	Increasing the number of hidden units increases the representational capacity of the model.	Increasing the number of hidden units increases both the time and memory cost of essentially every operation on the model.
Learning rate	tuned optimally	An improper learning rate, whether too high or too low, results in a model with low effective capacity due to optimization failure.	
Convolution kernel width	increased	Increasing the kernel width increases the number of parameters in the model.	A wider kernel results in a narrower output dimension, reducing model capacity unless you use implicit zero padding to reduce this effect. Wider kernels require more memory for parameter storage and increase runtime, but a narrower output reduces memory cost.
Implicit zero padding	increased	Adding implicit zeros before convolution keeps the representation size large.	Increases time and memory cost of most operations.
Weight decay coefficient	decreased	Decreasing the weight decay coefficient frees the model parameters to become larger.	
Dropout rate	decreased	Dropping units less often gives the units more opportunities to "conspire" with each other to fit the training set.	

Table 11.1: The effect of various hyperparameters on model capacity.

problems optimization does not seem to be a significant barrier, provided that the model is chosen appropriately.

11.4.2 Automatic Hyperparameter Optimization Algorithms

The ideal learning algorithm just takes a dataset and outputs a function, without requiring hand tuning of hyperparameters. The popularity of several learning algorithms such as logistic regression and SVMs stems in part from their ability to perform well with only one or two tuned hyperparameters. Neural networks can sometimes perform well with only a small number of tuned hyperparameters, but often benefit significantly from tuning of forty or more. Manual hyperparameter tuning can work very well when the user has a good starting point, such as one determined by others having worked on the same type of application and architecture, or when the user has months or years of experience in exploring hyperparameter values for neural networks applied to similar tasks. For many applications, however, these starting points are not available. In these cases, automated algorithms can find useful values of the hyperparameters.

If we think about the way in which the user of a learning algorithm searches for good values of the hyperparameters, we realize that an optimization is taking place: we are trying to find a value of the hyperparameters that optimizes an objective function, such as validation error, sometimes under constraints (such as a budget for training time, memory or recognition time). It is therefore possible, in principle, to develop **hyperparameter optimization** algorithms that wrap a learning algorithm and choose its hyperparameters, thus hiding the hyperparameters of the learning algorithm from the user. Unfortunately, hyperparameter optimization algorithms often have their own hyperparameters, such as the range of values that should be explored for each of the learning algorithm's hyperparameters. These secondary hyperparameters are usually easier to choose, however, in the sense that acceptable performance may be achieved on a wide range of tasks using the same secondary hyperparameters for all tasks.

11.4.3 Grid Search

When there are three or fewer hyperparameters, the common practice is to perform **grid search**. For each hyperparameter, the user selects a small finite set of values to explore. The grid search algorithm then trains a model for every joint specification of hyperparameter values in the Cartesian product of the set of values for each individual hyperparameter. The experiment that yields the best validation set error is then chosen as having found the best hyperparameters. See the left of figure 11.2 for an illustration of a grid of hyperparameter values.

How should the lists of values to search over be chosen? In the case of numerical (ordered) hyperparameters, the smallest and largest element of each list is chosen

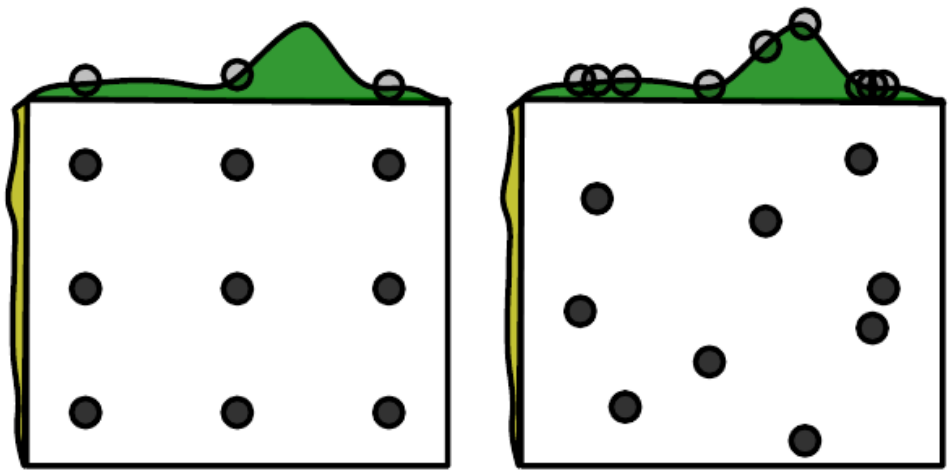

Figure 11.2: Comparison of grid search and random search. For illustration purposes, we display two hyperparameters, but we are typically interested in having many more. *(Left)* To perform grid search, we provide a set of values for each hyperparameter. The search algorithm runs training for every joint hyperparameter setting in the cross product of these sets. *(Right)* To perform random search, we provide a probability distribution over joint hyperparameter configurations. Usually most of these hyperparameters are independent from each other. Common choices for the distribution over a single hyperparameter include uniform and log-uniform (to sample from a log-uniform distribution, take the exp of a sample from a uniform distribution). The search algorithm then randomly samples joint hyperparameter configurations and runs training with each of them. Both grid search and random search evaluate the validation set error and return the best configuration. The figure illustrates the typical case where only some hyperparameters have a significant influence on the result. In this illustration, only the hyperparameter on the horizontal axis has a significant effect. Grid search wastes an amount of computation that is exponential in the number of noninfluential hyperparameters, while random search tests a unique value of every influential hyperparameter on nearly every trial. Figure reproduced with permission from Bergstra and Bengio (2012).

conservatively, based on prior experience with similar experiments, to make sure that the optimal value is likely to be in the selected range. Typically, a grid search involves picking values approximately on a *logarithmic scale*, e.g., a learning rate taken within the set $\{0.1, 0.01, 10^{-3}, 10^{-4}, 10^{-5}\}$, or a number of hidden units taken with the set $\{50, 100, 200, 500, 1000, 2000\}$.

Grid search usually performs best when it is performed repeatedly. For example, suppose that we ran a grid search over a hyperparameter α using values of $\{-1, 0, 1\}$. If the best value found is 1, then we underestimated the range in which the best α lies and should shift the grid and run another search with α in, for example, $\{1, 2, 3\}$. If we find that the best value of α is 0, then we may wish to refine our

estimate by zooming in and running a grid search over $\{-0.1, 0, 0.1\}$.

The obvious problem with grid search is that its computational cost grows exponentially with the number of hyperparameters. If there are m hyperparameters, each taking at most n values, then the number of training and evaluation trials required grows as $O(n^m)$. The trials may be run in parallel and exploit loose parallelism (with almost no need for communication between different machines carrying out the search). Unfortunately, because of the exponential cost of grid search, even parallelization may not provide a satisfactory size of search.

11.4.4 Random Search

Fortunately, there is an alternative to grid search that is as simple to program, more convenient to use, and converges much faster to good values of the hyperparameters: random search (Bergstra and Bengio, 2012).

A random search proceeds as follows. First we define a marginal distribution for each hyperparameter, for example, a Bernoulli or multinoulli for binary or discrete hyperparameters, or a uniform distribution on a log-scale for positive real-valued hyperparameters. For example,

$$\texttt{log\_learning\_rate} \sim u(-1, -5), \tag{11.2}$$

$$\texttt{learning\_rate} = 10^{\texttt{log\_learning\_rate}}, \tag{11.3}$$

where $u(a, b)$ indicates a sample of the uniform distribution in the interval (a, b). Similarly the `log_number_of_hidden_units` may be sampled from $u(\log(50), \log(2000))$.

Unlike in a grid search, we *should not discretize* or bin the values of the hyperparameters, so that we can explore a larger set of values and avoid additional computational cost. In fact, as illustrated in figure 11.2, a random search can be exponentially more efficient than a grid search, when there are several hyperparameters that do not strongly affect the performance measure. This is studied at length in Bergstra and Bengio (2012), who found that random search reduces the validation set error much faster than grid search, in terms of the number of trials run by each method.

As with grid search, we may often want to run repeated versions of random search, to refine the search based on the results of the first run.

The main reason that random search finds good solutions faster than grid search is that it has no wasted experimental runs, unlike in the case of grid search, when two values of a hyperparameter (given values of the other hyperparameters)

would give the same result. In the case of grid search, the other hyperparameters would have the same values for these two runs, whereas with random search, they would usually have different values. Hence if the change between these two values does not marginally make much difference in terms of validation set error, grid search will unnecessarily repeat two equivalent experiments while random search will still give two independent explorations of the other hyperparameters.

11.4.5 Model-Based Hyperparameter Optimization

The search for good hyperparameters can be cast as an optimization problem. The decision variables are the hyperparameters. The cost to be optimized is the validation set error that results from training using these hyperparameters. In simplified settings where it is feasible to compute the gradient of some differentiable error measure on the validation set with respect to the hyperparameters, we can simply follow this gradient (Bengio *et al.*, 1999; Bengio, 2000; Maclaurin *et al.*, 2015). Unfortunately, in most practical settings, this gradient is unavailable, either because of its high computation and memory cost, or because of hyperparameters that have intrinsically nondifferentiable interactions with the validation set error, as in the case of discrete-valued hyperparameters.

To compensate for this lack of a gradient, we can build a model of the validation set error, then propose new hyperparameter guesses by performing optimization within this model. Most model-based algorithms for hyperparameter search use a Bayesian regression model to estimate both the expected value of the validation set error for each hyperparameter and the uncertainty around this expectation. Optimization thus involves a trade-off between exploration (proposing hyperparameters for that there is high uncertainty, which may lead to a large improvement but may also perform poorly) and exploitation (proposing hyperparameters that the model is confident will perform as well as any hyperparameters it has seen so far—usually hyperparameters that are very similar to ones it has seen before). Contemporary approaches to hyperparameter optimization include Spearmint (Snoek *et al.*, 2012), TPE (Bergstra *et al.*, 2011) and SMAC (Hutter *et al.*, 2011).

Currently, we cannot unambiguously recommend Bayesian hyperparameter optimization as an established tool for achieving better deep learning results or for obtaining those results with less effort. Bayesian hyperparameter optimization sometimes performs comparably to human experts, sometimes better, but fails catastrophically on other problems. It may be worth trying to see if it works on a particular problem but is not yet sufficiently mature or reliable. That being said, hyperparameter optimization is an important field of research that, while often driven primarily by the needs of deep learning, holds the potential to benefit not

only the entire field of machine learning but also the discipline of engineering in general.

One drawback common to most hyperparameter optimization algorithms with more sophistication than random search is that they require for a training experiment to run to completion before they are able to extract any information from the experiment. This is much less efficient, in the sense of how much information can be gleaned early in an experiment, than manual search by a human practitioner, since one can usually tell early on if some set of hyperparameters is completely pathological. Swersky *et al.* (2014) have introduced an early version of an algorithm that maintains a set of multiple experiments. At various time points, the hyperparameter optimization algorithm can choose to begin a new experiment, to "freeze" a running experiment that is not promising, or to "thaw" and resume an experiment that was earlier frozen but now appears promising given more information.

11.5 Debugging Strategies

When a machine learning system performs poorly, it is usually difficult to tell whether the poor performance is intrinsic to the algorithm itself or whether there is a bug in the implementation of the algorithm. Machine learning systems are difficult to debug for various reasons.

In most cases, we do not know a priori what the intended behavior of the algorithm is. In fact, the entire point of using machine learning is that it will discover useful behavior that we were not able to specify ourselves. If we train a neural network on a *new* classification task and it achieves 5 percent test error, we have no straightforward way of knowing if this is the expected behavior or suboptimal behavior.

A further difficulty is that most machine learning models have multiple parts that are each adaptive. If one part is broken, the other parts can adapt and still achieve roughly acceptable performance. For example, suppose that we are training a neural net with several layers parametrized by weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$. Suppose further that we have manually implemented the gradient descent rule for each parameter separately, and we made an error in the update for the biases:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha, \tag{11.4}$$

where α is the learning rate. This erroneous update does not use the gradient at all. It causes the biases to constantly become negative throughout learning, which

is clearly not a correct implementation of any reasonable learning algorithm. The bug may not be apparent just from examining the output of the model though. Depending on the distribution of the input, the weights may be able to adapt to compensate for the negative biases.

Most debugging strategies for neural nets are designed to get around one or both of these two difficulties. Either we design a case that is so simple that the correct behavior actually can be predicted, or we design a test that exercises one part of the neural net implementation in isolation.

Some important debugging tests include the following.

Visualize the model in action: When training a model to detect objects in images, view some images with the detections proposed by the model displayed superimposed on the image. When training a generative model of speech, listen to some of the speech samples it produces. This may seem obvious, but it is easy to fall into the practice of looking only at quantitative performance measurements like accuracy or log-likelihood. Directly observing the machine learning model performing its task will help to determine whether the quantitative performance numbers it achieves seem reasonable. Evaluation bugs can be some of the most devastating bugs because they can mislead you into believing your system is performing well when it is not.

Visualize the worst mistakes: Most models are able to output some sort of confidence measure for the task they perform. For example, classifiers based on a softmax output layer assign a probability to each class. The probability assigned to the most likely class thus gives an estimate of the confidence the model has in its classification decision. Typically, maximum likelihood training results in these values being overestimates rather than accurate probabilities of correct prediction, but they are somewhat useful in the sense that examples that are actually less likely to be correctly labeled receive smaller probabilities under the model. By viewing the training set examples that are the hardest to model correctly, one can often discover problems with the way the data have been preprocessed or labeled. For example, the Street View transcription system originally had a problem where the address number detection system would crop the image too tightly and omit some digits. The transcription network then assigned very low probability to the correct answer on these images. Sorting the images to identify the most confident mistakes showed that there was a systematic problem with the cropping. Modifying the detection system to crop much wider images resulted in much better performance of the overall system, even though the transcription network needed to be able to process greater variation in the position and scale of the address numbers.

Reason about software using training and test error: It is often difficult to

determine whether the underlying software is correctly implemented. Some clues can be obtained from the training and test errors. If training error is low but test error is high, then it is likely that that the training procedure works correctly, and the model is overfitting for fundamental algorithmic reasons. An alternative possibility is that the test error is measured incorrectly because of a problem with saving the model after training then reloading it for test set evaluation, or because the test data was prepared differently from the training data. If both training and test errors are high, then it is difficult to determine whether there is a software defect or whether the model is underfitting due to fundamental algorithmic reasons. This scenario requires further tests, described next.

Fit a tiny dataset: If you have high error on the training set, determine whether it is due to genuine underfitting or due to a software defect. Usually even small models can be guaranteed to be able fit a sufficiently small dataset. For example, a classification dataset with only one example can be fit just by setting the biases of the output layer correctly. Usually if you cannot train a classifier to correctly label a single example, an autoencoder to successfully reproduce a single example with high fidelity, or a generative model to consistently emit samples resembling a single example, there is a software defect preventing successful optimization on the training set. This test can be extended to a small dataset with few examples.

Compare back-propagated derivatives to numerical derivatives: If you are using a software framework that requires you to implement your own gradient computations, or if you are adding a new operation to a differentiation library and must define its `bprop` method, then a common source of error is implementing this gradient expression incorrectly. One way to verify that these derivatives are correct is to compare the derivatives computed by your implementation of automatic differentiation to the derivatives computed by **finite differences**. Because

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}, \tag{11.5}$$

we can approximate the derivative by using a small, finite ϵ:

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{11.6}$$

We can improve the accuracy of the approximation by using the **centered difference**:

$$f'(x) \approx \frac{f(x+\frac{1}{2}\epsilon) - f(x-\frac{1}{2}\epsilon)}{\epsilon}. \tag{11.7}$$

The perturbation size ϵ must be large enough to ensure that the perturbation is not rounded down too much by finite-precision numerical computations.

Usually, we will want to test the gradient or Jacobian of a vector-valued function $g : \mathbb{R}^m \to \mathbb{R}^n$. Unfortunately, finite differencing only allows us to take a single derivative at a time. We can either run finite differencing mn times to evaluate all the partial derivatives of g, or apply the test to a new function that uses random projections at both the input and the output of g. For example, we can apply our test of the implementation of the derivatives to $f(x)$, where $f(x) = \boldsymbol{u}^T g(\boldsymbol{v}x)$, and $\boldsymbol{u}$ and $\boldsymbol{v}$ are randomly chosen vectors. Computing $f'(x)$ correctly requires being able to back-propagate through g correctly yet is efficient to do with finite differences because f has only a single input and a single output. It is usually a good idea to repeat this test for more than one value of $\boldsymbol{u}$ and $\boldsymbol{v}$ to reduce the chance of the test overlooking mistakes that are orthogonal to the random projection.

If one has access to numerical computation on complex numbers, then there is a very efficient way to numerically estimate the gradient by using complex numbers as input to the function (Squire and Trapp, 1998). The method is based on the observation that

$$f(x + i\epsilon) = f(x) + i\epsilon f'(x) + O(\epsilon^2), \tag{11.8}$$

$$\text{real}(f(x + i\epsilon)) = f(x) + O(\epsilon^2), \quad \text{imag}(\frac{f(x + i\epsilon)}{\epsilon}) = f'(x) + O(\epsilon^2), \tag{11.9}$$

where $i = \sqrt{-1}$. Unlike in the real-valued case above, there is no cancellation effect because we take the difference between the value of f at different points. This allows the use of tiny values of ϵ, like $\epsilon = 10^{-150}$, which make the $O(\epsilon^2)$ error insignificant for all practical purposes.

Monitor histograms of activations and gradient: It is often useful to visualize statistics of neural network activations and gradients, collected over a large amount of training iterations (maybe one epoch). The preactivation value of hidden units can tell us if the units saturate, or how often they do. For example, for rectifiers, how often are they off? Are there units that are always off? For tanh units, the average of the absolute value of the preactivations tells us how saturated the unit is. In a deep network where the propagated gradients quickly grow or quickly vanish, optimization may be hampered. Finally, it is useful to compare the magnitude of parameter gradients to the magnitude of the parameters themselves. As suggested by Bottou (2015), we would like the magnitude of parameter updates over a minibatch to represent something like 1 percent of the magnitude of the parameter, not 50 percent or 0.001 percent (which would make the parameters move too slowly). It may be that some groups of parameters are moving at a good pace while others are stalled. When the data is sparse (like in natural language),

some parameters may be very rarely updated, and this should be kept in mind when monitoring their evolution.

Finally, many deep learning algorithms provide some sort of guarantee about the results produced at each step. For example, in part III, we will see some approximate inference algorithms that work by using algebraic solutions to optimization problems. Typically these can be debugged by testing each of their guarantees. Some guarantees that some optimization algorithms offer include that the objective function will never increase after one step of the algorithm, that the gradient with respect to some subset of variables will be zero after each step of the algorithm, and that the gradient with respect to all variables will be zero at convergence. Usually due to rounding error, these conditions will not hold exactly in a digital computer, so the debugging test should include some tolerance parameter.

11.6 Example: Multi-Digit Number Recognition

To provide an end-to-end description of how to apply our design methodology in practice, we present a brief account of the Street View transcription system, from the point of view of designing the deep learning components. Obviously, many other components of the complete system, such as the Street View cars, the database infrastructure, and so on, were of paramount importance.

From the point of view of the machine learning task, the process began with data collection. The cars collected the raw data, and human operators provided labels. The transcription task was preceded by a significant amount of dataset curation, including using other machine learning techniques to *detect* the house numbers prior to transcribing them.

The transcription project began with a choice of performance metrics and desired values for these metrics. An important general principle is to tailor the choice of metric to the business goals for the project. Because maps are only useful if they have high accuracy, it was important to set a high accuracy requirement for this project. Specifically, the goal was to obtain human-level, 98 percent accuracy. This level of accuracy may not always be feasible to obtain. To reach this level of accuracy, the Street View transcription system sacrificed coverage. Coverage thus became the main performance metric optimized during the project, with accuracy held at 98 percent. As the convolutional network improved, it became possible to reduce the confidence threshold below which the network refused to transcribe the input, eventually exceeding the goal of 95 percent coverage.

After choosing quantitative goals, the next step in our recommended methodol-

ogy is to rapidly establish a sensible baseline system. For vision tasks, this means a convolutional network with rectified linear units. The transcription project began with such a model. At the time, it was not common for a convolutional network to output a sequence of predictions. To begin with the simplest possible baseline, the first implementation of the output layer of the model consisted of n different softmax units to predict a sequence of n characters. These softmax units were trained exactly the same as if the task were classification, with each softmax unit trained independently.

Our recommended methodology is to iteratively refine the baseline and test whether each change makes an improvement. The first change to the Street View transcription system was motivated by a theoretical understanding of the coverage metric and the structure of the data. Specifically, the network refused to classify an input $\boldsymbol{x}$ whenever the probability of the output sequence $p(\boldsymbol{y} \mid \boldsymbol{x}) < t$ for some threshold t. Initially, the definition of $p(\boldsymbol{y} \mid \boldsymbol{x})$ was ad-hoc, based on simply multiplying all the softmax outputs together. This motivated the development of a specialized output layer and cost function that actually computed a principled log-likelihood. This approach allowed the example rejection mechanism to function much more effectively.

At this point, coverage was still below 90 percent, yet there were no obvious theoretical problems with the approach. Our methodology therefore suggested instrumenting the training and test set performance to determine whether the problem was underfitting or overfitting. In this case, training and test set error were nearly identical. Indeed, the main reason this project proceeded so smoothly was the availability of a dataset with tens of millions of labeled examples. Because training and test set error were so similar, this suggested that the problem was due to either underfitting or a problem with the training data. One of the debugging strategies we recommend is to visualize the model's worst errors. In this case, that meant visualizing the incorrect training set transcriptions that the model gave the highest confidence. These proved to mostly consist of examples where the input image had been cropped too tightly, with some of the digits of the address being removed by the cropping operation. For example, a photo of an address "1849" might be cropped too tightly, with only the "849" remaining visible. This problem could have been resolved by spending weeks improving the accuracy of the address number detection system responsible for determining the cropping regions. Instead, the team made a much more practical decision, to simply expand the width of the crop region to be systematically wider than the address number detection system predicted. This single change added ten percentage points to the transcription system's coverage.

Finally, the last few percentage points of performance came from adjusting hyperparameters. This mostly consisted of making the model larger while maintaining some restrictions on its computational cost. Because train and test error remained roughly equal, it was always clear that any performance deficits were due to underfitting, as well as to a few remaining problems with the dataset itself.

Overall, the transcription project was a great success and allowed hundreds of millions of addresses to be transcribed both faster and at lower cost than would have been possible via human effort.

We hope that the design principles described in this chapter will lead to many other similar successes.