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 of 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 a lecture by Andrew Ng (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 application of interest.

- 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 it 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.

Determining your goals, in terms of which error metric to use, is a necessary first step because your error metric will guide all of 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. 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. 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). 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.

When should one decide to gather more data? More data always helps generalization, but the cost of collecting that data may not be justified by the incremental improvement. It is possible to forecast the effect of increasing the amount of data, typically using an empirical methodology resting on the assumption that the relationship between training set size and generalization error has a smooth and saturating shape. This makes it possible to approximately extrapolate from experiments done with less data than what is currently available. Note that adding a small fraction of the total number of examples will typically not have a noticeable impact 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.

This chapter contains further guidelines regarding the exploration of other aspects of the machine learning experiments, in particular concerning the choice of hyper-parameters, which often change the model size (which influences computational costs) or its capacity (its ability to extract more information from examples, often related to the number of free parameters), and yield different effects on the training set and validation or test sets.

11.1 Default Baseline Models

The first step in any practical application is to establish a reasonable end-to-end system. Note that the defaults suggested below are likely to change in the future as research progresses.

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 Naive Bayes.

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 feed-forward 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 like Leaky ReLUs, PreLus and 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 very reasonable alternative is Adam, although we expect that the technology for adaptive momentum and learning rates will improve in the future, so keep an eye on that.

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 is another excellent addition to standard neural networks and it has been shown to help substantially and is becoming commonly used. The authors of the paper introducing batch normalization (Ioffe and Szegedy, 2015) argue that

it acts as regularization strategy.

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.

A common question is whether to begin by using unsupervised pretraining. 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.

11.2 Selecting Hyperparameters

Most deep learning algorithms come with many 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 hyparameters: 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.

11.2.1 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 from Chapter 5.

The goal of manual hyperparameter search is usually to find the lowest generalization error subject to some runtime and memory budget. Understanding the runtime and memory effect of most hyperparameters is a relatively straightforward software engineering exercise[1] that we do not describe in detail here. Instead, we focus on how to reduce generalization error by modifying hyperparameters.

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 can not necessarily actually learn all of 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 Fig. 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. For example, models with a larger number of parameters tend to have higher capacity. For other hyperparameters, overfitting occurs when the value of the hyperparameter is small. For example, the smallest allowable weight decay coefficient of 0 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 to use some optional component of the learning algorithm, such as normalization of the input, or not. These hy-

[1] it becomes less straighforward when we consider a distributed computation setup, taking into account bandwidth and communication bottlenecks as well as reliability issues on large computer clusters

perparameters can only explore 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 0. This means that if the model is underfitting when weight decay is 0, we can not enter the overfitting region by modifying the weight decay coefficient.

The learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate. It also behaves somewhat differently from capacity control hyperparameters. The learning rate has a U-shaped curve for *training* error, illustraed in Fig. 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).

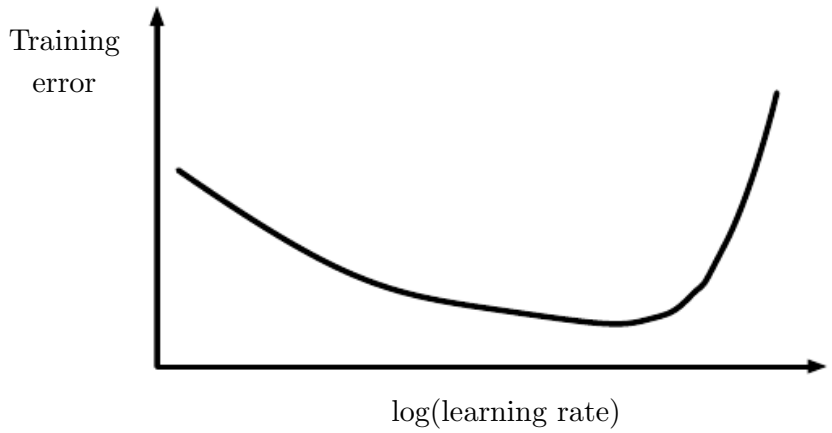

Figure 11.1: Typical relationship between the logarithm of the learning rate and the training error. Notice the sharp rise in error when the learning is above an optimal value. This is for a fixed training time, as a smaller learning rate may sometimes only slow down training by a factor proportional by the learning rate reduction. Generalization error can follow this curve or be complicated by regularization effects arising out of having a 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.

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 the model capacity.

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

For most hyperparameters, you can get a sense of how to set them 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 problems optimization does not seem to be a significant barrier, provided that the model is chosen appropriately.

11.2.2 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 from the fact that they can get away with one or two hyperparameters. For many years, the widespread use of neural networks has been hampered by the existence of many hyperparameters[2]. 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.

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 constraint (such as

[2] from 3 or 4 to possibly dozens if one wants to separately set hyperparameters for different layers

Hyper-parameter	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	Increased time and memory cost of most operations. Can be used to offset the output narrowing effect of wide kernels.
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.

a budget for training time, memory or recognition time). It is therefore possible, in principle, to apply optimization algorithms to *encapsulate a learning algorithm that has hyperparameters and yield a hyperparameter-free learning algorithm.* We call these procedures *hyperparameter optimization algorithms.* Unfortunately, they typically also involve hyperparameters, such as the range of values of be explored, but these tend to have much less influence on the final result.

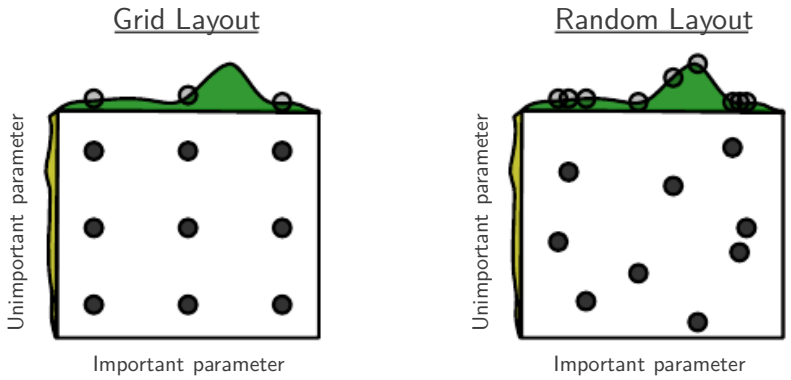

Figure 11.2: Comparison of grid search (left) and random search (right). For illustration purposes we display 2 hyperparameters but we are typically interested in having many more. To perform grid search, we provide a set of values for each hyperparameter. The search algorithm runs training for every joint hyperparameter setting in the cross product of these sets. 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 that case, grid search wastes an amount of computation that is exponential in the number of non-influential hyperparameters.

11.2.3 Grid Search

When the number of hyperparameters is 1 or 2, at most 3, the common practice is to perform what is called a *grid search.* For ordered hyperparameters (e.g., discrete-valued ones like number of hidden units, or continuous-valued ones like learning rate), one would select a small finite set of values which will be explored. See the left of Figure 11.2 for an illustration of a grid of hyperparameter values. How should these lists of values be chosen? In the case of numerical (ordered) hyperparameters, the smallest and largest element of each list is chosen conservatively, based on prior experience with similar experiments, to make sure that the optimal value is very 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 $\{.1, .01, 10^{-3}, 10^{-4}, 10^{-5}\}$, or a number of hidden units taken with the set $\{50, 100, 200, 500, 1000, 2000\}$. In effect, practitioners who do a grid search often do it repeatedly, zooming in on one or more regions in the space of hyperparameter values, in order to search for good configurations. When the optimal value found is on at an extreme of the range of values, it means that the range should be extended beyond that optimal value.

The obvious problem with a 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 $O(n^m)$. Fortunately, the trials may be run in parallel and exploit loose parallelism (with almost no need for communication between nodes) —but due to the exponential cost of grid search, even parallelization may not provide a satisfactory size of search.

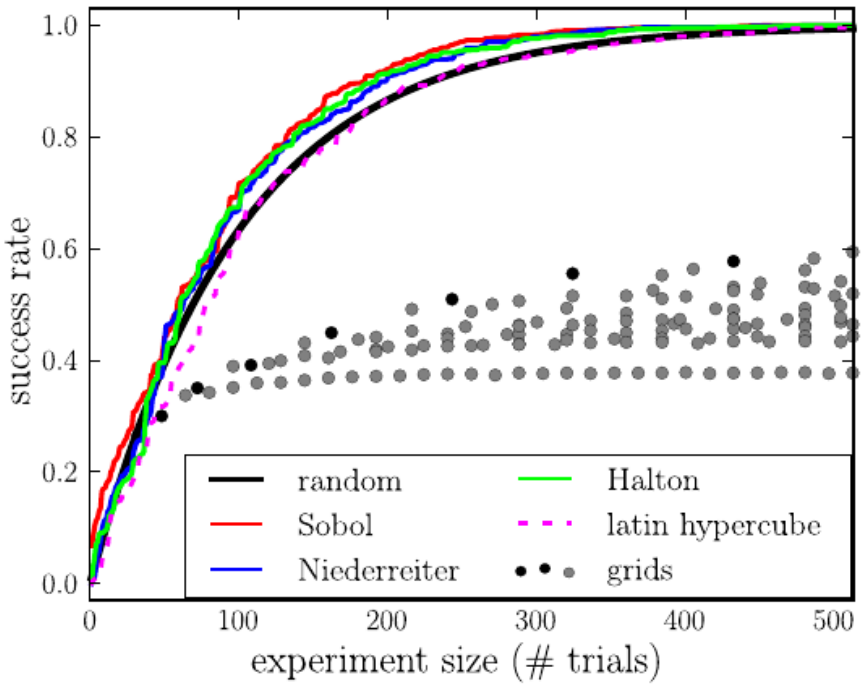

Figure 11.3: Experimental comparison of the convergence of grid search (circles) and random search (black curve) as well as quasi-random search (which better fill the space than a purely random search), see Bergstra and Bengio (2012). The vertical axis is the validation set accuracy and the horizontal axis is the number of training runs. We see that random or quasi-random search can converge much faster than grid search, for the reason illustrated in Figure 11.2.

11.2.4 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, e.g., a Bernoulli or multinoulli for binary or symbolic hyperparameters, or a uniform distribution on a log-scale. For example, to continue with the examples of number of hidden units and learning rate illustrated for the grid search, we could sample these hyperparameters as follows:

$$\begin{aligned} \texttt{log\_learning\_rate} &\sim U(-1, -5) \\ \texttt{learning\_rate} &= 10^{\texttt{log\_learning\_rate}}. \end{aligned} \tag{11.1}$$

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 $U(\log(50), \log(2000))$.

Note how, unlike in the case of a grid search, one *should not discretize* or bin the values of the hyperpameters. This allows one to explore a larger set of values, and does not incur additional computational cost. In fact, as illustrated in Fig. 11.2, a random search can be exponentially more efficient than a grid search, when there are several hyperparameters whose value do not change much in the result. This is studied at length in Bergstra and Bengio (2012), where the following advantages of of random search are highlighted:

1. The best validation error found as a function of the number of runs tends to converge faster with a random search than a grid search, as illustrated in the experiment from Bergstra and Bengio (2012) shown in Figure 11.3.

2. With random search, if some runs do not complete, it is not necessary to restart them in order to obtain interpretable and usable results, contrary to grid search.

3. If one wants to explore more finely and sample more runs, it is not necessary to restart from scratch: more runs can be added to the past runs, because all the runs are i.i.d., contrary to the grid case.

The main reason why random search finds good solutions faster than grid search is that the there are 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 have different values. Hence if the change between these two values does not marginally make much difference, grid search will unnecessarily repeat

two equivalent experiments whereas random search will still give two independent explorations of the other hyperparameters.

11.2.5 Model-Based Hyperparameter Optimization

The search for good hyperparameters can be cast as an optimization problem. The decision variables are the hyperpararameters. 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 due to its high computation and memory cost, or due to hyperparameters having intrinsically non-differentiable 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 tradeoff between exploration (proposing hyperparameters for which there is high uncertainty, which may lead to a large improvement but may also perform poorly) and exploitation (proposing hyperparameters which 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 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.

11.3 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 a variety of 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% test error, we have no straightforward way of knowing if this is the expected behavior or sub-optimal 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 parameterized by weights $\boldsymbol{W}$ and offsets $\boldsymbol{b}$. 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:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha$$

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:

Visualize the model in action : If you're training a model to detect objects in images, view some images with the bounding boxes on them. If you're 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 only looking at quantitative performance measurements like accuracy or log-likelihood. Directly observing the machine learning model performing its task will help you to determine whether the quantitative performance numbers you're achieving 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 score for the task they perform. For example, classifiers based on a softmax output layer give the probability assigned to each class. The probability assigned to the most likely class thus gives a confidence score. By viewing the training set examples that are the hardest to model correctly, you can often discover problems with the way the data has 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 of the 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.

Compare train and test error, evaluate if overfitting or underfitting: If both train and test error are high and similar in value to each other, this could mean that the model is underfitting, or it could mean that a bug is preventing proper fitting. If training error is low but test error is high, this usually just indicates that the model is overfitting, unless there is some difference in the way the test data is prepared or there is a problem with saving and reloading the model between training and test evaluation. To better ascertain whether one is underfitting or overfitting, it suffices to *vary one of the hyperparameters* and note the effect on the validation error. First, note whether the hyperparameter is one whose increases tends to increase effective capacity or decrease it (see Table 11.1 for some examples). The simplest one is the number of training iterations, which increases effective capacity. Second, verify if the increase in effective capacity (corresponding to the increase or decrease in the hyperparameter value) yields an increase or decrease in validation error. If the former, it means that we are overfitting (at least for that hyperparameter). If the latter, it means that we are

underfitting. It is therefore a good practice to always monitor the evolution of training and validation error as a function of the number of training iterations, since that immediately answers the question, and also tells us if the optimization is going astray (when training error goes up).

Fit a tiny dataset: If you have high error on the training set, determine whether it is due to genuine underfitting or a bug. Usually even small models can be guaranteed to 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 auto-encoder to successfully reproduce a single example with high fidelity, or a generative model to consistently emit samples resembling a single example, there is some sort of bug preventing successful optimization on the training set. When trying it out with just a few examples, more of the parameters are exercised and this tests the settings of the optimization hyperparameters, like learning rate.

Compare symbolic 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 symbolic operation to a symbolic differentiation environment and must define its gradient, 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 symbolic differentiation to the derivatives computed by a *finite differences*. Because

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

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

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

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

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

Note that ϵ cannot be chosen too small because, due to finite-precision computation (and especially the presence of non-linearities), the difference $f(x+\epsilon) - f(x)$ or $f(x+\frac{1}{2}\epsilon) - f(x-\frac{1}{2}\epsilon)$ might end up being numerically zero. This means that the Taylor expansion error can be non-negligeable, in some cases.

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 of the partial derivatives of g, or we can use random projections to construct a 1-dimensional function to test with a single call to finite differencing.

For example, we can apply finite differencing to $f(x)$ where $f(x) = \boldsymbol{u}^T g(\boldsymbol{v}x)$, where $\boldsymbol{u}$ and $\boldsymbol{v}$ are randomly chosen vectors. Computing $f'(x)$ correctly requires being able to back-propagate through g correctly. 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 that the test overlooks 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, when feeding the model with complex-valued parameters. Indeed, note that

$$f(x + i\epsilon) = f(x) + i\epsilon f'(x) + O(\epsilon^2)$$
$$\text{real}(f(x + i\epsilon)) = f(x) + O(\epsilon^2), \quad \text{imag}(\frac{f(x + i\epsilon)}{\epsilon}) = f'(x) + O(\epsilon^2), \tag{11.2}$$

where $i = \sqrt{-1}$. Unlike in the real case above, there is no cancellation effect due to taking the difference between the value of f at different points. Hence, it is possible to use tiny values of ϵ like $\epsilon = 10^{-150}$, which make the $O(\epsilon^2)$ error insignificant for all practical purposes. This was first remarked by Squire and Trapp (1998).

Monitor histograms of activations and gradient: It is often useful to visualize statistics of neural network activations and gradients, collected over a large chunk of data (maybe one epoch). The pre-activation 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 pre-activations 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% of the magnitude of the parameter, not 50% or 0.001% (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 have a sparse nature (like in natural language), some parameters may be very rarely updated, and this should be kept in mind when monitoring their evolution. See Glorot and Bengio (2010a) for an early example of using such monitoring to study the effect of initial values and nonlinearities, along with the evolution of training

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.