

## 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 [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 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.

As a running example, we will use 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 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.

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-world 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% 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% 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  $\hat{y} = P(y = 1 \mid \mathbf{x})$ , estimating the probability that a person whose medical results are described by features  $\mathbf{x}$  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

$$F = \frac{2pr}{p + r}. \quad (11.1)$$

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 in order 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% accuracy by refusing to process any example, but this reduces the coverage to 0%. For the Street View task, the goal for the project was to reach human-level transcription accuracy while maintaining 95% coverage. Human-level performance on this task is 98% 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 (ReLU 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. 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, due to 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. However, 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 the other 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, like 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 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.

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 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 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 from 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 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 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 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 zero. This means that if the model is underfitting when weight decay is zero, we can not 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 than your target error rate, you can

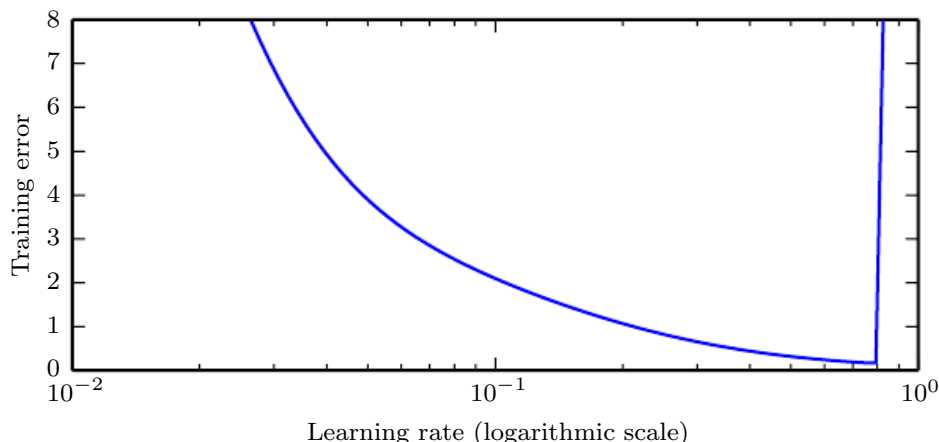


Figure 11.1: Typical relationship between the learning rate and the training error. Notice the sharp rise in error when the learning is above an optimal value. This is for a fixed training time, as a smaller learning rate may sometimes only slow down training by a factor proportional to the learning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

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.

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

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	Increased 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.

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.

### 11.4.2 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 hyperparameters. 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. However, for many applications, 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. However, these secondary hyperparameters are usually easier to choose, 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

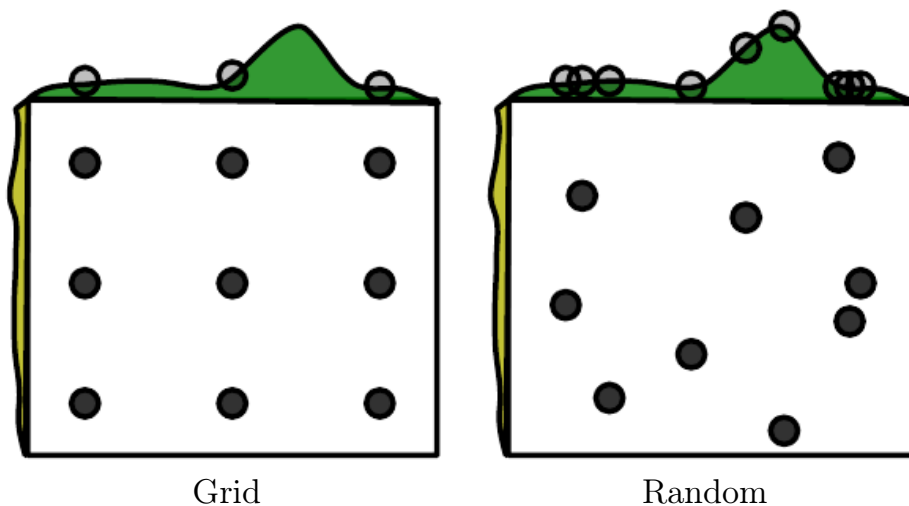


Figure 11.2: Comparison of grid search and random search. For illustration purposes we display two hyperparameters but we are typically interested in having many more. *(Left)* To perform grid search, we provide a set of values for each hyperparameter. The search algorithm runs training for every joint 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 non-influential hyperparameters, while random search tests a unique value of every influential hyperparameter on nearly every trial. Figure reproduced with permission from [Bergstra and Bengio \(2012\)](#).

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 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  $\{50, 100, 200, 500, 1000, 2000\}$ .

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

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, due to 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, e.g., a Bernoulli or multinoulli for binary or discrete hyperparameters, or a uniform distribution on a log-scale for positive real-valued hyperparameters. For example,

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

$$\text{learning\_rate} = 10^{\text{log\_learning\_rate}}. \tag{11.3}$$

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 the case of a grid search, one *should not discretize* or bin the values of the hyperparameters. This allows one to explore a larger set of values, and does not incur 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, one 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 why random search finds good solutions faster than grid search is that 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 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 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.

## 11.5 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 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 parametrized by weights  $\mathbf{W}$  and biases  $\mathbf{b}$ . 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:

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \tag{11.4}$$

where  $\alpha$  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* : 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 only looking 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 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.

*Reasoning about software using train and test error:* It is often difficult to determine whether the underlying software is correctly implemented. Some clues can be obtained from the train and test error. If training error is low but test error is high, then it is likely 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 due to a problem with saving the model after training then reloading it for test set evaluation, or if the test data was prepared differently from the training data. If both train and test error 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 a **finite differences**. Because

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

we can approximate the derivative by using a small, finite  $\epsilon$ :

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

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

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

The perturbation size  $\epsilon$  must be chosen to 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  $g : \mathbb{R}^m \rightarrow \mathbb{R}^n$ . 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 apply the test to a new function that uses random projections at both the input and output of  $g$ . For example, we can apply our test of the implementation of the derivatives to  $f(x)$  where  $f(x) = \mathbf{u}^T g(\mathbf{v}x)$ , where  $\mathbf{u}$  and  $\mathbf{v}$  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  $\mathbf{u}$  and  $\mathbf{v}$  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 by using complex numbers as input to the function (Squire and Trapp, 1998). The method is based on the observation that

$$f(x + i\epsilon) = f(x) + i\epsilon f'(x) + O(\epsilon^2) \quad (11.8)$$

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

where  $i = \sqrt{-1}$ . Unlike in the real-valued case above, there is no cancellation effect due to taking the difference between the value of  $f$  at different points. This allows the use of tiny values of  $\epsilon$  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 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 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% accuracy. This level of accuracy may not always be feasible to obtain. In order to reach this level of accuracy, the Street View transcription system sacrifices coverage. Coverage thus became the main performance metric optimized during the project, with accuracy held at 98%. As the convolutional network improved, it became possible to reduce the confidence threshold below which the network refuses to transcribe the input, eventually exceeding the goal of 95% coverage.

After choosing quantitative goals, the next step in our recommended methodology 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. In order 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 refuses to classify an input  $\mathbf{x}$  whenever the probability of the output sequence  $p(\mathbf{y} \mid \mathbf{x}) < t$  for some threshold  $t$ . Initially, the definition of  $p(\mathbf{y} \mid \mathbf{x})$  was ad-hoc, based on simply multiplying all of 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%, yet there were no obvious theoretical problems with the approach. Our methodology therefore suggests to instrument the train and test set performance in order to determine whether the problem is underfitting or overfitting. In this case, train 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 train and test set error were so similar, this suggested that the problem was either due

to underfitting or due to 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 took 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 due 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.

# Chapter 12

## Applications

In this chapter, we describe how to use deep learning to solve applications in computer vision, speech recognition, natural language processing, and other application areas of commercial interest. We begin by discussing the large scale neural network implementations required for most serious AI applications. Next, we review several specific application areas that deep learning has been used to solve. While one goal of deep learning is to design algorithms that are capable of solving a broad variety of tasks, so far some degree of specialization is needed. For example, vision tasks require processing a large number of input features (pixels) per example. Language tasks require modeling a large number of possible values (words in the vocabulary) per input feature.

### 12.1 Large-Scale Deep Learning

Deep learning is based on the philosophy of connectionism: while an individual biological neuron or an individual feature in a machine learning model is not intelligent, a large population of these neurons or features acting together can exhibit intelligent behavior. It truly is important to emphasize the fact that the number of neurons must be *large*. One of the key factors responsible for the improvement in neural network's accuracy and the improvement of the complexity of tasks they can solve between the 1980s and today is the dramatic increase in the size of the networks we use. As we saw in section 1.2.3, network sizes have grown exponentially for the past three decades, yet artificial neural networks are only as large as the nervous systems of insects.

Because the size of neural networks is of paramount importance, deep learning

requires high performance hardware and software infrastructure.

### 12.1.1 Fast CPU Implementations

Traditionally, neural networks were trained using the CPU of a single machine. Today, this approach is generally considered insufficient. We now mostly use GPU computing or the CPUs of many machines networked together. Before moving to these expensive setups, researchers worked hard to demonstrate that CPUs could not manage the high computational workload required by neural networks.

A description of how to implement efficient numerical CPU code is beyond the scope of this book, but we emphasize here that careful implementation for specific CPU families can yield large improvements. For example, in 2011, the best CPUs available could run neural network workloads faster when using fixed-point arithmetic rather than floating-point arithmetic. By creating a carefully tuned fixed-point implementation, [Vanhoucke \*et al.\* \(2011\)](#) obtained a threefold speedup over a strong floating-point system. Each new model of CPU has different performance characteristics, so sometimes floating-point implementations can be faster too. The important principle is that careful specialization of numerical computation routines can yield a large payoff. Other strategies, besides choosing whether to use fixed or floating point, include optimizing data structures to avoid cache misses and using vector instructions. Many machine learning researchers neglect these implementation details, but when the performance of an implementation restricts the size of the model, the accuracy of the model suffers.

### 12.1.2 GPU Implementations

Most modern neural network implementations are based on graphics processing units. Graphics processing units (GPUs) are specialized hardware components that were originally developed for graphics applications. The consumer market for video gaming systems spurred development of graphics processing hardware. The performance characteristics needed for good video gaming systems turn out to be beneficial for neural networks as well.

Video game rendering requires performing many operations in parallel quickly. Models of characters and environments are specified in terms of lists of 3-D coordinates of vertices. Graphics cards must perform matrix multiplication and division on many vertices in parallel to convert these 3-D coordinates into 2-D on-screen coordinates. The graphics card must then perform many computations at each pixel in parallel to determine the color of each pixel. In both cases, the



computations are fairly simple and do not involve much branching compared to the computational workload that a CPU usually encounters. For example, each vertex in the same rigid object will be multiplied by the same matrix; there is no need to evaluate an if statement per-vertex to determine which matrix to multiply by. The computations are also entirely independent of each other, and thus may be parallelized easily. The computations also involve processing massive buffers of memory, containing bitmaps describing the texture (color pattern) of each object to be rendered. Together, this results in graphics cards having been designed to have a high degree of parallelism and high memory bandwidth, at the cost of having a lower clock speed and less branching capability relative to traditional CPUs.

Neural network algorithms require the same performance characteristics as the real-time graphics algorithms described above. Neural networks usually involve large and numerous buffers of parameters, activation values, and gradient values, each of which must be completely updated during every step of training. These buffers are large enough to fall outside the cache of a traditional desktop computer so the memory bandwidth of the system often becomes the rate limiting factor. GPUs offer a compelling advantage over CPUs due to their high memory bandwidth. Neural network training algorithms typically do not involve much branching or sophisticated control, so they are appropriate for GPU hardware. Since neural networks can be divided into multiple individual “neurons” that can be processed independently from the other neurons in the same layer, neural networks easily benefit from the parallelism of GPU computing.

GPU hardware was originally so specialized that it could only be used for graphics tasks. Over time, GPU hardware became more flexible, allowing custom subroutines to be used to transform the coordinates of vertices or assign colors to pixels. In principle, there was no requirement that these pixel values actually be based on a rendering task. These GPUs could be used for scientific computing by writing the output of a computation to a buffer of pixel values. [Steinkrau \*et al.\* \(2005\)](#) implemented a two-layer fully connected neural network on a GPU and reported a threefold speedup over their CPU-based baseline. Shortly thereafter, [Chellapilla \*et al.\* \(2006\)](#) demonstrated that the same technique could be used to accelerate supervised convolutional networks.

The popularity of graphics cards for neural network training exploded after the advent of **general purpose GPUs**. These GP-GPUs could execute arbitrary code, not just rendering subroutines. NVIDIA’s CUDA programming language provided a way to write this arbitrary code in a C-like language. With their relatively convenient programming model, massive parallelism, and high memory

bandwidth, GP-GPUs now offer an ideal platform for neural network programming. This platform was rapidly adopted by deep learning researchers soon after it became available ([Raina \*et al.\*, 2009](#); [Ciresan \*et al.\*, 2010](#)).

Writing efficient code for GP-GPUs remains a difficult task best left to specialists. The techniques required to obtain good performance on GPU are very different from those used on CPU. For example, good CPU-based code is usually designed to read information from the cache as much as possible. On GPU, most writable memory locations are not cached, so it can actually be faster to compute the same value twice, rather than compute it once and read it back from memory. GPU code is also inherently multi-threaded and the different threads must be coordinated with each other carefully. For example, memory operations are faster if they can be **coalesced**. Coalesced reads or writes occur when several threads can each read or write a value that they need simultaneously, as part of a single memory transaction. Different models of GPUs are able to coalesce different kinds of read or write patterns. Typically, memory operations are easier to coalesce if among  $n$  threads, thread  $i$  accesses byte  $i + j$  of memory, and  $j$  is a multiple of some power of 2. The exact specifications differ between models of GPU. Another common consideration for GPUs is making sure that each thread in a group executes the same instruction simultaneously. This means that branching can be difficult on GPU. Threads are divided into small groups called **warps**. Each thread in a warp executes the same instruction during each cycle, so if different threads within the same warp need to execute different code paths, these different code paths must be traversed sequentially rather than in parallel.

Due to the difficulty of writing high performance GPU code, researchers should structure their workflow to avoid needing to write new GPU code in order to test new models or algorithms. Typically, one can do this by building a software library of high performance operations like convolution and matrix multiplication, then specifying models in terms of calls to this library of operations. For example, the machine learning library Pylearn2 ([Goodfellow \*et al.\*, 2013c](#)) specifies all of its machine learning algorithms in terms of calls to Theano ([Bergstra \*et al.\*, 2010](#); [Bastien \*et al.\*, 2012](#)) and cuda-convnet ([Krizhevsky, 2010](#)), which provide these high-performance operations. This factored approach can also ease support for multiple kinds of hardware. For example, the same Theano program can run on either CPU or GPU, without needing to change any of the calls to Theano itself. Other libraries like TensorFlow ([Abadi \*et al.\*, 2015](#)) and Torch ([Collobert \*et al.\*, 2011b](#)) provide similar features.

### 12.1.3 Large-Scale Distributed Implementations

In many cases, the computational resources available on a single machine are insufficient. We therefore want to distribute the workload of training and inference across many machines.

Distributing inference is simple, because each input example we want to process can be run by a separate machine. This is known as **data parallelism**.

It is also possible to get **model parallelism**, where multiple machines work together on a single datapoint, with each machine running a different part of the model. This is feasible for both inference and training.

Data parallelism during training is somewhat harder. We can increase the size of the minibatch used for a single SGD step, but usually we get less than linear returns in terms of optimization performance. It would be better to allow multiple machines to compute multiple gradient descent steps in parallel. Unfortunately, the standard definition of gradient descent is as a completely sequential algorithm: the gradient at step  $t$  is a function of the parameters produced by step  $t - 1$ .

This can be solved using **asynchronous stochastic gradient descent** (Bengio *et al.*, 2001; Recht *et al.*, 2011). In this approach, several processor cores share the memory representing the parameters. Each core reads parameters without a lock, then computes a gradient, then increments the parameters without a lock. This reduces the average amount of improvement that each gradient descent step yields, because some of the cores overwrite each other's progress, but the increased rate of production of steps causes the learning process to be faster overall. Dean *et al.* (2012) pioneered the multi-machine implementation of this lock-free approach to gradient descent, where the parameters are managed by a **parameter server** rather than stored in shared memory. Distributed asynchronous gradient descent remains the primary strategy for training large deep networks and is used by most major deep learning groups in industry (Chilimbi *et al.*, 2014; Wu *et al.*, 2015). Academic deep learning researchers typically cannot afford the same scale of distributed learning systems but some research has focused on how to build distributed networks with relatively low-cost hardware available in the university setting (Coates *et al.*, 2013).

### 12.1.4 Model Compression

In many commercial applications, it is much more important that the time and memory cost of running inference in a machine learning model be low than that the time and memory cost of training be low. For applications that do not require

personalization, it is possible to train a model once, then deploy it to be used by billions of users. In many cases, the end user is more resource-constrained than the developer. For example, one might train a speech recognition network with a powerful computer cluster, then deploy it on mobile phones.

A key strategy for reducing the cost of inference is **model compression** (Buciluă *et al.*, 2006). The basic idea of model compression is to replace the original, expensive model with a smaller model that requires less memory and runtime to store and evaluate.

Model compression is applicable when the size of the original model is driven primarily by a need to prevent overfitting. In most cases, the model with the lowest generalization error is an ensemble of several independently trained models. Evaluating all  $n$  ensemble members is expensive. Sometimes, even a single model generalizes better if it is large (for example, if it is regularized with dropout).

These large models learn some function  $f(\mathbf{x})$ , but do so using many more parameters than are necessary for the task. Their size is necessary only due to the limited number of training examples. As soon as we have fit this function  $f(\mathbf{x})$ , we can generate a training set containing infinitely many examples, simply by applying  $f$  to randomly sampled points  $\mathbf{x}$ . We then train the new, smaller, model to match  $f(\mathbf{x})$  on these points. In order to most efficiently use the capacity of the new, small model, it is best to sample the new  $\mathbf{x}$  points from a distribution resembling the actual test inputs that will be supplied to the model later. This can be done by corrupting training examples or by drawing points from a generative model trained on the original training set.

Alternatively, one can train the smaller model only on the original training points, but train it to copy other features of the model, such as its posterior distribution over the incorrect classes (Hinton *et al.*, 2014, 2015).

### 12.1.5 Dynamic Structure

One strategy for accelerating data processing systems in general is to build systems that have **dynamic structure** in the graph describing the computation needed to process an input. Data processing systems can dynamically determine which subset of many neural networks should be run on a given input. Individual neural networks can also exhibit dynamic structure internally by determining which subset of features (hidden units) to compute given information from the input. This form of dynamic structure inside neural networks is sometimes called **conditional computation** (Bengio, 2013; Bengio *et al.*, 2013b). Since many components of the architecture may be relevant only for a small amount of possible inputs, the

system can run faster by computing these features only when they are needed.

Dynamic structure of computations is a basic computer science principle applied generally throughout the software engineering discipline. The simplest versions of dynamic structure applied to neural networks are based on determining which subset of some group of neural networks (or other machine learning models) should be applied to a particular input.

A venerable strategy for accelerating inference in a classifier is to use a **cascade** of classifiers. The cascade strategy may be applied when the goal is to detect the presence of a rare object (or event). To know for sure that the object is present, we must use a sophisticated classifier with high capacity, that is expensive to run. However, because the object is rare, we can usually use much less computation to reject inputs as not containing the object. In these situations, we can train a sequence of classifiers. The first classifiers in the sequence have low capacity, and are trained to have high recall. In other words, they are trained to make sure we do not wrongly reject an input when the object is present. The final classifier is trained to have high precision. At test time, we run inference by running the classifiers in a sequence, abandoning any example as soon as any one element in the cascade rejects it. Overall, this allows us to verify the presence of objects with high confidence, using a high capacity model, but does not force us to pay the cost of full inference for every example. There are two different ways that the cascade can achieve high capacity. One way is to make the later members of the cascade individually have high capacity. In this case, the system as a whole obviously has high capacity, because some of its individual members do. It is also possible to make a cascade in which every individual model has low capacity but the system as a whole has high capacity due to the combination of many small models. [Viola and Jones \(2001\)](#) used a cascade of boosted decision trees to implement a fast and robust face detector suitable for use in handheld digital cameras. Their classifier localizes a face using essentially a sliding window approach in which many windows are examined and rejected if they do not contain faces. Another version of cascades uses the earlier models to implement a sort of hard attention mechanism: the early members of the cascade localize an object and later members of the cascade perform further processing given the location of the object. For example, Google transcribes address numbers from Street View imagery using a two-step cascade that first locates the address number with one machine learning model and then transcribes it with another ([Goodfellow \*et al.\*, 2014d](#)).

Decision trees themselves are an example of dynamic structure, because each node in the tree determines which of its subtrees should be evaluated for each input. A simple way to accomplish the union of deep learning and dynamic structure

is to train a decision tree in which each node uses a neural network to make the splitting decision (Guo and Gelfand, 1992), though this has typically not been done with the primary goal of accelerating inference computations.

In the same spirit, one can use a neural network, called the **gater** to select which one out of several **expert networks** will be used to compute the output, given the current input. The first version of this idea is called the **mixture of experts** (Nowlan, 1990; Jacobs *et al.*, 1991), in which the gater outputs a set of probabilities or weights (obtained via a softmax nonlinearity), one per expert, and the final output is obtained by the weighted combination of the output of the experts. In that case, the use of the gater does not offer a reduction in computational cost, but if a single expert is chosen by the gater for each example, we obtain the **hard mixture of experts** (Collobert *et al.*, 2001, 2002), which can considerably accelerate training and inference time. This strategy works well when the number of gating decisions is small because it is not combinatorial. But when we want to select different subsets of units or parameters, it is not possible to use a “soft switch” because it requires enumerating (and computing outputs for) all the gater configurations. To deal with this problem, several approaches have been explored to train combinatorial gaters. Bengio *et al.* (2013b) experiment with several estimators of the gradient on the gating probabilities, while Bacon *et al.* (2015) and Bengio *et al.* (2015a) use reinforcement learning techniques (policy gradient) to learn a form of conditional dropout on blocks of hidden units and get an actual reduction in computational cost without impacting negatively on the quality of the approximation.

Another kind of dynamic structure is a switch, where a hidden unit can receive input from different units depending on the context. This dynamic routing approach can be interpreted as an attention mechanism (Olshausen *et al.*, 1993). So far, the use of a hard switch has not proven effective on large-scale applications. Contemporary approaches instead use a weighted average over many possible inputs, and thus do not achieve all of the possible computational benefits of dynamic structure. Contemporary attention mechanisms are described in section 12.4.5.1.

One major obstacle to using dynamically structured systems is the decreased degree of parallelism that results from the system following different code branches for different inputs. This means that few operations in the network can be described as matrix multiplication or batch convolution on a minibatch of examples. We can write more specialized sub-routines that convolve each example with different kernels or multiply each row of a design matrix by a different set of columns of weights. Unfortunately, these more specialized subroutines are difficult to implement efficiently. CPU implementations will be slow due to the lack of cache



coherence and GPU implementations will be slow due to the lack of coalesced memory transactions and the need to serialize warps when members of a warp take different branches. In some cases, these issues can be mitigated by partitioning the examples into groups that all take the same branch, and processing these groups of examples simultaneously. This can be an acceptable strategy for minimizing the time required to process a fixed amount of examples in an offline setting. In a real-time setting where examples must be processed continuously, partitioning the workload can result in load-balancing issues. For example, if we assign one machine to process the first step in a cascade and another machine to process the last step in a cascade, then the first will tend to be overloaded and the last will tend to be underloaded. Similar issues arise if each machine is assigned to implement different nodes of a neural decision tree.

### 12.1.6 Specialized Hardware Implementations of Deep Networks

Since the early days of neural networks research, hardware designers have worked on specialized hardware implementations that could speed up training and/or inference of neural network algorithms. See early and more recent reviews of specialized hardware for deep networks ([Lindsey and Lindblad, 1994](#); [Beiu \*et al.\*, 2003](#); [Misra and Saha, 2010](#)).

Different forms of specialized hardware ([Graf and Jackel, 1989](#); [Mead and Ismail, 2012](#); [Kim \*et al.\*, 2009](#); [Pham \*et al.\*, 2012](#); [Chen \*et al.\*, 2014a,b](#)) have been developed over the last decades, either with ASICs (application-specific integrated circuit), either with digital (based on binary representations of numbers), analog ([Graf and Jackel, 1989](#); [Mead and Ismail, 2012](#)) (based on physical implementations of continuous values as voltages or currents) or hybrid implementations (combining digital and analog components). In recent years more flexible FPGA (field programmable gated array) implementations (where the particulars of the circuit can be written on the chip after it has been built) have been developed.

Though software implementations on general-purpose processing units (CPUs and GPUs) typically use 32 or 64 bits of precision to represent floating point numbers, it has long been known that it was possible to use less precision, at least at inference time ([Holt and Baker, 1991](#); [Holi and Hwang, 1993](#); [Presley and Haggard, 1994](#); [Simard and Graf, 1994](#); [Wawrzynek \*et al.\*, 1996](#); [Savich \*et al.\*, 2007](#)). This has become a more pressing issue in recent years as deep learning has gained in popularity in industrial products, and as the great impact of faster hardware was demonstrated with GPUs. Another factor that motivates current research on specialized hardware for deep networks is that the rate of progress of a single CPU or GPU core has slowed down, and most recent improvements in

computing speed have come from parallelization across cores (either in CPUs or GPUs). This is very different from the situation of the 1990s (the previous neural network era) where the hardware implementations of neural networks (which might take two years from inception to availability of a chip) could not keep up with the rapid progress and low prices of general-purpose CPUs. Building specialized hardware is thus a way to push the envelope further, at a time when new hardware designs are being developed for low-power devices such as phones, aiming for general-public applications of deep learning (e.g., with speech, computer vision or natural language).

Recent work on low-precision implementations of backprop-based neural nets (Vanhoucke *et al.*, 2011; Courbariaux *et al.*, 2015; Gupta *et al.*, 2015) suggests that between 8 and 16 bits of precision can suffice for using or training deep neural networks with back-propagation. What is clear is that more precision is required during training than at inference time, and that some forms of dynamic fixed point representation of numbers can be used to reduce how many bits are required per number. Traditional fixed point numbers are restricted to a fixed range (which corresponds to a given exponent in a floating point representation). Dynamic fixed point representations share that range among a set of numbers (such as all the weights in one layer). Using fixed point rather than floating point representations and using less bits per number reduces the hardware surface area, power requirements and computing time needed for performing multiplications, and multiplications are the most demanding of the operations needed to use or train a modern deep network with backprop.

## 12.2 Computer Vision

Computer vision has traditionally been one of the most active research areas for deep learning applications, because vision is a task that is effortless for humans and many animals but challenging for computers (Ballard *et al.*, 1983). Many of the most popular standard benchmark tasks for deep learning algorithms are forms of object recognition or optical character recognition.

Computer vision is a very broad field encompassing a wide variety of ways of processing images, and an amazing diversity of applications. Applications of computer vision range from reproducing human visual abilities, such as recognizing faces, to creating entirely new categories of visual abilities. As an example of the latter category, one recent computer vision application is to recognize sound waves from the vibrations they induce in objects visible in a video (Davis *et al.*, 2014). Most deep learning research on computer vision has not focused on such



exotic applications that expand the realm of what is possible with imagery but rather a small core of AI goals aimed at replicating human abilities. Most deep learning for computer vision is used for object recognition or detection of some form, whether this means reporting which object is present in an image, annotating an image with bounding boxes around each object, transcribing a sequence of symbols from an image, or labeling each pixel in an image with the identity of the object it belongs to. Because generative modeling has been a guiding principle of deep learning research, there is also a large body of work on image synthesis using deep models. While image synthesis *ex nihilo* is usually not considered a computer vision endeavor, models capable of image synthesis are usually useful for image restoration, a computer vision task involving repairing defects in images or removing objects from images.

### 12.2.1 Preprocessing

Many application areas require sophisticated preprocessing because the original input comes in a form that is difficult for many deep learning architectures to represent. Computer vision usually requires relatively little of this kind of preprocessing. The images should be standardized so that their pixels all lie in the same, reasonable range, like  $[0,1]$  or  $[-1, 1]$ . Mixing images that lie in  $[0,1]$  with images that lie in  $[0, 255]$  will usually result in failure. Formatting images to have the same scale is the only kind of preprocessing that is strictly necessary. Many computer vision architectures require images of a standard size, so images must be cropped or scaled to fit that size. Even this rescaling is not always strictly necessary. Some convolutional models accept variably-sized inputs and dynamically adjust the size of their pooling regions to keep the output size constant (Waibel *et al.*, 1989). Other convolutional models have variable-sized output that automatically scales in size with the input, such as models that denoise or label each pixel in an image (Hadsell *et al.*, 2007).

Dataset augmentation may be seen as a way of preprocessing the training set only. Dataset augmentation is an excellent way to reduce the generalization error of most computer vision models. A related idea applicable at test time is to show the model many different versions of the same input (for example, the same image cropped at slightly different locations) and have the different instantiations of the model vote to determine the output. This latter idea can be interpreted as an ensemble approach, and helps to reduce generalization error.

Other kinds of preprocessing are applied to both the train and the test set with the goal of putting each example into a more canonical form in order to reduce the amount of variation that the model needs to account for. Reducing the amount of

variation in the data can both reduce generalization error and reduce the size of the model needed to fit the training set. Simpler tasks may be solved by smaller models, and simpler solutions are more likely to generalize well. Preprocessing of this kind is usually designed to remove some kind of variability in the input data that is easy for a human designer to describe and that the human designer is confident has no relevance to the task. When training with large datasets and large models, this kind of preprocessing is often unnecessary, and it is best to just let the model learn which kinds of variability it should become invariant to. For example, the AlexNet system for classifying ImageNet only has one preprocessing step: subtracting the mean across training examples of each pixel (Krizhevsky *et al.*, 2012).

### 12.2.1.1 Contrast Normalization

One of the most obvious sources of variation that can be safely removed for many tasks is the amount of contrast in the image. Contrast simply refers to the magnitude of the difference between the bright and the dark pixels in an image. There are many ways of quantifying the contrast of an image. In the context of deep learning, contrast usually refers to the standard deviation of the pixels in an image or region of an image. Suppose we have an image represented by a tensor  $\mathbf{X} \in \mathbb{R}^{r \times c \times 3}$ , with  $X_{i,j,1}$  being the red intensity at row  $i$  and column  $j$ ,  $X_{i,j,2}$  giving the green intensity and  $X_{i,j,3}$  giving the blue intensity. Then the contrast of the entire image is given by

$$\sqrt{\frac{1}{3rc} \sum_{i=1}^r \sum_{j=1}^c \sum_{k=1}^3 (X_{i,j,k} - \bar{\mathbf{X}})^2} \quad (12.1)$$

where  $\bar{\mathbf{X}}$  is the mean intensity of the entire image:

$$\bar{\mathbf{X}} = \frac{1}{3rc} \sum_{i=1}^r \sum_{j=1}^c \sum_{k=1}^3 X_{i,j,k}. \quad (12.2)$$

**Global contrast normalization** (GCN) aims to prevent images from having varying amounts of contrast by subtracting the mean from each image, then rescaling it so that the standard deviation across its pixels is equal to some constant  $s$ . This approach is complicated by the fact that no scaling factor can change the contrast of a zero-contrast image (one whose pixels all have equal intensity). Images with very low but non-zero contrast often have little information content. Dividing by the true standard deviation usually accomplishes nothing

more than amplifying sensor noise or compression artifacts in such cases. This motivates introducing a small, positive regularization parameter  $\lambda$  to bias the estimate of the standard deviation. Alternately, one can constrain the denominator to be at least  $\epsilon$ . Given an input image  $\mathbf{X}$ , GCN produces an output image  $\mathbf{X}'$ , defined such that

$$X'_{i,j,k} = s \frac{X_{i,j,k} - \bar{X}}{\max \left\{ \epsilon, \sqrt{\lambda + \frac{1}{3rc} \sum_{i=1}^r \sum_{j=1}^c \sum_{k=1}^3 (X_{i,j,k} - \bar{X})^2} \right\}}. \quad (12.3)$$

Datasets consisting of large images cropped to interesting objects are unlikely to contain any images with nearly constant intensity. In these cases, it is safe to practically ignore the small denominator problem by setting  $\lambda = 0$  and avoid division by 0 in extremely rare cases by setting  $\epsilon$  to an extremely low value like  $10^{-8}$ . This is the approach used by Goodfellow *et al.* (2013a) on the CIFAR-10 dataset. Small images cropped randomly are more likely to have nearly constant intensity, making aggressive regularization more useful. Coates *et al.* (2011) used  $\epsilon = 0$  and  $\lambda = 10$  on small, randomly selected patches drawn from CIFAR-10.

The scale parameter  $s$  can usually be set to 1, as done by Coates *et al.* (2011), or chosen to make each individual pixel have standard deviation across examples close to 1, as done by Goodfellow *et al.* (2013a).

The standard deviation in equation 12.3 is just a rescaling of the  $L^2$  norm of the image (assuming the mean of the image has already been removed). It is preferable to define GCN in terms of standard deviation rather than  $L^2$  norm because the standard deviation includes division by the number of pixels, so GCN based on standard deviation allows the same  $s$  to be used regardless of image size. However, the observation that the  $L^2$  norm is proportional to the standard deviation can help build a useful intuition. One can understand GCN as mapping examples to a spherical shell. See figure 12.1 for an illustration. This can be a useful property because neural networks are often better at responding to directions in space rather than exact locations. Responding to multiple distances in the same direction requires hidden units with collinear weight vectors but different biases. Such coordination can be difficult for the learning algorithm to discover. Additionally, many shallow graphical models have problems with representing multiple separated modes along the same line. GCN avoids these problems by reducing each example to a direction rather than a direction and a distance.

Counterintuitively, there is a preprocessing operation known as **sphering** and it is not the same operation as GCN. Sphering does not refer to making the data lie on a spherical shell, but rather to rescaling the principal components to have

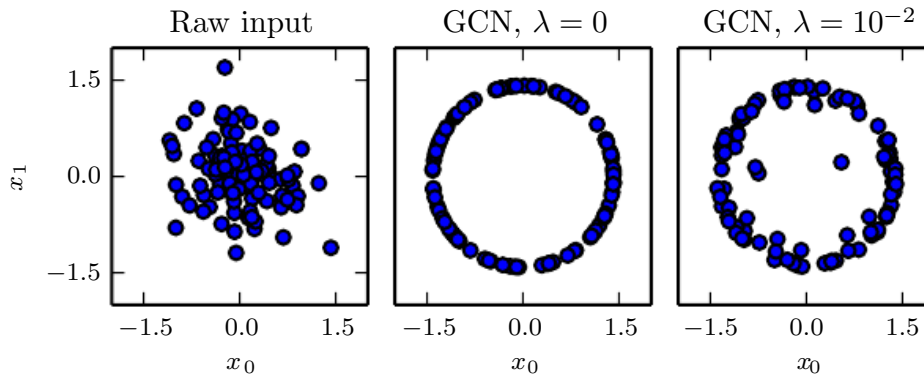


Figure 12.1: GCN maps examples onto a sphere. *(Left)* Raw input data may have any norm. *(Center)* GCN with  $\lambda = 0$  maps all non-zero examples perfectly onto a sphere. Here we use  $s = 1$  and  $\epsilon = 10^{-8}$ . Because we use GCN based on normalizing the standard deviation rather than the  $L^2$  norm, the resulting sphere is not the unit sphere. *(Right)* Regularized GCN, with  $\lambda > 0$ , draws examples toward the sphere but does not completely discard the variation in their norm. We leave  $s$  and  $\epsilon$  the same as before.

equal variance, so that the multivariate normal distribution used by PCA has spherical contours. Sphering is more commonly known as **whitening**.

Global contrast normalization will often fail to highlight image features we would like to stand out, such as edges and corners. If we have a scene with a large dark area and a large bright area (such as a city square with half the image in the shadow of a building) then global contrast normalization will ensure there is a large difference between the brightness of the dark area and the brightness of the light area. It will not, however, ensure that edges within the dark region stand out.

This motivates **local contrast normalization**. Local contrast normalization ensures that the contrast is normalized across each small window, rather than over the image as a whole. See figure 12.2 for a comparison of global and local contrast normalization.

Various definitions of local contrast normalization are possible. In all cases, one modifies each pixel by subtracting a mean of nearby pixels and dividing by a standard deviation of nearby pixels. In some cases, this is literally the mean and standard deviation of all pixels in a rectangular window centered on the pixel to be modified (Pinto *et al.*, 2008). In other cases, this is a weighted mean and weighted standard deviation using Gaussian weights centered on the pixel to be modified. In the case of color images, some strategies process different color

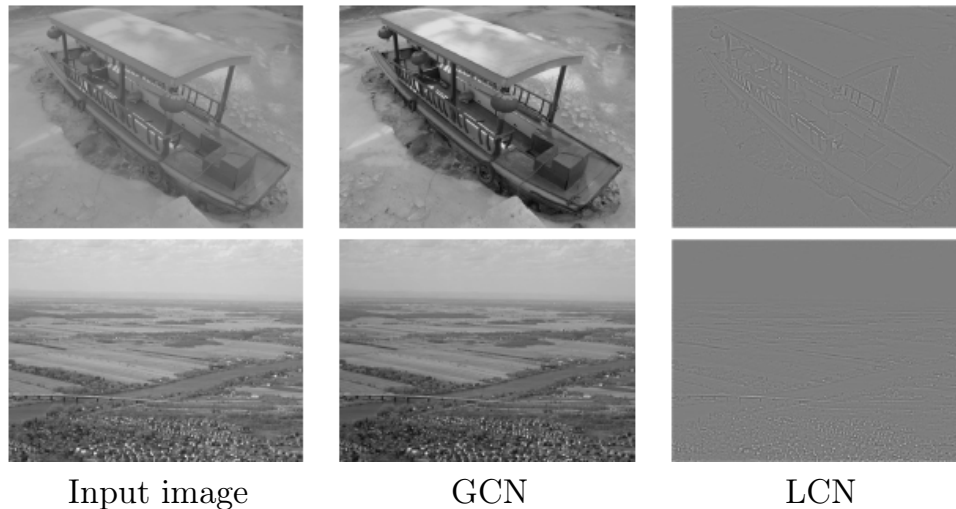


Figure 12.2: A comparison of global and local contrast normalization. Visually, the effects of global contrast normalization are subtle. It places all images on roughly the same scale, which reduces the burden on the learning algorithm to handle multiple scales. Local contrast normalization modifies the image much more, discarding all regions of constant intensity. This allows the model to focus on just the edges. Regions of fine texture, such as the houses in the second row, may lose some detail due to the bandwidth of the normalization kernel being too high.

channels separately while others combine information from different channels to normalize each pixel ([Sermanet \*et al.\*, 2012](#)).

Local contrast normalization can usually be implemented efficiently by using separable convolution (see section 9.8) to compute feature maps of local means and local standard deviations, then using element-wise subtraction and element-wise division on different feature maps.

Local contrast normalization is a differentiable operation and can also be used as a nonlinearity applied to the hidden layers of a network, as well as a preprocessing operation applied to the input.

As with global contrast normalization, we typically need to regularize local contrast normalization to avoid division by zero. In fact, because local contrast normalization typically acts on smaller windows, it is even more important to regularize. Smaller windows are more likely to contain values that are all nearly the same as each other, and thus more likely to have zero standard deviation.

### 12.2.1.2 Dataset Augmentation

As described in section 7.4, it is easy to improve the generalization of a classifier by increasing the size of the training set by adding extra copies of the training examples that have been modified with transformations that do not change the class. Object recognition is a classification task that is especially amenable to this form of dataset augmentation because the class is invariant to so many transformations and the input can be easily transformed with many geometric operations. As described before, classifiers can benefit from random translations, rotations, and in some cases, flips of the input to augment the dataset. In specialized computer vision applications, more advanced transformations are commonly used for dataset augmentation. These schemes include random perturbation of the colors in an image (Krizhevsky *et al.*, 2012) and nonlinear geometric distortions of the input (LeCun *et al.*, 1998b).

## 12.3 Speech Recognition

The task of speech recognition is to map an acoustic signal containing a spoken natural language utterance into the corresponding sequence of words intended by the speaker. Let  $\mathbf{X} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)})$  denote the sequence of acoustic input vectors (traditionally produced by splitting the audio into 20ms frames). Most speech recognition systems preprocess the input using specialized hand-designed features, but some (Jaitly and Hinton, 2011) deep learning systems learn features from raw input. Let  $\mathbf{y} = (y_1, y_2, \dots, y_N)$  denote the target output sequence (usually a sequence of words or characters). The **automatic speech recognition** (ASR) task consists of creating a function  $f_{\text{ASR}}^*$  that computes the most probable linguistic sequence  $\mathbf{y}$  given the acoustic sequence  $\mathbf{X}$ :

$$f_{\text{ASR}}^*(\mathbf{X}) = \arg \max_{\mathbf{y}} P^*(\mathbf{y} \mid \mathbf{X} = \mathbf{X}) \quad (12.4)$$

where  $P^*$  is the true conditional distribution relating the inputs  $\mathbf{X}$  to the targets  $\mathbf{y}$ .

Since the 1980s and until about 2009–2012, state-of-the art speech recognition systems primarily combined hidden Markov models (HMMs) and Gaussian mixture models (GMMs). GMMs modeled the association between acoustic features and phonemes (Bahl *et al.*, 1987), while HMMs modeled the sequence of phonemes. The GMM-HMM model family treats acoustic waveforms as being generated by the following process: first an HMM generates a sequence of phonemes and discrete sub-phonemic states (such as the beginning, middle, and end of each



phoneme), then a GMM transforms each discrete symbol into a brief segment of audio waveform. Although GMM-HMM systems dominated ASR until recently, speech recognition was actually one of the first areas where neural networks were applied, and numerous ASR systems from the late 1980s and early 1990s used neural nets (Bourlard and Wellekens, 1989; Waibel *et al.*, 1989; Robinson and Fallside, 1991; Bengio *et al.*, 1991, 1992; Konig *et al.*, 1996). At the time, the performance of ASR based on neural nets approximately matched the performance of GMM-HMM systems. For example, Robinson and Fallside (1991) achieved 26% phoneme error rate on the TIMIT (Garofolo *et al.*, 1993) corpus (with 39 phonemes to discriminate between), which was better than or comparable to HMM-based systems. Since then, TIMIT has been a benchmark for phoneme recognition, playing a role similar to the role MNIST plays for object recognition. However, because of the complex engineering involved in software systems for speech recognition and the effort that had been invested in building these systems on the basis of GMM-HMMs, the industry did not see a compelling argument for switching to neural networks. As a consequence, until the late 2000s, both academic and industrial research in using neural nets for speech recognition mostly focused on using neural nets to learn extra features for GMM-HMM systems.

Later, with *much larger and deeper models* and much larger datasets, recognition accuracy was dramatically improved by using neural networks to replace GMMs for the task of associating acoustic features to phonemes (or sub-phonemic states). Starting in 2009, speech researchers applied a form of deep learning based on unsupervised learning to speech recognition. This approach to deep learning was based on training undirected probabilistic models called restricted Boltzmann machines (RBMs) to model the input data. RBMs will be described in part III.

To solve speech recognition tasks, unsupervised pretraining was used to build deep feedforward networks whose layers were each initialized by training an RBM. These networks take spectral acoustic representations in a fixed-size input window (around a center frame) and predict the conditional probabilities of HMM states for that center frame. Training such deep networks helped to significantly improve the recognition rate on TIMIT (Mohamed *et al.*, 2009, 2012a), bringing down the phoneme error rate from about 26% to 20.7%. See Mohamed *et al.* (2012b) for an analysis of reasons for the success of these models. Extensions to the basic phone recognition pipeline included the addition of speaker-adaptive features (Mohamed *et al.*, 2011) that further reduced the error rate. This was quickly followed up by work to expand the architecture from phoneme recognition (which is what TIMIT is focused on) to large-vocabulary speech recognition (Dahl *et al.*, 2012), which involves not just recognizing phonemes but also recognizing sequences of words from a large vocabulary. Deep networks for speech recognition eventually

shifted from being based on pretraining and Boltzmann machines to being based on techniques such as rectified linear units and dropout (Zeiler *et al.*, 2013; Dahl *et al.*, 2013). By that time, several of the major speech groups in industry had started exploring deep learning in collaboration with academic researchers. Hinton *et al.* (2012a) describe the breakthroughs achieved by these collaborators, which are now deployed in products such as mobile phones.

Later, as these groups explored larger and larger labeled datasets and incorporated some of the methods for initializing, training, and setting up the architecture of deep nets, they realized that the unsupervised pretraining phase was either unnecessary or did not bring any significant improvement.

These breakthroughs in recognition performance for word error rate in speech recognition were unprecedented (around 30% improvement) and were following a long period of about ten years during which error rates did not improve much with the traditional GMM-HMM technology, in spite of the continuously growing size of training sets (see figure 2.4 of Deng and Yu (2014)). This created a rapid shift in the speech recognition community towards deep learning. In a matter of roughly two years, most of the industrial products for speech recognition incorporated deep neural networks and this success spurred a new wave of research into deep learning algorithms and architectures for ASR, which is still ongoing today.

One of these innovations was the use of convolutional networks (Sainath *et al.*, 2013) that replicate weights across time and frequency, improving over the earlier time-delay neural networks that replicated weights only across time. The new two-dimensional convolutional models regard the input spectrogram not as one long vector but as an image, with one axis corresponding to time and the other to frequency of spectral components.

Another important push, still ongoing, has been towards end-to-end deep learning speech recognition systems that completely remove the HMM. The first major breakthrough in this direction came from Graves *et al.* (2013) who trained a deep LSTM RNN (see section 10.10), using MAP inference over the frame-to-phoneme alignment, as in LeCun *et al.* (1998b) and in the CTC framework (Graves *et al.*, 2006; Graves, 2012). A deep RNN (Graves *et al.*, 2013) has state variables from several layers at each time step, giving the unfolded graph two kinds of depth: ordinary depth due to a stack of layers, and depth due to time unfolding. This work brought the phoneme error rate on TIMIT to a record low of 17.7%. See Pascanu *et al.* (2014a) and Chung *et al.* (2014) for other variants of deep RNNs, applied in other settings.

Another contemporary step toward end-to-end deep learning ASR is to let the system learn how to “align” the acoustic-level information with the phonetic-level



information (Chorowski *et al.*, 2014; Lu *et al.*, 2015).

## 12.4 Natural Language Processing

**Natural language processing** (NLP) is the use of human languages, such as English or French, by a computer. Computer programs typically read and emit specialized languages designed to allow efficient and unambiguous parsing by simple programs. More naturally occurring languages are often ambiguous and defy formal description. Natural language processing includes applications such as machine translation, in which the learner must read a sentence in one human language and emit an equivalent sentence in another human language. Many NLP applications are based on language models that define a probability distribution over sequences of words, characters or bytes in a natural language.

As with the other applications discussed in this chapter, very generic neural network techniques can be successfully applied to natural language processing. However, to achieve excellent performance and to scale well to large applications, some domain-specific strategies become important. To build an efficient model of natural language, we must usually use techniques that are specialized for processing sequential data. In many cases, we choose to regard natural language as a sequence of words, rather than a sequence of individual characters or bytes. Because the total number of possible words is so large, word-based language models must operate on an extremely high-dimensional and sparse discrete space. Several strategies have been developed to make models of such a space efficient, both in a computational and in a statistical sense.

### 12.4.1 $n$ -grams

A **language model** defines a probability distribution over sequences of tokens in a natural language. Depending on how the model is designed, a token may be a word, a character, or even a byte. Tokens are always discrete entities. The earliest successful language models were based on models of fixed-length sequences of tokens called  $n$ -grams. An  $n$ -gram is a sequence of  $n$  tokens.

Models based on  $n$ -grams define the conditional probability of the  $n$ -th token given the preceding  $n - 1$  tokens. The model uses products of these conditional distributions to define the probability distribution over longer sequences:

$$P(x_1, \dots, x_\tau) = P(x_1, \dots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t \mid x_{t-n+1}, \dots, x_{t-1}). \quad (12.5)$$

This decomposition is justified by the chain rule of probability. The probability distribution over the initial sequence  $P(x_1, \dots, x_{n-1})$  may be modeled by a different model with a smaller value of  $n$ .

Training  $n$ -gram models is straightforward because the maximum likelihood estimate can be computed simply by counting how many times each possible  $n$  gram occurs in the training set. Models based on  $n$ -grams have been the core building block of statistical language modeling for many decades (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999).

For small values of  $n$ , models have particular names: **unigram** for  $n=1$ , **bigram** for  $n=2$ , and **trigram** for  $n=3$ . These names derive from the Latin prefixes for the corresponding numbers and the Greek suffix “-gram” denoting something that is written.

Usually we train both an  $n$ -gram model and an  $n-1$  gram model simultaneously. This makes it easy to compute

$$P(x_t \mid x_{t-n+1}, \dots, x_{t-1}) = \frac{P_n(x_{t-n+1}, \dots, x_t)}{P_{n-1}(x_{t-n+1}, \dots, x_{t-1})} \quad (12.6)$$

simply by looking up two stored probabilities. For this to exactly reproduce inference in  $P_n$ , we must omit the final character from each sequence when we train  $P_{n-1}$ .

As an example, we demonstrate how a trigram model computes the probability of the sentence “THE DOG RAN AWAY.” The first words of the sentence cannot be handled by the default formula based on conditional probability because there is no context at the beginning of the sentence. Instead, we must use the marginal probability over words at the start of the sentence. We thus evaluate  $P_3(\text{THE DOG RAN})$ . Finally, the last word may be predicted using the typical case, of using the conditional distribution  $P(\text{AWAY} \mid \text{DOG RAN})$ . Putting this together with equation 12.6, we obtain:

$$P(\text{THE DOG RAN AWAY}) = P_3(\text{THE DOG RAN})P_3(\text{DOG RAN AWAY})/P_2(\text{DOG RAN}). \quad (12.7)$$

A fundamental limitation of maximum likelihood for  $n$ -gram models is that  $P_n$  as estimated from training set counts is very likely to be zero in many cases, even though the tuple  $(x_{t-n+1}, \dots, x_t)$  may appear in the test set. This can cause two different kinds of catastrophic outcomes. When  $P_{n-1}$  is zero, the ratio is undefined, so the model does not even produce a sensible output. When  $P_{n-1}$  is non-zero but  $P_n$  is zero, the test log-likelihood is  $-\infty$ . To avoid such catastrophic outcomes, most  $n$ -gram models employ some form of **smoothing**. Smoothing techniques

shift probability mass from the observed tuples to unobserved ones that are similar. See [Chen and Goodman \(1999\)](#) for a review and empirical comparisons. One basic technique consists of adding non-zero probability mass to all of the possible next symbol values. This method can be justified as Bayesian inference with a uniform or Dirichlet prior over the count parameters. Another very popular idea is to form a mixture model containing higher-order and lower-order  $n$ -gram models, with the higher-order models providing more capacity and the lower-order models being more likely to avoid counts of zero. **Back-off methods** look-up the lower-order  $n$ -grams if the frequency of the context  $x_{t-1}, \dots, x_{t-n+1}$  is too small to use the higher-order model. More formally, they estimate the distribution over  $x_t$  by using contexts  $x_{t-n+k}, \dots, x_{t-1}$ , for increasing  $k$ , until a sufficiently reliable estimate is found.

Classical  $n$ -gram models are particularly vulnerable to the curse of dimensionality. There are  $|\mathbb{V}|^n$  possible  $n$ -grams and  $|\mathbb{V}|$  is often very large. Even with a massive training set and modest  $n$ , most  $n$ -grams will not occur in the training set. One way to view a classical  $n$ -gram model is that it is performing nearest-neighbor lookup. In other words, it can be viewed as a local non-parametric predictor, similar to  $k$ -nearest neighbors. The statistical problems facing these extremely local predictors are described in section 5.11.2. The problem for a language model is even more severe than usual, because any two different words have the same distance from each other in one-hot vector space. It is thus difficult to leverage much information from any “neighbors”—only training examples that repeat literally the same context are useful for local generalization. To overcome these problems, a language model must be able to share knowledge between one word and other semantically similar words.

To improve the statistical efficiency of  $n$ -gram models, **class-based language models** ([Brown \*et al.\*, 1992](#); [Ney and Kneser, 1993](#); [Niesler \*et al.\*, 1998](#)) introduce the notion of word categories and then share statistical strength between words that are in the same category. The idea is to use a clustering algorithm to partition the set of words into clusters or classes, based on their co-occurrence frequencies with other words. The model can then use word class IDs rather than individual word IDs to represent the context on the right side of the conditioning bar. Composite models combining word-based and class-based models via mixing or back-off are also possible. Although word classes provide a way to generalize between sequences in which some word is replaced by another of the same class, much information is lost in this representation.

## 12.4.2 Neural Language Models

**Neural language models** or NLMs are a class of language model designed to overcome the curse of dimensionality problem for modeling natural language sequences by using a distributed representation of words (Bengio *et al.*, 2001). Unlike class-based  $n$ -gram models, neural language models are able to recognize that two words are similar without losing the ability to encode each word as distinct from the other. Neural language models share statistical strength between one word (and its context) and other similar words and contexts. The distributed representation the model learns for each word enables this sharing by allowing the model to treat words that have features in common similarly. For example, if the word `dog` and the word `cat` map to representations that share many attributes, then sentences that contain the word `cat` can inform the predictions that will be made by the model for sentences that contain the word `dog`, and vice-versa. Because there are many such attributes, there are many ways in which generalization can happen, transferring information from each training sentence to an exponentially large number of semantically related sentences. The curse of dimensionality requires the model to generalize to a number of sentences that is exponential in the sentence length. The model counters this curse by relating each training sentence to an exponential number of similar sentences.

We sometimes call these word representations **word embeddings**. In this interpretation, we view the raw symbols as points in a space of dimension equal to the vocabulary size. The word representations embed those points in a feature space of lower dimension. In the original space, every word is represented by a one-hot vector, so every pair of words is at Euclidean distance  $\sqrt{2}$  from each other. In the embedding space, words that frequently appear in similar contexts (or any pair of words sharing some “features” learned by the model) are close to each other. This often results in words with similar meanings being neighbors. Figure 12.3 zooms in on specific areas of a learned word embedding space to show how semantically similar words map to representations that are close to each other.

Neural networks in other domains also define embeddings. For example, a hidden layer of a convolutional network provides an “image embedding.” Usually NLP practitioners are much more interested in this idea of embeddings because natural language does not originally lie in a real-valued vector space. The hidden layer has provided a more qualitatively dramatic change in the way the data is represented.

The basic idea of using distributed representations to improve models for natural language processing is not restricted to neural networks. It may also be used with graphical models that have distributed representations in the form of

multiple latent variables (Mnih and Hinton, 2007).

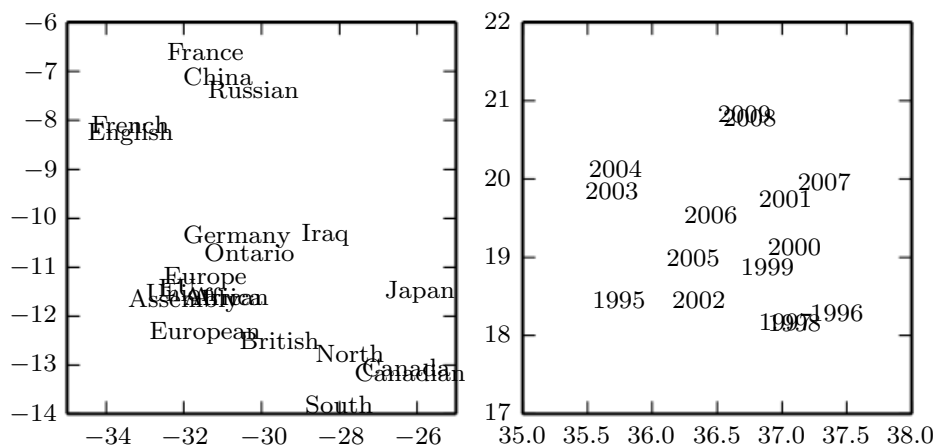


Figure 12.3: Two-dimensional visualizations of word embeddings obtained from a neural machine translation model (Bahdanau *et al.*, 2015), zooming in on specific areas where semantically related words have embedding vectors that are close to each other. Countries appear on the left and numbers on the right. Keep in mind that these embeddings are 2-D for the purpose of visualization. In real applications, embeddings typically have higher dimensionality and can simultaneously capture many kinds of similarity between words.

### 12.4.3 High-Dimensional Outputs

In many natural language applications, we often want our models to produce words (rather than characters) as the fundamental unit of the output. For large vocabularies, it can be very computationally expensive to represent an output distribution over the choice of a word, because the vocabulary size is large. In many applications,  $\mathbb{V}$  contains hundreds of thousands of words. The naive approach to representing such a distribution is to apply an affine transformation from a hidden representation to the output space, then apply the softmax function. Suppose we have a vocabulary  $\mathbb{V}$  with size  $|\mathbb{V}|$ . The weight matrix describing the linear component of this affine transformation is very large, because its output dimension is  $|\mathbb{V}|$ . This imposes a high memory cost to represent the matrix, and a high computational cost to multiply by it. Because the softmax is normalized across all  $|\mathbb{V}|$  outputs, it is necessary to perform the full matrix multiplication at training time as well as test time—we cannot calculate only the dot product with the weight vector for the correct output. The high computational costs of the output layer thus arise both at training time (to compute the likelihood and its gradient) and at test time (to compute probabilities for all or selected words). For specialized

loss functions, the gradient can be computed efficiently (Vincent *et al.*, 2015), but the standard cross-entropy loss applied to a traditional softmax output layer poses many difficulties.

Suppose that  $\mathbf{h}$  is the top hidden layer used to predict the output probabilities  $\hat{\mathbf{y}}$ . If we parametrize the transformation from  $\mathbf{h}$  to  $\hat{\mathbf{y}}$  with learned weights  $\mathbf{W}$  and learned biases  $\mathbf{b}$ , then the affine-softmax output layer performs the following computations:

$$a_i = b_i + \sum_j W_{ij} h_j \quad \forall i \in \{1, \dots, |\mathbb{V}|\}, \quad (12.8)$$

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{i'=1}^{|\mathbb{V}|} e^{a_{i'}}}. \quad (12.9)$$

If  $\mathbf{h}$  contains  $n_h$  elements then the above operation is  $O(|\mathbb{V}|n_h)$ . With  $n_h$  in the thousands and  $|\mathbb{V}|$  in the hundreds of thousands, this operation dominates the computation of most neural language models.

### 12.4.3.1 Use of a Short List

The first neural language models (Bengio *et al.*, 2001, 2003) dealt with the high cost of using a softmax over a large number of output words by limiting the vocabulary size to 10,000 or 20,000 words. Schwenk and Gauvain (2002) and Schwenk (2007) built upon this approach by splitting the vocabulary  $\mathbb{V}$  into a **shortlist**  $\mathbb{L}$  of most frequent words (handled by the neural net) and a tail  $\mathbb{T} = \mathbb{V} \setminus \mathbb{L}$  of more rare words (handled by an  $n$ -gram model). To be able to combine the two predictions, the neural net also has to predict the probability that a word appearing after context  $C$  belongs to the tail list. This may be achieved by adding an extra sigmoid output unit to provide an estimate of  $P(i \in \mathbb{T} \mid C)$ . The extra output can then be used to achieve an estimate of the probability distribution over all words in  $\mathbb{V}$  as follows:

$$\begin{aligned} P(y = i \mid C) = & 1_{i \in \mathbb{L}} P(y = i \mid C, i \in \mathbb{L}) (1 - P(i \in \mathbb{T} \mid C)) \\ & + 1_{i \in \mathbb{T}} P(y = i \mid C, i \in \mathbb{T}) P(i \in \mathbb{T} \mid C) \end{aligned} \quad (12.10)$$

where  $P(y = i \mid C, i \in \mathbb{L})$  is provided by the neural language model and  $P(y = i \mid C, i \in \mathbb{T})$  is provided by the  $n$ -gram model. With slight modification, this approach can also work using an extra output value in the neural language model's softmax layer, rather than a separate sigmoid unit.

An obvious disadvantage of the short list approach is that the potential generalization advantage of the neural language models is limited to the most frequent

words, where, arguably, it is the least useful. This disadvantage has stimulated the exploration of alternative methods to deal with high-dimensional outputs, described below.

### 12.4.3.2 Hierarchical Softmax

A classical approach (Goodman, 2001) to reducing the computational burden of high-dimensional output layers over large vocabulary sets  $\mathbb{V}$  is to decompose probabilities hierarchically. Instead of necessitating a number of computations proportional to  $|\mathbb{V}|$  (and also proportional to the number of hidden units,  $n_h$ ), the  $|\mathbb{V}|$  factor can be reduced to as low as  $\log |\mathbb{V}|$ . Bengio (2002) and Morin and Bengio (2005) introduced this factorized approach to the context of neural language models.

One can think of this hierarchy as building categories of words, then categories of categories of words, then categories of categories of categories of words, etc. These nested categories form a tree, with words at the leaves. In a balanced tree, the tree has depth  $O(\log |\mathbb{V}|)$ . The probability of a choosing a word is given by the product of the probabilities of choosing the branch leading to that word at every node on a path from the root of the tree to the leaf containing the word. Figure 12.4 illustrates a simple example. Mnih and Hinton (2009) also describe how to use multiple paths to identify a single word in order to better model words that have multiple meanings. Computing the probability of a word then involves summation over all of the paths that lead to that word.

To predict the conditional probabilities required at each node of the tree, we typically use a logistic regression model at each node of the tree, and provide the same context  $C$  as input to all of these models. Because the correct output is encoded in the training set, we can use supervised learning to train the logistic regression models. This is typically done using a standard cross-entropy loss, corresponding to maximizing the log-likelihood of the correct sequence of decisions.

Because the output log-likelihood can be computed efficiently (as low as  $\log |\mathbb{V}|$  rather than  $|\mathbb{V}|$ ), its gradients may also be computed efficiently. This includes not only the gradient with respect to the output parameters but also the gradients with respect to the hidden layer activations.

It is possible but usually not practical to optimize the tree structure to minimize the expected number of computations. Tools from information theory specify how to choose the optimal binary code given the relative frequencies of the words. To do so, we could structure the tree so that the number of bits associated with a word is approximately equal to the logarithm of the frequency of that word. However, in



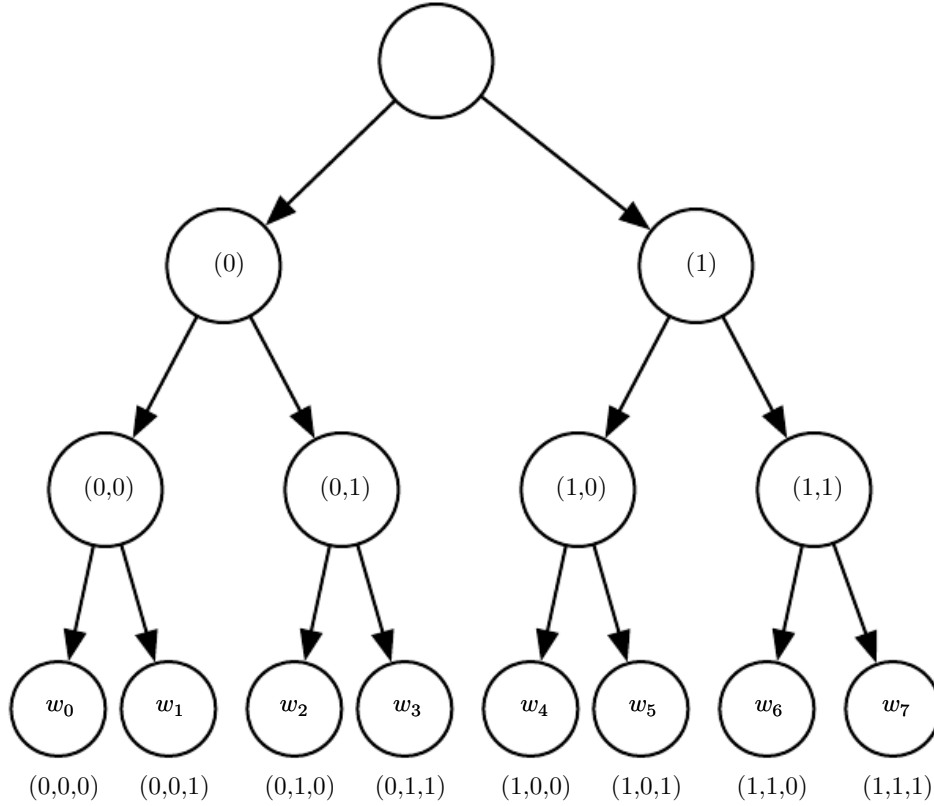


Figure 12.4: Illustration of a simple hierarchy of word categories, with 8 words  $w_0, \dots, w_7$  organized into a three level hierarchy. The leaves of the tree represent actual specific words. Internal nodes represent groups of words. Any node can be indexed by the sequence of binary decisions (0=left, 1=right) to reach the node from the root. Super-class (0) contains the classes (0, 0) and (0, 1), which respectively contain the sets of words  $\{w_0, w_1\}$  and  $\{w_2, w_3\}$ , and similarly super-class (1) contains the classes (1, 0) and (1, 1), which respectively contain the words  $(w_4, w_5)$  and  $(w_6, w_7)$ . If the tree is sufficiently balanced, the maximum depth (number of binary decisions) is on the order of the logarithm of the number of words  $|\mathbb{V}|$ : the choice of one out of  $|\mathbb{V}|$  words can be obtained by doing  $O(\log |\mathbb{V}|)$  operations (one for each of the nodes on the path from the root). In this example, computing the probability of a word  $y$  can be done by multiplying three probabilities, associated with the binary decisions to move left or right at each node on the path from the root to a node  $y$ . Let  $b_i(y)$  be the  $i$ -th binary decision when traversing the tree towards the value  $y$ . The probability of sampling an output  $y$  decomposes into a product of conditional probabilities, using the chain rule for conditional probabilities, with each node indexed by the prefix of these bits. For example, node (1, 0) corresponds to the prefix  $(b_0(w_4) = 1, b_1(w_4) = 0)$ , and the probability of  $w_4$  can be decomposed as follows:

$$P(y = w_4) = P(b_0 = 1, b_1 = 0, b_2 = 0) \quad (12.11)$$

$$= P(b_0 = 1)P(b_1 = 0 \mid b_0 = 1)P(b_2 = 0 \mid b_0 = 1, b_1 = 0). \quad (12.12)$$



practice, the computational savings are typically not worth the effort because the computation of the output probabilities is only one part of the total computation in the neural language model. For example, suppose there are  $l$  fully connected hidden layers of width  $n_h$ . Let  $n_b$  be the weighted average of the number of bits required to identify a word, with the weighting given by the frequency of these words. In this example, the number of operations needed to compute the hidden activations grows as  $O(ln_h^2)$  while the output computations grow as  $O(n_h n_b)$ . As long as  $n_b \leq ln_h$ , we can reduce computation more by shrinking  $n_h$  than by shrinking  $n_b$ . Indeed,  $n_b$  is often small. Because the size of the vocabulary rarely exceeds a million words and  $\log_2(10^6) \approx 20$ , it is possible to reduce  $n_b$  to about 20, but  $n_h$  is often much larger, around  $10^3$  or more. Rather than carefully optimizing a tree with a branching factor of 2, one can instead define a tree with depth two and a branching factor of  $\sqrt{|\mathbb{V}|}$ . Such a tree corresponds to simply defining a set of mutually exclusive word classes. The simple approach based on a tree of depth two captures most of the computational benefit of the hierarchical strategy.

One question that remains somewhat open is how to best define these word classes, or how to define the word hierarchy in general. Early work used existing hierarchies (Morin and Bengio, 2005) but the hierarchy can also be learned, ideally jointly with the neural language model. Learning the hierarchy is difficult. An exact optimization of the log-likelihood appears intractable because the choice of a word hierarchy is a discrete one, not amenable to gradient-based optimization. However, one could use discrete optimization to approximately optimize the partition of words into word classes.

An important advantage of the hierarchical softmax is that it brings computational benefits both at training time and at test time, if at test time we want to compute the probability of specific words.

Of course, computing the probability of all  $|\mathbb{V}|$  words will remain expensive even with the hierarchical softmax. Another important operation is selecting the most likely word in a given context. Unfortunately the tree structure does not provide an efficient and exact solution to this problem.

A disadvantage is that in practice the hierarchical softmax tends to give worse test results than sampling-based methods we will describe next. This may be due to a poor choice of word classes.

### 12.4.3.3 Importance Sampling

One way to speed up the training of neural language models is to avoid explicitly computing the contribution of the gradient from all of the words that do not appear

in the next position. Every incorrect word should have low probability under the model. It can be computationally costly to enumerate all of these words. Instead, it is possible to sample only a subset of the words. Using the notation introduced in equation 12.8, the gradient can be written as follows:

$$\frac{\partial \log P(y | C)}{\partial \theta} = \frac{\partial \log \text{softmax}_y(\mathbf{a})}{\partial \theta} \quad (12.13)$$

$$= \frac{\partial}{\partial \theta} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \quad (12.14)$$

$$= \frac{\partial}{\partial \theta} (a_y - \log \sum_i e^{a_i}) \quad (12.15)$$

$$= \frac{\partial a_y}{\partial \theta} - \sum_i P(y = i | C) \frac{\partial a_i}{\partial \theta} \quad (12.16)$$

where  $\mathbf{a}$  is the vector of pre-softmax activations (or scores), with one element per word. The first term is the **positive phase** term (pushing  $a_y$  up) while the second term is the **negative phase** term (pushing  $a_i$  down for all  $i$ , with weight  $P(i | C)$ ). Since the negative phase term is an expectation, we can estimate it with a Monte Carlo sample. However, that would require sampling from the model itself. Sampling from the model requires computing  $P(i | C)$  for all  $i$  in the vocabulary, which is precisely what we are trying to avoid.

Instead of sampling from the model, one can sample from another distribution, called the proposal distribution (denoted  $q$ ), and use appropriate weights to correct for the bias introduced by sampling from the wrong distribution (Bengio and S  n  cal, 2003; Bengio and S  n  cal, 2008). This is an application of a more general technique called **importance sampling**, which will be described in more detail in section 17.2. Unfortunately, even exact importance sampling is not efficient because it requires computing weights  $p_i/q_i$ , where  $p_i = P(i | C)$ , which can only be computed if all the scores  $a_i$  are computed. The solution adopted for this application is called **biased importance sampling**, where the importance weights are normalized to sum to 1. When negative word  $n_i$  is sampled, the associated gradient is weighted by

$$w_i = \frac{p_{n_i}/q_{n_i}}{\sum_{j=1}^N p_{n_j}/q_{n_j}}. \quad (12.17)$$

These weights are used to give the appropriate importance to the  $m$  negative samples from  $q$  used to form the estimated negative phase contribution to the

gradient:

$$\sum_{i=1}^{|\mathbb{V}|} P(i \mid C) \frac{\partial a_i}{\partial \theta} \approx \frac{1}{m} \sum_{i=1}^m w_i \frac{\partial a_{n_i}}{\partial \theta}. \quad (12.18)$$

A unigram or a bigram distribution works well as the proposal distribution  $q$ . It is easy to estimate the parameters of such a distribution from data. After estimating the parameters, it is also possible to sample from such a distribution very efficiently.

Importance sampling is not only useful for speeding up models with large softmax outputs. More generally, it is useful for accelerating training with large sparse output layers, where the output is a sparse vector rather than a 1-of- $n$  choice. An example is a **bag of words**. A bag of words is a sparse vector  $\mathbf{v}$  where  $v_i$  indicates the presence or absence of word  $i$  from the vocabulary in the document. Alternately,  $v_i$  can indicate the number of times that word  $i$  appears. Machine learning models that emit such sparse vectors can be expensive to train for a variety of reasons. Early in learning, the model may not actually choose to make the output truly sparse. Moreover, the loss function we use for training might most naturally be described in terms of comparing every element of the output to every element of the target. This means that it is not always clear that there is a computational benefit to using sparse outputs, because the model may choose to make the majority of the output non-zero and all of these non-zero values need to be compared to the corresponding training target, even if the training target is zero. [Dauphin et al. \(2011\)](#) demonstrated that such models can be accelerated using importance sampling. The efficient algorithm minimizes the loss reconstruction for the “positive words” (those that are non-zero in the target) and an equal number of “negative words.” The negative words are chosen randomly, using a heuristic to sample words that are more likely to be mistaken. The bias introduced by this heuristic oversampling can then be corrected using importance weights.

In all of these cases, the computational complexity of gradient estimation for the output layer is reduced to be proportional to the number of negative samples rather than proportional to the size of the output vector.

#### 12.4.3.4 Noise-Contrastive Estimation and Ranking Loss

Other approaches based on sampling have been proposed to reduce the computational cost of training neural language models with large vocabularies. An early example is the ranking loss proposed by [Collobert and Weston \(2008a\)](#), which views the output of the neural language model for each word as a score and tries to make the score of the correct word  $a_y$  be ranked high in comparison to the other

scores  $a_i$ . The ranking loss proposed then is

$$L = \sum_i \max(0, 1 - a_y + a_i). \quad (12.19)$$

The gradient is zero for the  $i$ -th term if the score of the observed word,  $a_y$ , is greater than the score of the negative word  $a_i$  by a margin of 1. One issue with this criterion is that it does not provide estimated conditional probabilities, which are useful in some applications, including speech recognition and text generation (including conditional text generation tasks such as translation).

A more recently used training objective for neural language model is noise-contrastive estimation, which is introduced in section 18.6. This approach has been successfully applied to neural language models (Mnih and Teh, 2012; Mnih and Kavukcuoglu, 2013).

#### 12.4.4 Combining Neural Language Models with $n$ -grams

A major advantage of  $n$ -gram models over neural networks is that  $n$ -gram models achieve high model capacity (by storing the frequencies of very many tuples) while requiring very little computation to process an example (by looking up only a few tuples that match the current context). If we use hash tables or trees to access the counts, the computation used for  $n$ -grams is almost independent of capacity. In comparison, doubling a neural network's number of parameters typically also roughly doubles its computation time. Exceptions include models that avoid using all parameters on each pass. Embedding layers index only a single embedding in each pass, so we can increase the vocabulary size without increasing the computation time per example. Some other models, such as tiled convolutional networks, can add parameters while reducing the degree of parameter sharing in order to maintain the same amount of computation. However, typical neural network layers based on matrix multiplication use an amount of computation proportional to the number of parameters.

One easy way to add capacity is thus to combine both approaches in an ensemble consisting of a neural language model and an  $n$ -gram language model (Bengio *et al.*, 2001, 2003). As with any ensemble, this technique can reduce test error if the ensemble members make independent mistakes. The field of ensemble learning provides many ways of combining the ensemble members' predictions, including uniform weighting and weights chosen on a validation set. Mikolov *et al.* (2011a) extended the ensemble to include not just two models but a large array of models. It is also possible to pair a neural network with a maximum entropy model and train both jointly (Mikolov *et al.*, 2011b). This approach can be viewed as training

a neural network with an extra set of inputs that are connected directly to the output, and not connected to any other part of the model. The extra inputs are indicators for the presence of particular  $n$ -grams in the input context, so these variables are very high-dimensional and very sparse. The increase in model capacity is huge—the new portion of the architecture contains up to  $|sV|^n$  parameters—but the amount of added computation needed to process an input is minimal because the extra inputs are very sparse.

### 12.4.5 Neural Machine Translation

Machine translation is the task of reading a sentence in one natural language and emitting a sentence with the equivalent meaning in another language. Machine translation systems often involve many components. At a high level, there is often one component that proposes many candidate translations. Many of these translations will not be grammatical due to differences between the languages. For example, many languages put adjectives after nouns, so when translated to English directly they yield phrases such as “apple red.” The proposal mechanism suggests many variants of the suggested translation, ideally including “red apple.” A second component of the translation system, a language model, evaluates the proposed translations, and can score “red apple” as better than “apple red.”

The earliest use of neural networks for machine translation was to upgrade the language model of a translation system by using a neural language model (Schwenk *et al.*, 2006; Schwenk, 2010). Previously, most machine translation systems had used an  $n$ -gram model for this component. The  $n$ -gram based models used for machine translation include not just traditional back-off  $n$ -gram models (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999) but also **maximum entropy language models** (Berger *et al.*, 1996), in which an affine-softmax layer predicts the next word given the presence of frequent  $n$ -grams in the context.

Traditional language models simply report the probability of a natural language sentence. Because machine translation involves producing an output sentence given an input sentence, it makes sense to extend the natural language model to be conditional. As described in section 6.2.1.1, it is straightforward to extend a model that defines a marginal distribution over some variable to define a conditional distribution over that variable given a context  $C$ , where  $C$  might be a single variable or a list of variables. Devlin *et al.* (2014) beat the state-of-the-art in some statistical machine translation benchmarks by using an MLP to score a phrase  $t_1, t_2, \dots, t_k$  in the target language given a phrase  $s_1, s_2, \dots, s_n$  in the source language. The MLP estimates  $P(t_1, t_2, \dots, t_k \mid s_1, s_2, \dots, s_n)$ . The estimate formed by this MLP replaces the estimate provided by conditional  $n$ -gram models.

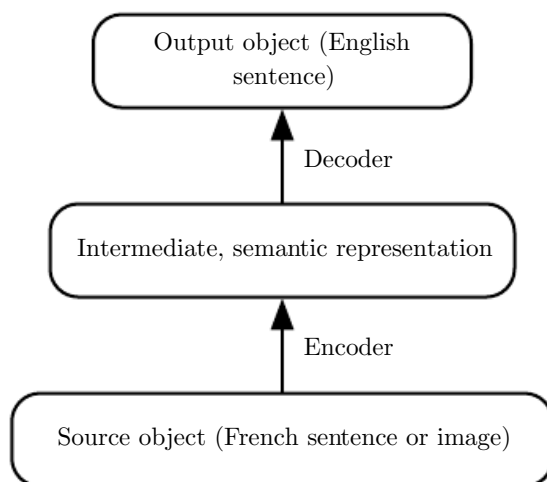


Figure 12.5: The encoder-decoder architecture to map back and forth between a surface representation (such as a sequence of words or an image) and a semantic representation. By using the output of an encoder of data from one modality (such as the encoder mapping from French sentences to hidden representations capturing the meaning of sentences) as the input to a decoder for another modality (such as the decoder mapping from hidden representations capturing the meaning of sentences to English), we can train systems that translate from one modality to another. This idea has been applied successfully not just to machine translation but also to caption generation from images.

A drawback of the MLP-based approach is that it requires the sequences to be preprocessed to be of fixed length. To make the translation more flexible, we would like to use a model that can accommodate variable length inputs and variable length outputs. An RNN provides this ability. Section 10.2.4 describes several ways of constructing an RNN that represents a conditional distribution over a sequence given some input, and section 10.4 describes how to accomplish this conditioning when the input is a sequence. In all cases, one model first reads the input sequence and emits a data structure that summarizes the input sequence. We call this summary the “context”  $C$ . The context  $C$  may be a list of vectors, or it may be a vector or tensor. The model that reads the input to produce  $C$  may be an RNN (Choi *et al.*, 2014a; Sutskever *et al.*, 2014; Jean *et al.*, 2014) or a convolutional network (Kalchbrenner and Blunsom, 2013). A second model, usually an RNN, then reads the context  $C$  and generates a sentence in the target language. This general idea of an encoder-decoder framework for machine translation is illustrated in figure 12.5.

In order to generate an entire sentence conditioned on the source sentence, the model must have a way to represent the entire source sentence. Earlier models were only able to represent individual words or phrases. From a representation

learning point of view, it can be useful to learn a representation in which sentences that have the same meaning have similar representations regardless of whether they were written in the source language or the target language. This strategy was explored first using a combination of convolutions and RNNs (Kalchbrenner and Blunsom, 2013). Later work introduced the use of an RNN for scoring proposed translations (Cho *et al.*, 2014a) and for generating translated sentences (Sutskever *et al.*, 2014). Jean *et al.* (2014) scaled these models to larger vocabularies.

#### 12.4.5.1 Using an Attention Mechanism and Aligning Pieces of Data

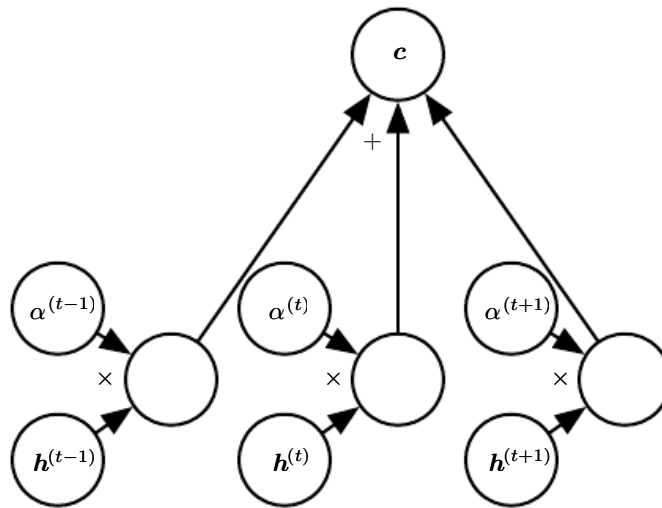


Figure 12.6: A modern attention mechanism, as introduced by Bahdanau *et al.* (2015), is essentially a weighted average. A context vector  $c$  is formed by taking a weighted average of feature vectors  $h^{(t)}$  with weights  $\alpha^{(t)}$ . In some applications, the feature vectors  $h$  are hidden units of a neural network, but they may also be raw input to the model. The weights  $\alpha^{(t)}$  are produced by the model itself. They are usually values in the interval  $[0, 1]$  and are intended to concentrate around just one  $h^{(t)}$  so that the weighted average approximates reading that one specific time step precisely. The weights  $\alpha^{(t)}$  are usually produced by applying a softmax function to relevance scores emitted by another portion of the model. The attention mechanism is more expensive computationally than directly indexing the desired  $h^{(t)}$ , but direct indexing cannot be trained with gradient descent. The attention mechanism based on weighted averages is a smooth, differentiable approximation that can be trained with existing optimization algorithms.

Using a fixed-size representation to capture all the semantic details of a very long sentence of say 60 words is very difficult. It can be achieved by training a sufficiently large RNN well enough and for long enough, as demonstrated by Cho *et al.* (2014a) and Sutskever *et al.* (2014). However, a more efficient approach is to read the whole sentence or paragraph (to get the context and the gist of what



is being expressed), then produce the translated words one at a time, each time focusing on a different part of the input sentence in order to gather the semantic details that are required to produce the next output word. That is exactly the idea that Bahdanau *et al.* (2015) first introduced. The attention mechanism used to focus on specific parts of the input sequence at each time step is illustrated in figure 12.6.

We can think of an attention-based system as having three components:

1. A process that “reads” raw data (such as source words in a source sentence), and converts them into distributed representations, with one feature vector associated with each word position.
2. A list of feature vectors storing the output of the reader. This can be understood as a “memory” containing a sequence of facts, which can be retrieved later, not necessarily in the same order, without having to visit all of them.
3. A process that “exploits” the content of the memory to sequentially perform a task, at each time step having the ability put attention on the content of one memory element (or a few, with a different weight).

The third component generates the translated sentence.

When words in a sentence written in one language are aligned with corresponding words in a translated sentence in another language, it becomes possible to relate the corresponding word embeddings. Earlier work showed that one could learn a kind of translation matrix relating the word embeddings in one language with the word embeddings in another (Kočiský *et al.*, 2014), yielding lower alignment error rates than traditional approaches based on the frequency counts in the phrase table. There is even earlier work on learning cross-lingual word vectors (Klementiev *et al.*, 2012). Many extensions to this approach are possible. For example, more efficient cross-lingual alignment (Gouws *et al.*, 2014) allows training on larger datasets.

### 12.4.6 Historical Perspective

The idea of distributed representations for symbols was introduced by Rumelhart *et al.* (1986a) in one of the first explorations of back-propagation, with symbols corresponding to the identity of family members and the neural network capturing the relationships between family members, with training examples forming triplets such as (Colin, Mother, Victoria). The first layer of the neural network learned a representation of each family member. For example, the features for Colin



might represent which family tree Colin was in, what branch of that tree he was in, what generation he was from, etc. One can think of the neural network as computing learned rules relating these attributes together in order to obtain the desired predictions. The model can then make predictions such as inferring who is the mother of Colin.

The idea of forming an embedding for a symbol was extended to the idea of an embedding for a word by [Deerwester \*et al.\* \(1990\)](#). These embeddings were learned using the SVD. Later, embeddings would be learned by neural networks.

The history of natural language processing is marked by transitions in the popularity of different ways of representing the input to the model. Following this early work on symbols or words, some of the earliest applications of neural networks to NLP ([Mäikkyläinen and Dyer, 1991](#); [Schmidhuber, 1996](#)) represented the input as a sequence of characters.

[Bengio \*et al.\* \(2001\)](#) returned the focus to modeling words and introduced neural language models, which produce interpretable word embeddings. These neural models have scaled up from defining representations of a small set of symbols in the 1980s to millions of words (including proper nouns and misspellings) in modern applications. This computational scaling effort led to the invention of the techniques described above in section [12.4.3](#).

Initially, the use of words as the fundamental units of language models yielded improved language modeling performance ([Bengio \*et al.\*, 2001](#)). To this day, new techniques continually push both character-based models ([Sutskever \*et al.\*, 2011](#)) and word-based models forward, with recent work ([Gillick \*et al.\*, 2015](#)) even modeling individual bytes of Unicode characters.

The ideas behind neural language models have been extended into several natural language processing applications, such as parsing ([Henderson, 2003, 2004](#); [Collobert, 2011](#)), part-of-speech tagging, semantic role labeling, chunking, etc, sometimes using a single multi-task learning architecture ([Collobert and Weston, 2008a](#); [Collobert \*et al.\*, 2011a](#)) in which the word embeddings are shared across tasks.

Two-dimensional visualizations of embeddings became a popular tool for analyzing language models following the development of the t-SNE dimensionality reduction algorithm ([van der Maaten and Hinton, 2008](#)) and its high-profile application to visualization word embeddings by Joseph Turian in 2009.

## 12.5 Other Applications

In this section we cover a few other types of applications of deep learning that are different from the standard object recognition, speech recognition and natural language processing tasks discussed above. Part III of this book will expand that scope even further to tasks that remain primarily research areas.

### 12.5.1 Recommender Systems

One of the major families of applications of machine learning in the information technology sector is the ability to make recommendations of items to potential users or customers. Two major types of applications can be distinguished: online advertising and item recommendations (often these recommendations are still for the purpose of selling a product). Both rely on predicting the association between a user and an item, either to predict the probability of some action (the user buying the product, or some proxy for this action) or the expected gain (which may depend on the value of the product) if an ad is shown or a recommendation is made regarding that product to that user. The internet is currently financed in great part by various forms of online advertising. There are major parts of the economy that rely on online shopping. Companies including Amazon and eBay use machine learning, including deep learning, for their product recommendations. Sometimes, the items are not products that are actually for sale. Examples include selecting posts to display on social network news feeds, recommending movies to watch, recommending jokes, recommending advice from experts, matching players for video games, or matching people in dating services.

Often, this association problem is handled like a supervised learning problem: given some information about the item and about the user, predict the proxy of interest (user clicks on ad, user enters a rating, user clicks on a “like” button, user buys product, user spends some amount of money on the product, user spends time visiting a page for the product, etc). This often ends up being either a regression problem (predicting some conditional expected value) or a probabilistic classification problem (predicting the conditional probability of some discrete event).

The early work on recommender systems relied on minimal information as inputs for these predictions: the user ID and the item ID. In this context, the only way to generalize is to rely on the similarity between the patterns of values of the target variable for different users or for different items. Suppose that user 1 and user 2 both like items A, B and C. From this, we may infer that user 1 and

user 2 have similar tastes. If user 1 likes item D, then this should be a strong cue that user 2 will also like D. Algorithms based on this principle come under the name of **collaborative filtering**. Both non-parametric approaches (such as nearest-neighbor methods based on the estimated similarity between patterns of preferences) and parametric methods are possible. Parametric methods often rely on learning a distributed representation (also called an embedding) for each user and for each item. Bilinear prediction of the target variable (such as a rating) is a simple parametric method that is highly successful and often found as a component of state-of-the-art systems. The prediction is obtained by the dot product between the user embedding and the item embedding (possibly corrected by constants that depend only on either the user ID or the item ID). Let  $\hat{\mathbf{R}}$  be the matrix containing our predictions,  $\mathbf{A}$  a matrix with user embeddings in its rows and  $\mathbf{B}$  a matrix with item embeddings in its columns. Let  $\mathbf{b}$  and  $\mathbf{c}$  be vectors that contain respectively a kind of bias for each user (representing how grumpy or positive that user is in general) and for each item (representing its general popularity). The bilinear prediction is thus obtained as follows:

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{u,j} B_{j,i}. \quad (12.20)$$

Typically one wants to minimize the squared error between predicted ratings  $\hat{R}_{u,i}$  and actual ratings  $R_{u,i}$ . User embeddings and item embeddings can then be conveniently visualized when they are first reduced to a low dimension (two or three), or they can be used to compare users or items against each other, just like word embeddings. One way to obtain these embeddings is by performing a singular value decomposition of the matrix  $\mathbf{R}$  of actual targets (such as ratings). This corresponds to factorizing  $\mathbf{R} = \mathbf{U}\mathbf{D}\mathbf{V}'$  (or a normalized variant) into the product of two factors, the lower rank matrices  $\mathbf{A} = \mathbf{U}\mathbf{D}$  and  $\mathbf{B} = \mathbf{V}'$ . One problem with the SVD is that it treats the missing entries in an arbitrary way, as if they corresponded to a target value of 0. Instead we would like to avoid paying any cost for the predictions made on missing entries. Fortunately, the sum of squared errors on the observed ratings can also be easily minimized by gradient-based optimization. The SVD and the bilinear prediction of equation 12.20 both performed very well in the competition for the Netflix prize (Bennett and Lanning, 2007), aiming at predicting ratings for films, based only on previous ratings by a large set of anonymous users. Many machine learning experts participated in this competition, which took place between 2006 and 2009. It raised the level of research in recommender systems using advanced machine learning and yielded improvements in recommender systems. Even though it did not win by itself, the simple bilinear prediction or SVD was a component of the ensemble models

presented by most of the competitors, including the winners (Töscher *et al.*, 2009; Koren, 2009).

Beyond these bilinear models with distributed representations, one of the first uses of neural networks for collaborative filtering is based on the RBM undirected probabilistic model (Salakhutdinov *et al.*, 2007). RBMs were an important element of the ensemble of methods that won the Netflix competition (Töscher *et al.*, 2009; Koren, 2009). More advanced variants on the idea of factorizing the ratings matrix have also been explored in the neural networks community (Salakhutdinov and Mnih, 2008).

However, there is a basic limitation of collaborative filtering systems: when a new item or a new user is introduced, its lack of rating history means that there is no way to evaluate its similarity with other items or users (respectively), or the degree of association between, say, that new user and existing items. This is called the problem of cold-start recommendations. A general way of solving the cold-start recommendation problem is to introduce extra information about the individual users and items. For example, this extra information could be user profile information or features of each item. Systems that use such information are called **content-based recommender systems**. The mapping from a rich set of user features or item features to an embedding can be learned through a deep learning architecture (Huang *et al.*, 2013; Elkahky *et al.*, 2015).

Specialized deep learning architectures such as convolutional networks have also been applied to learn to extract features from rich content such as from musical audio tracks, for music recommendation (van den Oörd *et al.*, 2013). In that work, the convolutional net takes acoustic features as input and computes an embedding for the associated song. The dot product between this song embedding and the embedding for a user is then used to predict whether a user will listen to the song.

### 12.5.1.1 Exploration Versus Exploitation

When making recommendations to users, an issue arises that goes beyond ordinary supervised learning and into the realm of reinforcement learning. Many recommendation problems are most accurately described theoretically as **contextual bandits** (Langford and Zhang, 2008; Lu *et al.*, 2010). The issue is that when we use the recommendation system to collect data, we get a biased and incomplete view of the preferences of users: we only see the responses of users to the items they were recommended and not to the other items. In addition, in some cases we may not get any information on users for whom no recommendation has been made (for example, with ad auctions, it may be that the price proposed for an