programming recursion will be complete because one can then use these values of $\frac{\partial L}{\partial a_j^r}$. One can compute the value of $\frac{\partial L}{\partial v_i^r}$ in terms of $\hat{v}_i^{(r)}$, μ_i, and σ_i, by observing that $v_i^{(r)}$ can be written as a (normalization) function of only $\hat{v}_i^{(r)}$, mean μ_i, and variance σ_i^2. Observe that μ_i and σ_i are not treated as constants, but as variables because they depend on the batch at hand. Therefore, we have the following:

$$\frac{\partial L}{\partial v_i^{(r)}} = \frac{\partial L}{\partial \hat{v}_i^{(r)}} \frac{\partial \hat{v}_i^{(r)}}{\partial v_i^{(r)}} + \frac{\partial L}{\partial \mu_i} \frac{\partial \mu_i}{\partial v_i^{(r)}} + \frac{\partial L}{\partial \sigma_i^2} \frac{\partial \sigma_i^2}{\partial v_i^{(r)}} \tag{3.62}$$

$$= \frac{\partial L}{\partial \hat{v}_i^{(r)}} \left(\frac{1}{\sigma_i}\right) + \frac{\partial L}{\partial \mu_i} \left(\frac{1}{m}\right) + \frac{\partial L}{\partial \sigma_i^2} \left(\frac{2(v_i^{(r)} - \mu_i)}{m}\right) \tag{3.63}$$

We need to evaluate each of the three partial derivatives on the right-hand side of the above equation in terms of the quantities that have been computed using the already-executed dynamic programming updates of backpropagation. This allows the creation of the recurrence equation for the batch normalization layer. Among these, the first expression, which is $\frac{\partial L}{\partial \hat{v}_i^{(r)}}$, can be substituted in terms of the loss derivatives of the next layer by observing that $a_i^{(r)}$ is related to $\hat{v}_i^{(r)}$ by a constant of proportionality γ_i:

$$\frac{\partial L}{\partial \hat{v}_i^{(r)}} = \gamma_i \frac{\partial L}{\partial a_i^{(r)}} \quad [\text{Since } a_i^{(r)} = \gamma_i \cdot \hat{v}_i^{(r)} + \beta_i] \tag{3.64}$$

Therefore, by substituting this value of $\frac{\partial L}{\partial \hat{v}_i^{(r)}}$ in Equation 3.63, we have the following:

$$\frac{\partial L}{\partial v_i^{(r)}} = \frac{\partial L}{\partial a_i^{(r)}} \left(\frac{\gamma_i}{\sigma_i}\right) + \frac{\partial L}{\partial \mu_i} \left(\frac{1}{m}\right) + \frac{\partial L}{\partial \sigma_i^2} \left(\frac{2(v_i^{(r)} - \mu_i)}{m}\right) \tag{3.65}$$

It now remains to compute the partial derivative of the loss with respect to the mean and the variance. The partial derivative of the loss with respect to the variance is computed as follows:

$$\frac{\partial L}{\partial \sigma_i^2} = \underbrace{\sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} \cdot \frac{\partial \hat{v}_i^{(q)}}{\partial \sigma_i^2}}_{\text{Chain rule}} = \underbrace{-\frac{1}{2\sigma_i^3} \sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} (v_i^{(q)} - \mu_i)}_{\text{Use Equation 3.60}} = \underbrace{-\frac{1}{2\sigma_i^3} \sum_{q=1}^{m} \frac{\partial L}{\partial a_i^{(q)}} \gamma_i \cdot (v_i^{(q)} - \mu_i)}_{\text{Substitution from Equation 3.64}}$$

The partial derivatives of the loss with respect to the mean can be computed as follows:

$$\frac{\partial L}{\partial \mu_i} = \underbrace{\sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} \cdot \frac{\partial \hat{v}_i^{(q)}}{\partial \mu_i} + \frac{\partial L}{\partial \sigma_i^2} \cdot \frac{\partial \sigma_i^2}{\partial \mu_i}}_{\text{Chain rule}} = \underbrace{-\frac{1}{\sigma_i} \sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} - 2 \frac{\partial L}{\partial \sigma_i^2} \cdot \frac{\sum_{q=1}^{m} (v_i^{(q)} - \mu_i)}{m}}_{\text{Use Equations 3.59 and 3.60}}$$

$$= \underbrace{-\frac{\gamma_i}{\sigma_i} \sum_{q=1}^{m} \frac{\partial L}{\partial a_i^{(q)}}}_{\text{Eq. 3.64}} + \underbrace{\left(\frac{1}{\sigma_i^3}\right) \cdot \left(\sum_{q=1}^{m} \frac{\partial L}{\partial a_i^{(q)}} \gamma_i \cdot (v_i^{(q)} - \mu_i)\right) \cdot \left(\frac{\sum_{q=1}^{m} (v_i^{(q)} - \mu_i)}{m}\right)}_{\text{Substitution for } \frac{\partial L}{\partial \sigma_i^2}}$$

By plugging in the partial derivatives of the loss with respect to the mean and variance in Equation 3.65, we get a full recursion for $\frac{\partial L}{\partial v_i^{(r)}}$ (value before batch-normalization layer) in terms of $\frac{\partial L}{\partial a_i^{(r)}}$ (value *after* the batch normalization layer). This provides a full view of the backpropagation of the loss through the batch-normalization layer corresponding to the BN node. The other aspects of backpropagation remain similar to the traditional case. Batch normalization enables faster inference because it prevents problems such as the exploding and vanishing gradient (which cause slow learning).

A natural question about batch normalization arises during inference (prediction) time. Since the transformation parameters μ_i and σ_i depend on the batch, how should one compute them during testing when a *single* test instance is available? In this case, the values of μ_i are σ_i are computed up front using the *entire* population (of training data), and then treated as constants during testing time. One can also keep an exponentially weighted average of these values during training. Therefore, the normalization is a simple linear transformation during inference.

An interesting property of batch normalization is that *it also acts as a regularizer*. Note that the same data point can cause somewhat different updates depending on which batch it is included in. One can view this effect as a kind of noise added to the update process. Regularization is often achieved by adding a small amount of noise to the training data. It has been experimentally observed that regularization methods like *Dropout* (cf. Section 4.5.4 of Chapter 4) do not seem to improve performance when batch normalization is used [184], although there is not a complete agreement on this point. A variant of batch normalization, known as *layer normalization*, is known to work well with recurrent networks. This approach is discussed in Section 7.3.1 of Chapter 7.

3.7 Practical Tricks for Acceleration and Compression

Neural network learning algorithms can be extremely expensive, both in terms of the number of parameters in the model and the amount of data that needs to be processed. There are several strategies that are used to accelerate and compress the underlying implementations. Some of the common strategies are as follows:

1. *GPU-acceleration:* Graphics Processor Units (GPUs) have historically been used for rendering video games with intensive graphics because of their efficiency in settings where repeated matrix operations (e.g., on graphics pixels) are required. It was eventually realized by the machine learning community (and GPU hardware companies) that such repetitive operations are also used in settings like neural networks, in which matrix operations are extensively used. Even the use of a single GPU can significantly speed up implementation because of its high memory bandwidth and multithreading within its multicore architecture.

2. *Parallel implementations:* One can parallelize the implementations of neural networks by using multiple GPUs or CPUs. Either the neural network model or the data can be partitioned across different processors. These implementations are referred to as *model-parallel* and *data-parallel* implementations.

3. *Algorithmic tricks for model compression during deployment:* A key point about the practical use of neural networks is that they have different computational requirements during training and deployment. While it is acceptable to train a model for a week with a large amount of memory, the final deployment might be performed on a mobile

phone, which is highly constrained both in terms of memory and computational power. Therefore, numerous tricks are used for model compression during testing time. This type of compression often results in better cache performance and efficiency as well.

In the following, we will discuss some of these acceleration and compression techniques.

3.7.1 GPU Acceleration

GPUs were originally developed for rendering graphics on screens with the use of lists of 3-dimensional coordinates. Therefore, graphics cards were inherently designed to perform many matrix multiplications in parallel to render the graphics rapidly. GPU processors have evolved significantly, moving well beyond their original functionality of graphics rendering. Like graphics applications, neural-network implementations require large matrix multiplications, which is inherently suited to the GPU setting. In a traditional neural network, each forward propagation is a multiplication of a matrix and vector, whereas in a convolutional neural network, two matrices are multiplied. When a mini-batch approach is used, activations become matrices (instead of vectors) in a traditional neural network. Therefore, forward propagations require matrix multiplications. A similar result is true for backpropagation, during which two matrices are multiplied frequently to propagate the derivatives backwards. In other words, most of the intensive computations involve vector, matrix, and tensor operations. Even a single GPU is good at parallelizing these operations in its different cores with multithreading [203], in which some groups of threads sharing the same code are executed concurrently. This principle is referred to as *Single Instruction Multiple Threads (SIMT)*. Although CPUs also support short-vector data parallelization via *Single Instruction Multiple Data (SIMD)* instructions, the degree of parallelism is much lower as compared to the GPU. There are different trade-offs when using GPUs as compared to traditional CPUs. GPUs are very good at repetitive operations, but they have difficulty at performing branching operations like *if-then* statements. Most of the intensive operations in neural network learning are repetitive matrix multiplications across different training instances, and therefore this setting is suited to the GPU. Although the clock speed of a single instruction in the GPU is slower than the traditional CPU, the parallelization is so much greater in the GPU that huge advantages are gained.

GPU threads are grouped into small units called *warps*. Each thread in the warp shares the same code in each cycle, and this restriction enables a concurrent execution of the threads. The implementation needs to be carefully tailored to reduce the use of memory bandwidth. This is done by *coalescing* the memory reads and writes from different threads, so that a single memory transaction can be used to read and write values from different threads. Consider a common operation like matrix multiplication in neural network settings. The matrices are multiplied by making each thread responsible for computing a single entry in the product matrix. For example, consider a situation in which a 100×50 matrix is multiplied with a 50×200 matrix. In such a case, a total of $100 \times 200 = 20000$ threads would be launched in order to compute the entries of the matrix. These threads will typically be partitioned into multiple warps, each of which is highly parallelized. Therefore, speedups are achieved. A discussion of matrix multiplication on GPUs is provided in [203].

With high amounts of parallelization, memory bandwidth is often the primary limiting factor. Memory bandwidth refers to the speed at which the processor can access the relevant parameters from their stored locations in memory. GPUs have a high degree of parallelism and high memory bandwidth as compared to traditional CPUs. Note that if one cannot access the relevant parameters from memory fast enough, then faster execution does not

help the speed of computation. In such cases, the memory transfer cannot keep up with the speed of the processor whether working with the CPU or the GPU, and the CPU/GPU cores will idle. GPUs have different trade-offs between cache access, computation, and memory access. CPUs have much larger caches than GPUs and they rely on the caches to store an intermediate result, such as the result of multiplying two numbers. Accessing a computed value from a cache is much faster than multiplying them again, which is where the CPU has an advantage over the GPU. However, this advantage is neutralized in neural network settings, where the sizes of the parameter matrices and activations are often too large to fit in the CPU cache. Even though the CPU cache is larger than that of the GPU, it is not large enough to handle the scale at which neural-network operations are performed. In such cases, one has to rely on high memory bandwidth, which is where the GPU has an advantage over the CPU. Furthermore, it is often faster to perform the same computation again rather than accessing it from memory, when working with the GPU (assuming that the result is unavailable in a cache). Therefore, GPU implementations are done somewhat differently from traditional CPU implementations. Furthermore, the advantage gained can be sensitive to the choice of neural network architecture, as the memory bandwidth requirements and multi-threading gains of different architectures can be different.

At first sight, it might seem from the above example that the use of a GPU requires a lot of low-level programming, and it would indeed be a challenge to create custom GPU code for each neural architecture. With this problem in mind, companies like NVIDIA have modularized the interface between the programmer and the GPU implementation. The key point is that the speeding of primitives like matrix multiplication and convolution can be hidden from the user by providing a library of neural network operations that perform these faster operations behind the scenes. The GPU library is tightly integrated with deep learning frameworks like Caffe or Torch to take advantage of the accelerated operations on the GPU. A specific example of such a library is the *NVIDIA CUDA Deep Neural Network Library* [643], which is referred to in short as *cuDNN*. CUDA is a parallel computing platform and programming model that works with CUDA-enabled GPU processors. However, it provides an abstraction and a programming interface that is easy to use with relatively limited rewriting of code. The cuDNN library can be integrated with multiple deep learning frameworks such as Caffe, TensorFlow, Theano, and Torch. The changes required to convert the training code of a particular neural network from its CPU version to a GPU version are often small. For example, in Torch, the CUDA Torch package is incorporated at the beginning of the code, and various data structures (like tensors) are initialized as CUDA tensors (instead of regular tensors). With these types of modest modifications, virtually the same code can run on a GPU instead of a CPU in Torch. A similar situation holds true in other deep learning frameworks. This type of approach shields the developers from the low-level performance tuning required in GPU frameworks, because the primitives in the library already have the code that takes care of all the low-level details of parallelization on the GPU.

3.7.2 Parallel and Distributed Implementations

It is possible to make training even faster by using multiple CPUs or GPUs. Since it is more common to use multiple GPUs, we focus on this setting. Parallelism is not a simple matter when working with GPUs because there are overheads associated with the communication between different processors. The delay caused by these overheads has recently been reduced with specialized network cards for GPU-to-GPU transfer. Furthermore, algorithmic tricks like using 8-bit approximations of the gradients [98] can help in speeding up the

communication. There are several ways in which one can partition the work across different processors, namely hyperparameter parallelism, model parallelism, and data parallelism. These methods are discussed below.

Hyperparameter Parallelism

The simplest possible way to achieve parallelism in the training process without much overhead is to train neural networks with different parameter settings on different processors. No communication is required across different executions, and therefore wasteful overhead is avoided. As discussed earlier in this chapter, runs with suboptimal hyperparameters are often terminated long before running them to completion. Nevertheless, a small number of different runs with optimized parameters are often used in order to create an ensemble of models. The training of different ensemble components can be performed independently on different processors.

Model Parallelism

Model parallelism is particularly useful when a single model is too large to fit on a GPU. In such a case, the hidden layer is divided across the different GPUs. The different GPUs work on exactly the same batch of training points, although different GPUs compute different parts of the activations and the gradients. Each GPU only contains the portion of the weight matrix that are multiplied with the hidden activations present in the GPU. However, it would still need to communicate the results of its activations to the other GPUs. Similarly, it would need to receive the derivatives with respect to the hidden units in other GPUs in order to compute the gradients of the weights between its hidden units and those of other GPUs. This is achieved with the use of inter-connections across GPUs, and the computations across these interconnections add to the overhead. In some cases, these interconnections are dropped in a subset of the layers in order to reduce the communication overhead (although the resulting model would not quite be the same as the sequential version). Model parallelism is not helpful in cases where the number of parameters in the neural network is small, and should only be used for large networks. A good practical example of model parallelism is the design of *AlexNet*, which is a convolutional neural network (cf. Section 8.4.1 of Chapter 8). A sequential version of *AlexNet* and a GPU-partitioned version of *AlexNet* are both shown in Figure 8.9 of Chapter 8. Note that the sequential version in Figure 8.9 is not exactly equivalent to the GPU-partitioned version because the interconnections between GPUs have been dropped in some of the layers. A discussion of model parallelism may be found in [74].

Data Parallelism

Data parallelism works best when the model is small enough to fit on each GPU, but the amount of training data is large. In these cases, the parameters are shared across the different GPUs and the goal of the updates is to use the different processors with different training points for faster updates. The problem is that perfect synchronization of the updates can slow down the process, because locking mechanisms would need to be used to synchronize the updates. The key point is that each processor would have to wait for the others to make their updates. As a result, the slowest processor creates a bottleneck. A method that uses *asynchronous* stochastic gradient descent was proposed in [91]. The basic idea is to use a parameter server in order to share the parameters across different GPU processors. The updates are performed without using any locking mechanism. In other words, each GPU can read the shared parameters at any time, perform the computation, and write the

parameters to the parameter server without worrying about locks. In this case, inefficiency would still be caused by one GPU processor overwriting the progress made by another, but there would be no waiting times for writes. As a result, the overall progress would still be faster than with a synchronized mechanism. Distributed asynchronous gradient descent is quite popular as a strategy for parallelism in large-scale industrial settings.

Exploiting the Trade-Offs for Hybrid Parallelism

It is evident from the above discussion that model parallelism is well suited to models with a large parameter footprint, whereas data parallelism is well suited to smaller models. It turns out that one can combine the two types of parallelism over different parts of the network. In certain types of convolutional neural networks that have fully connected layers, the vast majority of parameters occur in the fully connected layers, whereas more computations are performed in the earlier layers. In these cases, it makes sense to use data parallelism for the early part of the network, and model parallelism for the later part of the network. This type of approach is referred to as *hybrid parallelism*. A discussion of this type of approach may be found in [254].

3.7.3 Algorithmic Tricks for Model Compression

Training a neural network and deploying it typically have different requirements in terms of memory and efficiency requirements. While it may be acceptable to require a week to train a neural network to recognize faces in images, the end user might wish to use the trained neural network to recognize a face within a matter of a few seconds. Furthermore, the model might be deployed on a mobile device with little memory and computational availability. In such cases, it is crucial to be able to use the trained model efficiently, and also use it with a limited amount of storage. Efficiency is generally not a problem at deployment time, because the prediction of a test instance often requires straightforward matrix multiplications over a few layers. On the other hand, storage requirements are often a problem because of the large number of parameters in multilayer networks. There are several tricks that are used for model compression in such cases. In most of the cases, a larger trained neural network is modified so that it requires less space by approximating some parts of the model. In addition, some efficiency improvements can also be realized at prediction time by model compression because of better cache performance and fewer operations, although this is not the primary goal. Interestingly, this approximation might occasionally *improve* accuracy on out-of-sample predictions because of regularization effects, especially if the original model is unnecessarily large compared to the training data size.

Sparsifying Weights in Training

The links in a neural network are associated with weights. If the absolute value of a particular weight is small, then the model is not strongly influenced by that weight. Such weights can be dropped, and the neural network can be fine-tuned starting with the current weights on links that have not yet been dropped. The level of sparsification will depend on the weight threshold at which links are dropped. By choosing a larger threshold at which weights are dropped, the size of the model will reduce significantly. In such cases, it is particularly important to fine-tune the values of the retained weights with further epochs of training. One can also encourage the dropping of links by using L_1-regularization, which will be discussed in Chapter 4. When L_1-regularization is used during training, many of

the weights will have zero values anyway because of the natural mathematical properties of this form of regularization. However, it has been shown in [169] that L_2-regularization has the advantage of higher accuracy. Therefore, the work in [169] uses L_2-regularization and prunes the weights that are below a particular threshold.

Further enhancements were reported in [168], where the approach was combined with Huffman coding and quantization for compression. The goal of quantization is to reduce the number of bits representing each connection. This approach reduced the storage required by *AlexNet* [255] by a factor of 35, or from about 240MB to 6.9MB, with no loss of accuracy. It is now possible as a result of this reduction to fit the model into an on-chip SRAM cache rather than off-chip DRAM memory; this also provide a beneficial effect on prediction times.

Leveraging Redundancies in Weights

It was shown in [94] that the vast majority of the weights in a neural network are redundant. In other words, for any $m \times n$ weight matrix W between a pair of layers with m_1 and m_2 units respectively, one can express this weight matrix as $W \approx UV^T$, where U and V are of sizes $m_1 \times k$ and $m_2 \times k$, respectively. Furthermore, it is assumed that $k \ll \min\{m_1, m_2\}$. This phenomenon occurs because of several peculiarities in the training process. For example, the features and weights in a neural network tend to *co-adapt* because of different parts of the network training at different rates. Therefore, the faster parts of the network often adapt to the slower parts. As a result, there is a lot of redundancy in the network both in terms of the features and the weights, and the full expressivity of the network is never utilized. In such a case, one can replace the pair of layers (containing weight matrix W) with three layers of size m_1, k, and m_2. The weight matrices between the first pair of layers is U and the weight matrix between the second pair of layers is V^T. Even though the new matrix is deeper, it is better regularized as long as $W - UV^T$ only contains noise. Furthermore, the matrices U and V require $(m_1 + m_2) \cdot k$ parameters, which is less than the number of parameters in W as long as k is less than half the harmonic mean of m_1 and m_2:

$$\frac{\text{Parameters in } W}{\text{Parameters in } U, V} = \frac{m_1 \cdot m_2}{k(m_1 + m_2)} = \frac{\text{HARMONIC-MEAN}(m_1, m_2)}{2k}$$

As shown in [94], more than 95% of the parameters in the neural network are redundant, and therefore a low value of the rank k suffices for approximation.

An important point is that the replacement of W with U and V must be done *after* completion of the learning of W. For example, if we replaced the pair of layers corresponding to W with the three layers containing the two weight matrices U and V^T and trained from scratch, good results may not be obtained. This is because co-adaptation will occur again during training, and the resulting matrices U and V will have a rank even lower than k. As a result, under-fitting might occur.

Finally, one can compress even further by realizing that both U and V need not be learned because they are redundant with respect to each other. For any rank-k matrix U, one can learn V so that the product UV^T is the same value. Therefore, the work in [94] provides methods to fix U, and then learn V instead.

Hash-Based Compression

One can reduce the number of parameters to be stored by forcing randomly chosen entries of the weight matrix to take on shared values of the parameters. The random choice is achieved with the application of a hash function on the entry position (i, j) in the matrix.

For example, imagine a situation where we have a weight matrix of size 100×100 with 10^4 entries. In such a case, one can hash each weight to a value in the range $\{1, \ldots 1000\}$ to create 1000 groups. Each of these groups will contain an average of 10 connections that will share weights. Backpropagation can handle shared weights using the approach discussed in Section 3.2.9. This approach requires a space requirement of only 1000 for the matrix, which is 10% of the original space requirement. Note that one could instead use a matrix of size 100×10 to achieve the same compression, but the key point is that using shared weights does not hurt the expressivity of the model as much as would reducing the size of the weight matrix *a priori*. More details of this approach are discussed in [66].

Leveraging Mimic Models

Some interesting results in [13, 55] show that it is possible to significantly compress a model by creating a new training data set from a trained model, which is easier to model. This "easier" training data can be used to train a much smaller network without significant loss of accuracy. This smaller model is referred to as a *mimic model*. The following steps are used to create the mimic model:

1. A model is created on the original training data. This model might be very large, and potentially even created out of an ensemble of different models, further increasing the number of parameters; it would not be appropriate to use in space-constrained settings. It is assumed that the model outputs softmax probabilities of the different classes. This model is also referred to as the teacher model.

2. New training data is created by passing unlabeled examples through the trained network. The targets in the newly created training data are set to the softmax probability outputs of the trained model on the unlabeled examples. Since unlabeled data is often copious, it is possible to create a lot of training data in this way. It is noteworthy that the new training data contains soft (probabilistic) targets rather than the discrete targets in the original training data, which significantly contributes to the creation of the compressed model.

3. A much smaller and shallower network is trained using the new training data (with artificially generated labels). The original training data is not used at all. This much smaller and shallower network, which is referred to as the mimic or *student* model, is what is deployed in space-constrained settings. It can be shown that the accuracy of the mimic model does not substantially degrade from the model trained over the original neural network, even though it is much smaller in size.

A natural question arises as to why the mimic model should perform as well as the original model, even though it is much smaller in size both in terms of the depth as well as the number of parameters. Trying to construct a shallow model on the original data cannot match the accuracy of either the shallow model or the mimic model. A number of possible reasons have been hypothesized for the superior performance of the mimic model [13]:

1. If there are errors in the original training data because of mislabeling, it causes unnecessary complexity in the trained model. These types of errors are largely removed in the new training data.

2. If there are complex regions of the decision space, the teacher model simplifies them by providing softer labels in terms of probabilities. Complexity is washed away by filtering targets through the teacher model.

3. The original training data contains targets with 0/1 values, whereas the newly created training contains soft targets, which are more informative. This is particularly useful in one-hot encoded multilabel targets, where there are clear correlations across different classes.

4. The original targets might depend on inputs that are not available in the training data. On the other hand, the teacher-created labels depend on only the available inputs. This makes the model simpler to learn and washes away unexplained complexity. Unexplained complexity often leads to unnecessary parameters and depth.

One can view some of the above benefits as a kind of regularization effect. The results in [13] are stimulating, because they show that deep networks are not *theoretically* necessary, although the regularization effect of depth is practically necessary when working with the original training data. The mimic model enjoys the benefits of this regularization effect by using the artificially created targets instead of depth.

3.8 Summary

This chapter discusses the problem of training deep neural networks. We revisit the backpropagation algorithm in detail along with its challenges. The vanishing and the exploding gradient problems are introduced along with the challenges associated with varying sensitivity of the loss function to different optimization variables. Certain types of activation functions like ReLU are less sensitive to this problem. However, the use of the ReLU can sometimes lead to dead neurons, if one is not careful about the learning rate. The type of gradient descent used to accelerate learning is also important for more efficient executions. Modified stochastic gradient-descent methods include the use of Nesterov momentum, AdaGrad, AdaDelta, RMSProp, and Adam. All these methods encourage gradient-steps that accelerate the learning process.

Numerous methods have been introduced for addressing the problem of cliffs with the use of second-order optimization methods. In particular, Hessian-free optimization is seen as an effective approach for handling many of the underlying optimization issues. An exciting method that has been used recently to improve learning rates is the use of batch normalization. Batch normalization transforms the data layer by layer in order to ensure that the scaling of different variables is done in an optimum way. The use of batch normalization has become extremely common in different types of deep networks. Numerous methods have been proposed for accelerating and compressing neural network algorithms. Acceleration is often achieved via hardware improvements, whereas compression is achieved with algorithmic tricks.

3.9 Bibliographic Notes

The original idea of backpropagation was based on idea of differentiation of composition of functions as developed in control theory [54, 237] under the ambit of *automatic differentiation*. The adaptation of these methods to neural networks was proposed by Paul Werbos in his PhD thesis in 1974 [524], although a more modern form of the algorithm was proposed by Rumelhart *et al.* in 1986 [408]. A discussion of the history of the backpropagation algorithm may be found in the book by Paul Werbos [525].

A discussion of algorithms for hyperparameter optimization in neural networks and other machine learning algorithms may be found in [36, 38, 490]. The random search method for

hyperparameter optimization is discussed in [37]. The use of *Bayesian optimization* for hyperparameter tuning is discussed in [42, 306, 458]. Numerous libraries are available for Bayesian tuning such as *Hyperopt* [614], *Spearmint* [616], and *SMAC* [615].

The rule that the initial weights should depend on both the fan-in and fan-out of a node in proportion to $\sqrt{2/(r_{in} + r_{out})}$ is based on [140]. The analysis of initialization methods for rectifier neural networks is provided in [183]. Evaluations and analysis of the effect of feature preprocessing on neural network learning may be found in [278, 532]. The use of rectifier linear units for addressing some of the training challenges is discussed in [141].

Nesterov's algorithm for gradient descent may be found in [353]. The delta-bar-delta method was proposed by [217]. The AdaGrad algorithm was proposed in [108]. The RMSProp algorithm is discussed in [194]. Another adaptive algorithm using stochastic gradient descent, which is *AdaDelta*, is discussed in [553]. This algorithms shares some similarities with second-order methods, and in particular to the method in [429]. The Adam algorithm, which is a further enhancement along this line of ideas, is discussed in [241]. The practical importance of initialization and momentum in deep learning is discussed in [478]. Beyond the use of the stochastic gradient method, the use of coordinate descent has been proposed [273]. The strategy of *Polyak averaging* is discussed in [380].

Several of the challenges associated with the vanishing and exploding gradient problems are discussed in [140, 205, 368]. Ideas for parameter initialization that avoid some of these problems are discussed in [140]. The gradient clipping rule was discussed by Mikolov in his PhD thesis [324]. A discussion of the gradient clipping method in the context of recurrent neural networks is provided in [368]. The ReLU activation function was introduced in [167], and several of its interesting properties are explored in [141, 221].

A description of several second-order gradient optimization methods (such as the Newton method) is provided in [41, 545, 300]. The basic principles of the conjugate gradient method have been described in several classical books and papers [41, 189, 443], and the work in [313, 314] discusses applications to neural networks. The work in [316] leverages a Kronecker-factored curvature matrix for fast gradient descent. Another way of approximating the Newton method is the quasi-Newton method [273, 300], with the simplest approximation being a diagonal Hessian [24]. The acronym BFGS stands for the Broyden-Fletcher-Goldfarb-Shanno algorithm. A variant known as limited memory BFGS or L-BFGS [273, 300] does not require as much memory. Another popular second-order method is the Levenberg–Marquardt algorithm. This approach is, however, defined for squared loss functions and cannot be used with many forms of cross-entropy or log-losses that are common in neural networks. Overviews of the approach may be found in [133, 300]. General discussions of different types of nonlinear programming methods are provided in [23, 39].

The stability of neural networks to local minima is discussed in [88, 426]. Batch normalization methods were introduced recently in [214]. A method that uses whitening for batch normalization is discussed in [96], although the approach seems not to be practical. Batch normalization requires some minor adjustments for recurrent networks [81], although a more effective approach for recurrent networks is that of *layer normalization* [14]. In this method (cf. Section 7.3.1), a single training case is used for normalizing all units in a layer, rather than using mini-batch normalization of a single unit. The approach is useful for recurrent networks. An analogous notion to batch normalization is that of weight normalization [419], in which the magnitudes and directions of the weight vectors are decoupled during the learning process. Related training tricks are discussed in [362].

A broader discussion of accelerating machine learning algorithms with GPUs may be found in [644]. Various types of parallelization tricks for GPUs are discussed in [74, 91, 254], and specific discussions on convolutional neural networks are provided in [541]. Model

compression with regularization is discussed in [168, 169]. A related model compression method is proposed in [213]. The use of mimic models for compression is discussed in [55, 13]. A related approach is discussed in [202]. The leveraging of parameter redundancy for compressing neural networks is discussed in [94]. The compression of neural networks with the hashing trick is discussed in [66].

3.9.1 Software Resources

All the training algorithms discussed in this chapter are supported by numerous deep learning frameworks like *Caffe* [571], *Torch* [572], *Theano* [573], and *TensorFlow* [574]. Extensions of *Caffe* to Python and MATLAB are available. All these frameworks provide a variety of training algorithms that are discussed in this chapter. Options for batch normalization are available as separate layers in these frameworks. Several software libraries are available for Bayesian optimization of hyperparameters. These libraries include *Hyperopt* [614], *Spearmint* [616], and *SMAC* [615]. Although these are designed for smaller machine learning problems, they can still be used in some cases. Pointers to the NVIDIA cuDNN may be found in [643]. The different frameworks supported by cuDNN are discussed in [645].

3.10 Exercises

1. Consider the following recurrence:

$$(x_{t+1}, y_{t+1}) = (f(x_t, y_t), g(x_t, y_t)) \tag{3.66}$$

Here, $f()$ and $g()$ are multivariate functions.

(a) Derive an expression for $\frac{\partial x_{t+2}}{\partial x_t}$ in terms of only x_t and y_t.

(b) Can you draw an architecture of a neural network corresponding to the above recursion for t varying from 1 to 5? Assume that the neurons can compute any function you want.

2. Consider a two-input neuron that multiplies its two inputs x_1 and x_2 to obtain the output o. Let L be the loss function that is computed at o. Suppose that you know that $\frac{\partial L}{\partial o} = 5$, $x_1 = 2$, and $x_2 = 3$. Compute the values of $\frac{\partial L}{\partial x_1}$ and $\frac{\partial L}{\partial x_2}$.

3. Consider a neural network with three layers including an input layer. The first (input) layer has four inputs x_1, x_2, x_3, and x_4. The second layer has six hidden units corresponding to all pairwise multiplications. The output node o simply adds the values in the six hidden units. Let L be the loss at the output node. Suppose that you know that $\frac{\partial L}{\partial o} = 2$, and $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, and $x_4 = 4$. Compute $\frac{\partial L}{\partial x_i}$ for each i.

4. How does your answer to the previous question change when the output o is computed as a maximum of its six inputs rather than its sum?

5. The chapter discusses (cf. Table 3.1) how one can perform a backpropagation of an arbitrary function by using the multiplication with the Jacobian matrix. Discuss why one must be careful in using this matrix-centric approach.[Hint: Compute the Jacobian with respect to sigmoid function]

6. Consider the loss function $L = x^2 + y^{10}$. Implement a simple steepest-descent algorithm to plot the coordinates as they vary from the initialization point to the optimal value of 0. Consider two different initialization points of $(0.5, 0.5)$ and $(2, 2)$ and plot the trajectories in the two cases at a constant learning rate. What do you observe about the behavior of the algorithm in the two cases?

7. The Hessian H of a strongly convex quadratic function always satisfies $\overline{x}^T H \overline{x} > 0$ for any nonzero vector $\overline{x}$. For such problems, show that all conjugate directions are linearly independent.

8. Show that if the dot product of a d-dimensional vector $\overline{v}$ with d linearly independent vectors is 0, then $\overline{v}$ must be the zero vector.

9. This chapter discusses two variants of backpropagation, which use the pre-activation and the postactivation variables, respectively, for the dynamic programming recursion. Show that these two variants of backpropagation are mathematically equivalent.

10. Consider the softmax activation function in the output layer, in which real-valued outputs $v_1 \ldots v_k$ are converted into probabilities as follows (according to Equation 3.20):

$$o_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\}$$

(a) Show that the value of $\frac{\partial o_i}{\partial v_j}$ is $o_i(1 - o_i)$ when $i = j$. In the case that $i \neq j$, show that this value is $-o_i o_j$.

(b) Use the above result to show the correctness of Equation 3.22:

$$\frac{\partial L}{\partial v_i} = o_i - y_i$$

Assume that we are using the cross-entropy loss $L = -\sum_{i=1}^{k} y_i \log(o_i)$, where $y_i \in \{0, 1\}$ is the one-hot encoded class label over different values of $i \in \{1 \ldots k\}$.

11. The chapter uses steepest descent directions to iteratively generate conjugate directions. Suppose we pick d *arbitrary* directions $\overline{v}_0 \ldots \overline{v}_{d-1}$ that are linearly independent. Show that (with appropriate choice of β_{ti}) we can start with $\overline{q}_0 = \overline{v}_0$ and generate successive conjugate directions in the following form:

$$\overline{q}_{t+1} = \overline{v}_{t+1} + \sum_{i=0}^{t} \beta_{ti} \overline{q}_i$$

Discuss why this approach is more expensive than the one discussed in the chapter.

12. The definition of β_t in Section 3.5.6.1 ensures that $\overline{q}_t$ is conjugate to $\overline{q}_{t+1}$. This exercise systematically shows that *any* direction $\overline{q}_i$ for $i \leq t$ satisfies $\overline{q}_i^T H \overline{q}_{t+1} = 0$.
[Hint: Prove (b), (c), and (d) *jointly* with induction on t while staring at (a).]

(a) Recall from Equation 3.51 that $H\overline{q}_i = [\nabla L(\overline{W}_{i+1}) - \nabla L(\overline{W}_i)]/\delta_i$ for quadratic loss functions, where δ_i depends on ith step-size. Combine this condition with Equation 3.49 to show the following for all $i \leq t$:

$$\delta_i [\overline{q}_i^T H \overline{q}_{t+1}] = -[\nabla L(\overline{W}_{i+1}) - \nabla L(\overline{W}_i)]^T [\nabla L(\overline{W}_{t+1})] + \delta_i \beta_t (\overline{q}_i^T H \overline{q}_t)$$

Also show that $[\nabla L(\overline{W}_{t+1}) - \nabla L(\overline{W}_t)] \cdot \overline{q}_i = \delta_t \overline{q}_i^T H \overline{q}_t$.

(b) Show that $\nabla L(\overline{W}_{t+1})$ is orthogonal to each $\overline{q}_i$ for $i \leq t$. [The proof for the case when $i = t$ is trivial because the gradient at line-search termination is always orthogonal to the search direction.]

(c) Show that the loss gradients at $\overline{W}_0 \ldots \overline{W}_{t+1}$ are mutually orthogonal.

(d) Show that $\overline{q}_i^T H \overline{q}_{t+1} = 0$ for $i \leq t$. [The case for $i = t$ is trivial.]

Chapter 4

Teaching Deep Learners to Generalize

"All generalizations are dangerous, even this one."—Alexandre Dumas

4.1 Introduction

Neural networks are powerful learners that have repeatedly proven to be capable of learning complex functions in many domains. However, the great power of neural networks is also their greatest weakness; neural networks often simply overfit the training data if care is not taken to design the learning process carefully. In practical terms, what overfitting means is that a neural network will provide excellent prediction performance on the training data that it is built on, but will perform poorly on unseen test instances. This is caused by the fact that the learning process often remembers random artifacts of the training data that do not generalize well to the test data. Extreme forms of overfitting are referred to as *memorization*. A helpful analogy is to think of a child who can solve all the analytical problems for which he or she has seen the solutions, but is unable to provide useful solutions to a new problem. However, if the child is exposed to the solutions of more and more different types of problems, he or she will be more likely to solve a new problem by abstracting out the essence of the patterns that are repeated across different problems and their solutions. Machine learning proceeds in a similar way by identifying patterns that are useful for prediction. For example, in a spam detection application, if the pattern "*Free Money!!*" occurs thousands of times in spam emails, the machine learner generalizes this rule to identify spam email instances it has not seen before. On the other hand, a prediction that is based on the patterns seen in a tiny training data set of two emails will lead to good performance on those emails but not on new emails. The ability of a learner to provide useful predictions for instances it has not seen before is referred to as *generalization*.

Generalization is a useful practical property, and is therefore the holy grail in all machine learning applications. After all, if the training examples are already labeled, there is no practical use of predicting such examples again. For example, in an image-captioning application,

one is always looking to use the labeled images in order to learn captions for images that the learner has not seen before.

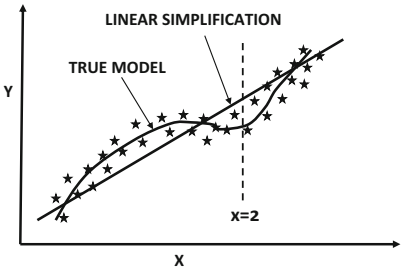


Figure 4.1: An example of a nonlinear distribution in which one would expect a model with $d = 3$ to work better than a linear model with $d = 1$.

The level of overfitting depends both on the complexity of the model and on the amount of data available. The complexity of the model defined by a neural network depends on the number of underlying parameters. Parameters provide additional degrees of freedom, which can be used to explain specific training data points without generalizing well to unseen points. For example, imagine a situation in which we attempt to predict the variable y from x using the following formula for polynomial regression:

$$\hat{y} = \sum_{i=0}^{d} w_i x^i \tag{4.1}$$

This is a model that uses $(d + 1)$ parameters $w_0 \ldots w_d$ in order to explain pairs (x, y) available to us. One could implement this model by using a neural network with d inputs corresponding to x, $x^2 \ldots x^d$, and a single bias neuron whose coefficient is w_0. The loss function uses the squared difference between the observed value y and predicted value $\hat{y}$. In general, larger values of d can capture better nonlinearity. For example, in the case of Figure 4.1, a nonlinear model with $d = 4$ should be able to fit the data better than a linear model with $d = 1$, *given an infinite amount (or a lot) of data*. However, when working with a small, finite data set, this does not always turn out to be the case.

If we have $(d+1)$ or less training pairs (x, y), it is possible to fit the data exactly with zero error *irrespective of how well these training pairs reflect the true distribution*. For example, consider a situation in which we have five training points available. One can show that it is possible to fit the training points exactly with zero error using a polynomial of degree 4. This does not, however, mean that zero error will be achieved on unseen test data. An example of this situation is illustrated in Figure 4.2, where both the linear and polynomial models on three sets of five randomly chosen data points are shown. It is clear that the linear model is stable, although it is unable to exactly model the curved nature of the true data distribution. On the other hand, even though the polynomial model is capable of modeling the true data distribution more closely, it varies wildly over the different training data sets. Therefore, the same test instance at $x = 2$ (shown in Figure 4.2) would receive similar predictions from the linear model, but would receive very different predictions from the polynomial model over different choices of training data sets. The behavior of the polynomial model is, of course, undesirable from a practitioner's point of view, who would expect similar predictions for a particular test instance, even when different samples of the training data set are used. Since all the different predictions of the polynomial model cannot be correct, it is evident that the increased power of the polynomial model over the linear model actually increases

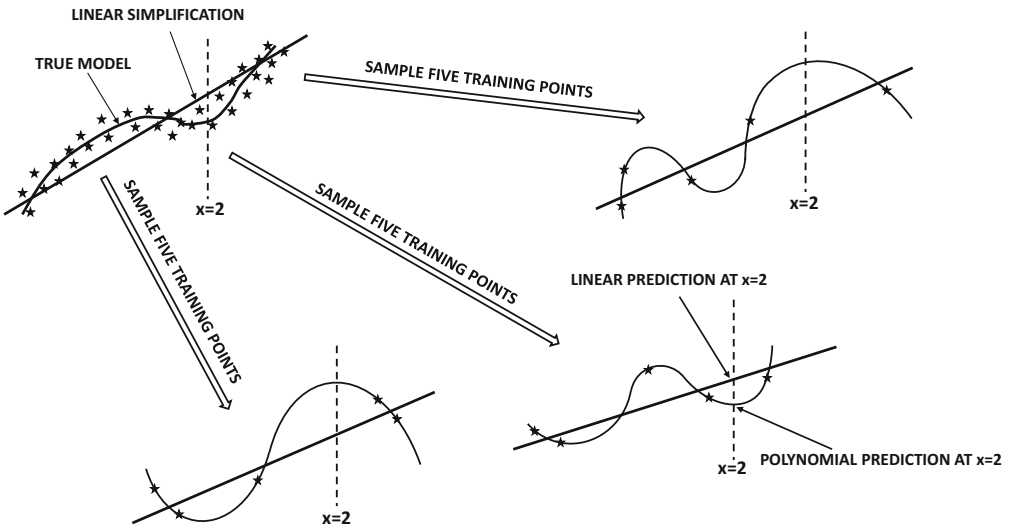


Figure 4.2: **Overfitting with increased model complexity:** The linear model does not change much with the training data, whereas the polynomial model changes drastically. As a result, the inconsistent predictions of the polynomial model at $x = 2$ are often more inaccurate than those of the linear model. The polynomial model does have the ability to outperform the linear model *if enough training data is provided.*

the error rather than reducing it. This difference in predictions for the same test instance (but different training data sets) is manifested as the *variance* of a model. As evident from Figure 4.2, models with high variance tend to memorize random artifacts of the training data, causing inconsistency and inaccuracy in the prediction of unseen test instances. It is noteworthy that a polynomial model with higher degree is inherently more powerful than a linear model because the higher-order coefficients could always be set to 0; however, it is unable to achieve its full potential when the amount of data is limited. Simply speaking, the variance inherent in the finiteness of the data set causes increased complexity to be counterproductive. This trade-off between the power of a model and its performance on limited data is captured with the *bias-variance trade-off*.

There are several tell-tale signs of overfitting:

1. When a model is trained on different data sets, the same test instance might obtain very different predictions. This is a sign that the training process is memorizing the nuances of the specific training data set, rather than learning patterns that generalize to unseen test instances. Note that the three predictions at $x = 2$ in Figure 4.2 are quite different for the polynomial model. This is not quite the case for the linear model.

2. The gap between the error of predicting training instances and unseen test instances is rather large. Note that in Figure 4.2, the predictions at the unseen test point $x = 2$ are often more inaccurate in the polynomial model than in the linear model. On the other hand, the training error is always zero for the polynomial model, whereas the training error is always nonzero for the linear model.

Because of the large gaps between training and test error, models are often tested on unseen portions of the training data. These unseen portions of the test data are often held out early on, and then used in order to make different types of algorithmic decisions such as parameter tuning. This set of points is referred to as the *validation set*. The final accuracy is tested on a fully out-of-sample set of points that was not used for either model building or for parameter tuning. The error on out-of-sample test data is also referred to as the *generalization error*.

Neural networks are large, and they might have millions of parameters in complex applications. In spite of these challenges, there are a number of tricks that one can use in order to ensure that overfitting is not a problem. The choice of method depends on the specific setting, and the type of neural network used. The key methods for avoiding overfitting in a neural network are as follows:

1. *Penalty-based regularization:* Penalty-based regularization is the most common technique used by neural networks in order to avoid overfitting. The idea in regularization is to create a penalty or other types of constraints on the parameters in order to favor simpler models. For example, in the case of polynomial regression, a possible constraint on the parameters would be to ensure that at most k different values of w_i are non-zero. This will ensure simpler models. However, since it is hard to impose such constraints explicitly, a simpler approach is to impose a softer penalty like $\lambda \sum_{i=0}^{d} w_i^2$ and add it to the loss function. Such an approach roughly amounts to multiplying each parameter w_i with a multiplicative decay factor of $(1 - \alpha\lambda)$ before each update at learning rate α. Aside from penalizing parameters of the network, one can also choose to penalize the activations of hidden units. This approach often leads to sparse hidden representations.

2. *Generic and tailored ensemble methods:* Many ensemble methods are not specific to neural networks, but can be used for other machine learning problems. We will discuss bagging and subsampling, which are two of the simplest ensemble methods that can be implemented for virtually any model or learning problem. These methods are inherited from traditional machine learning.

   There are several ensemble methods that are specifically designed for neural networks. A straightforward approach is to average the predictions of different neural architectures obtained by quick and dirty hyper-parameter optimization. *Dropout* is another ensemble technique that is designed for neural networks. This technique uses the selective dropping of nodes to create different neural networks. The predictions of different networks are combined to create the final result. Dropout reduces overfitting by indirectly acting as a regularizer.

3. *Early stopping:* In early stopping, the iterative optimization method is terminated early without converging to the optimal solution on the training data. The stopping point is determined using a portion of the training data that is not used for model building. One terminates when the error on the held-out data begins to rise. Even though this approach is not optimal for the training data, it seems to perform well on the test data because the stopping point is determined on the basis of the held-out data.

4. *Pretraining:* Pretraining is a form of learning in which a greedy algorithm is used to find a good initialization. The weights in different layers of the neural network are trained sequentially in greedy fashion. These trained weights are used as a good starting point for the overall process of learning. Pretraining can be shown to be an indirect form of regularization.

5. *Continuation and curriculum methods:* These methods perform more effectively by first training simple models, and then making them more complex. The idea is that it is easy to train simpler models without overfitting. Furthermore, starting with the optimum point of the simpler model provides a good initialization for a complex model that is closely related to the simpler model. It is noteworthy that some of these methods can be considered similar to pretraining. Pretraining also finds solutions from the simple to the complex by decomposing the training of a deep neural network into a set of shallow layers.

6. *Sharing parameters with domain-specific insights:* In some data-domains like text and images, one often has some insight about the structure of the parameter space. In such cases, some of the parameters in different parts of the network can be set to the same value. This reduces the number of degrees of freedom of the model. Such an approach is used in recurrent neural networks (for sequence data) and convolutional neural networks (for image data). Sharing parameters does come with its own set of challenges because the backpropagation algorithm needs to be appropriately modified to account for the sharing.

This chapter will first discuss the issue of model generalization in a generic way by introducing some theoretical results associated with the bias-variance trade-off. Subsequently, the different ways of reducing overfitting will be discussed.

An interesting observation is that several forms of regularization can be shown to be roughly equivalent to the injection of noise in either the input data or the hidden variables. For example, it can be shown that many penalty-based regularizers are equivalent to the addition of noise [44]. Furthermore, even the use of *stochastic* gradient descent instead of gradient descent can be viewed as a kind of noise addition to the steps of the algorithm. As a result, stochastic gradient descent often shows excellent accuracy on the test data, even though its performance on the training data might not be as good as that of gradient descent. Furthermore, some ensemble techniques like *Dropout* and data perturbation are equivalent to injecting noise. Throughout this chapter, the similarities between noise injection and regularization will be discussed where needed.

Even though a natural way of avoiding overfitting is to simply build smaller networks (with fewer units and parameters), it has often been observed that it is better to build large networks and then regularize them in order to avoid overfitting. This is because large networks retain the *option* of building a more complex model if it is truly warranted. At the same time, the regularization process can smooth out the random artifacts that are not supported by sufficient data. By using this approach, we are giving the model the choice to decide what complexity it needs, rather than making a rigid decision for the model up front (which might even underfit the data).

Supervised settings tend to be more prone to overfitting than unsupervised settings, and supervised problems are therefore the main focus of the literature on generalization. To understand this point, consider that a supervised application tries to learn a single target variable and might have hundreds of input (explanatory) variables. It is easy to overfit the process of learning a very focused goal because a limited degree of supervision (e.g., binary label) is available for each training example. On the other hand, an unsupervised application has the same number of target variables as the explanatory variables. After all, we are trying to model the entire data from itself. In the latter case, overfitting is less likely (albeit still possible) because a single training example has a larger number of bits of information. Nevertheless, regularization is still used in unsupervised applications, especially when the intent is to impose a desired structure on the learned representations.

Chapter Organization

This chapter is organized as follows. The next section introduces the bias-variance trade-off. The practical implications of the bias-variance trade-off for model training are discussed in Section 4.3. The use of penalty-based regularization to reduce overfitting is presented in Section 4.4. Ensemble methods are explained in Section 4.5. Some methods, such as bagging, are generic techniques, whereas others (like *Dropout*) are specifically designed for neural networks. Early stopping methods are discussed in Section 4.6. Methods for unsupervised pretraining are discussed in Section 4.7. Continuation and curriculum learning methods are presented in Section 4.8. Parameter sharing methods are discussed in Section 4.9. Unsupervised forms of regularization are discussed in Section 4.10. A summary is given in Section 4.11.

4.2 The Bias-Variance Trade-Off

The introduction section provides an example of how a polynomial model fits a smaller training data set, leading to the predictions on unseen test data being more erroneous than are the predictions of a (simpler) linear model. This is because a polynomial model requires more data in order to not be misled by random artifacts of the training data set. The fact that more powerful models do not always win in terms of prediction accuracy with a finite data set is the key take-away from the bias-variance trade-off.

The bias-variance trade-off states that the squared error of a learning algorithm can be partitioned into three components:

1. *Bias:* The bias is the error caused by the simplifying assumptions in the model, which causes certain test instances to have consistent errors across different choices of training data sets. Even if the model has access to an infinite source of training data, the bias cannot be removed. For example, in the case of Figure 4.2, the linear model has a higher model bias than the polynomial model, because it can never fit the (slightly curved) data distribution exactly, no matter how much data is available. The prediction of a particular out-of-sample test instance at $x = 2$ will always have an error in a particular direction when using a linear model for any choice of training sample. If we assume that the linear and curved lines in the top left of Figure 4.2 were estimated using an infinite amount of data, then the difference between the two at any particular values of x is the bias. An example of the bias at $x = 2$ is shown in Figure 4.2.

2. *Variance:* Variance is caused by the inability to learn all the parameters of the model in a statistically robust way, especially when the data is limited and the model tends to have a larger number of parameters. The presence of higher variance is manifested by overfitting to the specific training data set at hand. Therefore, if different choices of training data sets are used, different predictions will be provided for the same test instance. Note that the linear prediction provides similar predictions at $x = 2$ in Figure 4.2, whereas the predictions of the polynomial model vary widely over different choices of training instances. In many cases, the widely inconsistent predictions at $x = 2$ are wildly incorrect predictions, which is a manifestation of model variance. Therefore, the polynomial predictor has a higher variance than the linear predictor in Figure 4.2.

3. *Noise:* The noise is caused by the inherent error in the data. For example, all data points in the scatter plot vary from the true model in the upper-left corner of Fig-

ure 4.2. If there had been no noise, all points in the scatter plot would overlap with the curved line representing the true model.

The above description provides a qualitative view of the bias-variance trade-off. In the following, we will provide a more formal and mathematical view.

4.2.1 Formal View

We assume that the base distribution from which the training data set is generated is denoted by $\mathcal{B}$. One can generate a data set $\mathcal{D}$ from this base distribution:

$$\mathcal{D} \sim \mathcal{B} \tag{4.2}$$

One could draw the training data in many different ways, such as selecting only data sets of a particular size. For now, assume that we have some well-defined generative process according to which training data sets are drawn from $\mathcal{B}$. The analysis below does not rely on the specific mechanism with which training data sets are drawn from $\mathcal{B}$.

Access to the base distribution $\mathcal{B}$ is equivalent to having access to an infinite resource of training data, because one can use the base distribution an unlimited number of times to generate training data sets. In practice, such base distributions (i.e., infinite resources of data) are not available. As a practical matter, an analyst uses some data collection mechanism to collect only *one finite instance* of $\mathcal{D}$. However, the conceptual existence of a base distribution from which other training data sets can be generated is useful in theoretically quantifying the sources of error in training on this finite data set.

Now imagine that the analyst had a set of t test instances in d dimensions, denoted by $\overline{Z_1} \ldots \overline{Z_t}$. The dependent variables of these test instances are denoted by $y_1 \ldots y_t$. For clarity in discussion, let us assume that the test instances and their dependent variables were also generated from the same base distribution $\mathcal{B}$ by a third party, but the analyst was provided access only to the feature representations $\overline{Z_1} \ldots \overline{Z_t}$, and no access to the dependent variables $y_1 \ldots y_t$. Therefore, the analyst is tasked with job of using the single finite instance of the training data set $\mathcal{D}$ in order to predict the dependent variables of $\overline{Z_1} \ldots \overline{Z_t}$.

Now assume that the relationship between the dependent variable y_i and its feature representation $\overline{Z_i}$ is defined by the *unknown* function $f(\cdot)$ as follows:

$$y_i = f(\overline{Z_i}) + \epsilon_i \tag{4.3}$$

Here, the notation ϵ_i denotes the intrinsic noise, which is independent of the model being used. The value of ϵ_i might be positive or negative, although it is assumed that $E[\epsilon_i] = 0$. If the analyst knew what the function $f(\cdot)$ corresponding to this relationship was, then they could simply apply the function to each test point $\overline{Z_i}$ in order to approximate the dependent variable y_i, with the only remaining uncertainty being caused by the intrinsic noise.

The problem is that the analyst does not know what the function $f(\cdot)$ is in practice. Note that this function is used within the generative process of the base distribution $\mathcal{B}$, and the entire generating process is like an oracle that is unavailable to the analyst. The analyst only has examples of the input and output of this function. Clearly, the analyst would need to develop some type of *model* $g(\overline{Z_i}, \mathcal{D})$ using the training data in order to *approximate* this function in a data-driven way.

$$\hat{y}_i = g(\overline{Z_i}, \mathcal{D}) \tag{4.4}$$

Note the use of the circumflex (i.e., the symbol '^') on the variable $\hat{y}_i$ to indicate that it is a *predicted* value by a specific algorithm rather than the observed (true) value of y_i.

All prediction functions of learning models (including neural networks) are examples of the estimated function $g(\cdot, \cdot)$. Some algorithms (such as linear regression and perceptrons) can even be expressed in a concise and understandable way:

$$g(\overline{Z_i}, \mathcal{D}) = \underbrace{\overline{W} \cdot \overline{Z_i}}_{\text{Learn } \overline{W} \text{ with } \mathcal{D}} \quad \text{[Linear Regression]}$$

$$g(\overline{Z_i}, \mathcal{D}) = \underbrace{\text{sign}\{\overline{W} \cdot \overline{Z_i}\}}_{\text{Learn } \overline{W} \text{ with } \mathcal{D}} \quad \text{[Perceptron]}$$

Most neural networks are expressed algorithmically as compositions of multiple functions computed at different nodes. The choice of computational function includes the effect of its specific parameter setting, such as the coefficient vector $\overline{W}$ in a perceptron. Neural networks with a larger number of units will require more parameters to fully learn the function. This is where the variance in predictions arises on the same test instance; a model with a large parameter set $\overline{W}$ will learn very different values of these parameters, when a different choice of the training data set is used. Consequently, the prediction of the same test instance will also be very different for different training data sets. These inconsistencies add to the error, as illustrated in Figure 4.2.

The goal of the bias-variance trade-off is to quantify the expected error of the learning algorithm in terms of its bias, variance, and the (data-specific) noise. For generality in discussion, we assume a numeric form of the target variable, so that the error can be intuitively quantified by the *mean-squared error* between the predicted values $\hat{y}_i$ and the observed values y_i. This is a natural form of error quantification in regression, although one can also use it in classification in terms of probabilistic predictions of test instances. The mean squared error, MSE, of the learning algorithm $g(\cdot, \mathcal{D})$ is defined over the set of test instances $\overline{Z_1} \ldots \overline{Z_t}$ as follows:

$$MSE = \frac{1}{t} \sum_{i=1}^{t} (\hat{y}_i - y_i)^2 = \frac{1}{t} \sum_{i=1}^{t} (g(\overline{Z_i}, \mathcal{D}) - f(\overline{Z_i}) - \epsilon_i)^2$$

The best way to estimate the error in a way that is independent of the specific choice of training data set is to compute the *expected* error over different choices of training data sets:

$$\begin{aligned} E[MSE] &= \frac{1}{t} \sum_{i=1}^{t} E[(g(\overline{Z_i}, \mathcal{D}) - f(\overline{Z_i}) - \epsilon_i)^2] \\ &= \frac{1}{t} \sum_{i=1}^{t} E[(g(\overline{Z_i}, \mathcal{D}) - f(\overline{Z_i}))]^2 + \frac{\sum_{i=1}^{t} E[\epsilon_i^2]}{t} \end{aligned}$$

The second relationship is obtained by expanding the quadratic expression on the right-hand side of the first equation, and then using the fact that the average value of ϵ_i over a large number of test instances is 0.

The right-hand side of the above expression can be further decomposed by adding and subtracting $E[g(\overline{Z_i}, \mathcal{D})]$ within the squared term on the right-hand side:

$$E[MSE] = \frac{1}{t} \sum_{i=1}^{t} E[\{(f(\overline{Z_i}) - E[g(\overline{Z_i}, \mathcal{D})]) + (E[g(\overline{Z_i}, \mathcal{D})] - g(\overline{Z_i}, \mathcal{D}))\}^2] + \frac{\sum_{i=1}^{t} E[\epsilon_i^2]}{t}$$

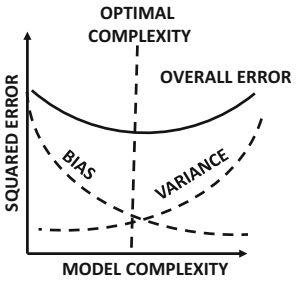


Figure 4.3: The trade-off between bias and variance usually causes a point of optimal model complexity.

One can expand the quadratic polynomial on the right-hand side to obtain the following:

$$\begin{aligned} E[MSE] =& \frac{1}{t}\sum_{i=1}^{t} E[\{f(\overline{Z_i}) - E[g(\overline{Z_i}, \mathcal{D})]\}^2] \\ &+ \frac{2}{t}\sum_{i=1}^{t} \{f(\overline{Z_i}) - E[g(\overline{Z_i}, \mathcal{D})]\}\{E[g(\overline{Z_i}, \mathcal{D})] - E[g(\overline{Z_i}, \mathcal{D})]\} \\ &+ \frac{1}{t}\sum_{i=1}^{t} E[\{E[g(\overline{Z_i}, \mathcal{D})] - g(\overline{Z_i}, \mathcal{D})\}^2] + \frac{\sum_{i=1}^{t} E[\epsilon_i^2]}{t} \end{aligned}$$

The second term on the right-hand side of the aforementioned expression evaluates to 0 because one of the multiplicative factors is $E[g(\overline{Z_i}, \mathcal{D})] - E[g(\overline{Z_i}, \mathcal{D})]$. On simplification, we obtain the following:

$$E[MSE] = \underbrace{\frac{1}{t}\sum_{i=1}^{t} \{f(\overline{Z_i}) - E[g(\overline{Z_i}, \mathcal{D})]\}^2}_{\text{Bias}^2} + \underbrace{\frac{1}{t}\sum_{i=1}^{t} E[\{g(\overline{Z_i}, \mathcal{D}) - E[g(\overline{Z_i}, \mathcal{D})]\}^2]}_{\text{Variance}} + \underbrace{\frac{\sum_{i=1}^{t} E[\epsilon_i^2]}{t}}_{\text{Noise}}$$

In other words, the squared error can be decomposed into the (squared) bias, variance, and noise. The variance is the key term that prevents neural networks from generalizing. In general, the variance will be higher for neural networks that have a large number of parameters. On the other hand, too few model parameters can cause bias because there are not sufficient degrees of freedom to model the complexities of the data distribution. This trade-off between bias and variance with increasing model complexity is illustrated in Figure 4.3. Clearly, there is a point of optimal model complexity where the performance is optimized. Furthermore, paucity of training data will increase variance. However, careful choice of design can reduce overfitting. This chapter will discuss several such choices.

4.3 Generalization Issues in Model Tuning and Evaluation

There are several practical issues in the training of neural network models that one must be careful of because of the bias-variance trade-off. The first of these issues is associated with model tuning and hyperparameter choice. For example, if one tuned the neural network with the same data that were used to train it, one would not obtain very good results because of overfitting. Therefore, the hyperparameters (e.g., regularization parameter) are tuned on a separate held-out set than the one on which the weight parameters on the neural network are learned.

Given a labeled data set, one needs to use this resource for training, tuning, and testing the accuracy of the model. Clearly, one cannot use the entire resource of labeled data for model building (i.e., learning the weight parameters). For example, using the same data set for both model building and testing grossly overestimates the accuracy. This is because the main goal of classification is to *generalize* a model of labeled data to unseen test instances. Furthermore, the portion of the data set used for *model selection* and *parameter tuning* also needs to be different from that used for model building. A common mistake is to use the same data set for both parameter tuning and final evaluation (testing). Such an approach partially mixes the training and test data, and the resulting accuracy is overly optimistic. A given data set should always be divided into three parts defined according to the way in which the data are used:

1. *Training data:* This part of the data is used to build the training model (i.e., during the process of learning the weights of the neural network). Several design choices may be available during the building of the model. The neural network might use different hyperparameters for the learning rate or for regularization. The same training data set may be tried multiple times over different choices for the hyperparameters or completely different algorithms to build the models in multiple ways. This process allows estimation of the relative accuracy of different algorithm settings. This process sets the stage for *model selection*, in which the best algorithm is selected out of these different models. However, the actual *evaluation* of these algorithms for selecting the best model is not done on the training data, but on a separate validation data set to avoid favoring overfitted models.

2. *Validation data:* This part of the data is used for model selection and parameter tuning. For example, the choice of the learning rate may be tuned by constructing the model multiple times on the first part of the data set (i.e., training data), and then using the validation set to estimate the accuracy of these different models. As discussed in Section 3.3.1 of Chapter 3, different combinations of parameters are sampled within a range and tested for accuracy on the validation set. The best choice of each combination of parameters is determined by using this accuracy. In a sense, validation data should be viewed as a kind of test data set to tune the parameters of the algorithm (e.g., learning rate, number of layers or units in each layer), or to select the best design choice (e.g., sigmoid versus tanh activation).

3. *Testing data:* This part of the data is used to test the accuracy of the final (tuned) model. It is important that the testing data are not even looked at during the process of parameter tuning and model selection to prevent overfitting. The testing data are *used only once at the very end of the process.* Furthermore, if the analyst uses the results on the test data to adjust the model in some way, then the results will be

contaminated with knowledge from the testing data. The idea that one is allowed to look at a test data set only once is an extraordinarily strict requirement (and an important one). Yet, it is frequently violated in real-life benchmarks. The temptation to use what one has learned from the final accuracy evaluation is simply too high.

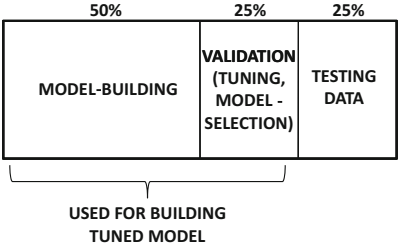


Figure 4.4: Partitioning a labeled data set for evaluation design

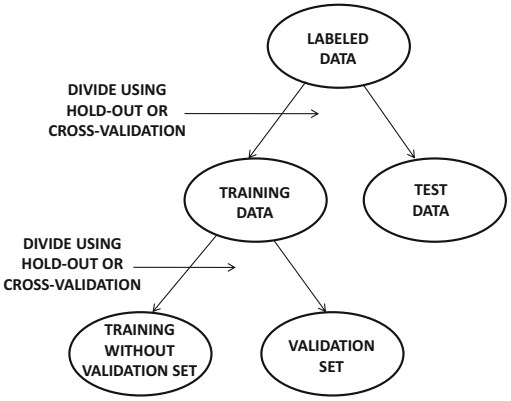


Figure 4.5: Hierarchical division into training, validation, and testing portions

The division of the labeled data set into training data, validation data, and test data is shown in Figure 4.4. Strictly speaking, the validation data is also a part of the training data, because it influences the final model (although only the model building portion is often referred to as the training data). The division in the ratio of 2:1:1 is a conventional rule of thumb that has been followed since the nineties. However, it should not be viewed as a strict rule. For very large labeled data sets, one needs only a modest number of examples to estimate accuracy. When a very large data set is available, it makes sense to use as much of it for model building as possible, because the variance induced by the validation and evaluation stage is often quite low. A constant number of examples (e.g., less than a few thousand) in the validation and test data sets are sufficient to provide accurate estimates. Therefore, the 2:1:1 division is a rule of thumb inherited from an era in which data sets were small. In the modern era, where data sets are large, almost all of the points are used for training, and a modest (constant) number are used for testing. It is not uncommon to have divisions such as 98:1:1.

4.3.1 Evaluating with Hold-Out and Cross-Validation

The aforementioned description of partitioning the labeled data into three segments is an implicit description of a method referred to as *hold-out* for segmenting the labeled data into various portions. However, the division into *three* parts is not done in one shot. Rather, the training data is first divided into *two* parts for training and testing. The testing part is then carefully hidden away from any further analysis *until the very end where it can be used only once*. The remainder of the data set is then divided again into the training and validation portions. This type of recursive division is shown in Figure 4.5.

A key point is that the types of division at both levels of the hierarchy are conceptually identical. In the following, we will consistently use the terminology of the first level of division in Figure 4.5 into "training" and "testing" data, even though the same approach

can also be used for the second-level division into model building and validation portions. This allows us to provide a common description of evaluation processes at both levels of the division.

Hold-Out

In the hold-out method, a fraction of the instances are used to build the training model. The remaining instances, which are also referred to as the *held-out* instances, are used for testing. The accuracy of predicting the labels of the held-out instances is then reported as the overall accuracy. Such an approach ensures that the reported accuracy is not a result of overfitting to the specific data set, because different instances are used for training and testing. The approach, however, underestimates the true accuracy. Consider the case where the held-out examples have a higher presence of a particular class than the labeled data set. This means that the held-in examples have a lower average presence of the same class, which will cause a mismatch between the training and test data. Furthermore, the class-wise frequency of the held-in examples will always be inversely related to that of the held-out examples. This will lead to a consistent pessimistic bias in the evaluation. In spite of these weaknesses, the hold-out method has the advantage of being simple and efficient, which makes it a popular choice in large-scale settings. From a deep-learning perspective, this is an important observation because large data sets are common.

Cross-Validation

In the cross-validation method, the labeled data is divided into q equal segments. One of the q segments is used for testing, and the remaining $(q-1)$ segments are used for training. This process is repeated q times by using each of the q segments as the test set. The average accuracy over the q different test sets is reported. Note that this approach can closely estimate the true accuracy when the value of q is large. A special case is one where q is chosen to be equal to the number of labeled data points and therefore a single point is used for testing. Since this single point is left out from the training data, this approach is referred to as *leave-one-out cross-validation*. Although such an approach can closely approximate the accuracy, it is usually too expensive to train the model a large number of times. In fact, cross-validation is sparingly used in neural networks because of efficiency issues.

4.3.2 Issues with Training at Scale

One practical issue that arises in the specific case of neural networks is when the sizes of the training data sets are large. Therefore, while methods like cross-validation are well established to be superior choices to hold-out in traditional machine learning, their technical soundness is often sacrificed in favor of efficiency. In general, training time is such an important consideration in neural network modeling that many compromises have to be made to enable practical implementation.

A computational problem often arises in the context of grid search of hyperparameters (cf. Section 3.3.1 of Chapter 3). Even a single hyperparameter choice can sometimes require a few days to evaluate, and a grid search requires the testing of a large number of possibilities. Therefore, a common strategy is to run the training process of each setting for a fixed number of epochs. Multiple runs are executed over different choices of hyperparameters in different threads of execution. Those choices of hyperparameters in which good progress is not made after a fixed number of epochs are terminated. In the end, only a few ensemble members are

allowed to run to completion. One reason that such an approach works well is because the vast majority of the progress is often made in the early phases of the training. This process is also described in Section 3.3.1 of Chapter 3.

4.3.3 How to Detect Need to Collect More Data

The high generalization error in a neural network may be caused by several reasons. First, the data itself might have a lot of noise, in which case there is little one can do in order to improve accuracy. Second, neural networks are hard to train, and the large error might be caused by the poor convergence behavior of the algorithm. The error might also be caused by high bias, which is referred to as *underfitting*. Finally, overfitting (i.e., high variance) may cause a large part of the generalization error. In most cases, the error is a combination of more than one of these different factors. However, one can detect overfitting in a specific training data set by examining the gap between the training and test accuracy. Overfitting is manifested *by a large gap between training and test accuracy*. It is not uncommon to have close to 100% training accuracy on a small training set, even when the test error is quite low. The first solution to this problem is to collect more data. With increased training data, the training accuracy will reduce, whereas the test/validation accuracy will increase. However, if more data is not available, one would need to use other techniques such as regularization in order to improve generalization performance.

4.4 Penalty-Based Regularization

Penalty-based regularization is the most common approach for reducing overfitting. In order to understand this point, let us revisit the example of the polynomial with degree d. In this case, the prediction $\hat{y}$ for a given value of x is as follows:

$$\hat{y} = \sum_{i=0}^{d} w_i x^i \tag{4.5}$$

It is possible to use a single-layer network with d inputs and a single bias neuron with weight w_0 in order to model this prediction. The ith input is x^i. This neural network uses linear activations, and the squared loss function for a set of training instances (x, y) from data set $\mathcal{D}$ can be defined as follows:

$$L = \sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2$$

As discussed in the example of Figure 4.2, a large value of d tends to increase overfitting. One possible solution to this problem is to reduce the value of d. In other words, using a model with *economy in parameters* leads to a simpler model. For example, reducing d to 1 creates a linear model that has fewer degrees of freedom and tends to fit the data in a similar way over different training samples. However, doing so does lose some expressivity when the data patterns are indeed complex. In other words, oversimplification reduces the expressive power of a neural network, so that it is unable to adjust sufficiently to the needs of different types of data sets.

How can one retain some of this expressiveness without causing too much overfitting? Instead of reducing the number of parameters in a hard way, one can use a *soft* penalty on the use of parameters. Furthermore, large (absolute) values of the parameters are penalized

more than small values, because small values do not affect the prediction significantly. What kind of penalty can one use? The most common choice is L_2-regularization, which is also referred to as *Tikhonov regularization*. In such a case, the additional penalty is defined by the sum of squares of the values of the parameters. Then, for the regularization parameter $\lambda > 0$, one can define the objective function as follows:

$$L = \sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2 + \lambda \cdot \sum_{i=0}^{d} w_i^2$$

Increasing or decreasing the value of λ reduces the softness of the penalty. One advantage of this type of parameterized penalty is that one can tune this parameter for optimum performance on a portion of the training data set that is not used for learning the parameters. This type of approach is referred to as *model validation*. Using this type of approach provides greater flexibility than fixing the economy of the model up front. Consider the case of polynomial regression discussed above. Restricting the number of parameters up front severely constrains the learned polynomial to a specific shape (e.g., a linear model), whereas a soft penalty is able to control the shape of the learned polynomial in a more data-driven manner. In general, it has been experimentally observed that it is more desirable to use complex models (e.g., larger neural networks) with regularization rather than simple models without regularization. The former also provides greater flexibility by providing a tunable knob (i.e., regularization parameter), which can be chosen in a data-driven manner. The value of the tunable knob is learned on a held-out portion of the data set.

How does regularization affect the updates in a neural network? For any given weight w_i in the neural network, the updates are defined by gradient descent (or the batched version of it):

$$w_i \Leftarrow w_i - \alpha \frac{\partial L}{\partial w_i}$$

Here, α is the learning rate. The use of L_2-regularization is roughly equivalent to the use of decay imposition after each parameter update:

$$w_i \Leftarrow w_i(1 - \alpha\lambda) - \alpha \frac{\partial L}{\partial w_i}$$

Note that the update above first multiplies the weight with the decay factor $(1 - \alpha\lambda)$, and then uses the gradient-based update. The decay of the weights can also be understood in terms of a biological interpretation, if we assume that the initial values of the weights are close to 0. One can view weight decay as a kind of forgetting mechanism, which brings the weights closer to their initial values. This ensures that only the repeated updates have a significant effect on the absolute magnitude of the weights. A forgetting mechanism prevents a model from *memorizing* the training data, because only significant and repeated updates will be reflected in the weights.

4.4.1 Connections with Noise Injection

The addition of noise to the input has connections with penalty-based regularization. It can be shown that the addition of an equal amount of Gaussian noise to each input is equivalent to Tikhonov regularization of a single-layer neural network with an identity activation function (for linear regression).

One way of showing this result is by examining a single training case $(\overline{X}, y)$, which becomes $(\overline{X} + \sqrt{\lambda}\overline{\epsilon}, y)$ after noise with variance λ is added to each feature. Here, $\overline{\epsilon}$ is a

random vector, in which each entry ϵ_i is independently drawn from the standard normal distribution with zero mean and unit variance. Then, the noisy prediction $\hat{y}$, which is based on $\overline{X} + \sqrt{\lambda}\overline{\epsilon}$, is as follows:

$$\hat{y} = \overline{W} \cdot (\overline{X} + \sqrt{\lambda}\epsilon) = \overline{W} \cdot \overline{X} + \sqrt{\lambda}\overline{W} \cdot \overline{\epsilon} \tag{4.6}$$

Now, let us examine the squared loss function $L = (y - \hat{y})^2$ contributed by a single training case. We will compute the *expected* value of the loss function. It is easy to show the following in expectation:

$$\begin{aligned} E[L] &= E[(y - \hat{y})^2] \\ &= E[(y - \overline{W} \cdot \overline{X} - \sqrt{\lambda}\overline{W} \cdot \overline{\epsilon})^2] \end{aligned}$$

One can then expand the expression on the right-hand side as follows:

$$\begin{aligned} E[L] &= (y - \overline{W} \cdot \overline{X})^2 - 2\sqrt{\lambda}(y - \overline{W} \cdot \overline{X}) \underbrace{E[\overline{W} \cdot \overline{\epsilon}]}_{0} + \lambda E[(\overline{W} \cdot \overline{\epsilon})^2] \\ &= (y - \overline{W} \cdot \overline{X})^2 + \lambda E[(\overline{W} \cdot \overline{\epsilon})^2] \end{aligned}$$

The second expression can be expanded using $\overline{\epsilon} = (\epsilon_1 \ldots \epsilon_d)$ and $\overline{W} = (w_1 \ldots w_d)$. Furthermore, one can set any term of the form $E[\epsilon_i \epsilon_j]$ to $E[\epsilon_i] \cdot E[\epsilon_j] = 0$ because of independence of the random variables ϵ_i and ϵ_j. Any term of the form $E[\epsilon_i^2]$ is set to 1, because each ϵ_i is drawn from a standard normal distribution. On expanding $E[(\overline{W} \cdot \overline{\epsilon})^2]$ and making the above substitutions, one finds the following:

$$E[L] = (y - \overline{W} \cdot \overline{X})^2 + \lambda(\sum_{i=1}^{d} w_i^2) \tag{4.7}$$

It is noteworthy that *this loss function is exactly the same as L_2-regularization* of a single instance.

Although the equivalence between weight decay and noise addition is exactly true for the case of linear regression, the analysis does not hold in the case of neural networks with nonlinear activations. Nevertheless, penalty-based regularization continues to be intuitively similar to noise addition even in these cases, although the results might be qualitatively different. Because of these similarities one sometimes tries to perform regularization by direct noise addition. One such approach is referred to as *data perturbation*, in which noise is added to the training input, and the test data points are predicted with the added noise. The approach is repeated multiple times with different training data sets created by adding noise repeatedly in Monte Carlo fashion. The prediction of the same test instance across different additions of noise is averaged in order to yield the improved results. In this case, the noise is added only to the training data, and it does not need to be added to the test data. When explicitly adding noise, it is important to average the prediction of the same test instance over multiple ensemble components in order to ensure that the solution properly represents the *expected* value of the loss (without added variance caused by the noise). This approach is described in Section 4.5.5.

4.4.2 L_1-Regularization

The use of the squared norm penalty, which is also referred to as L_2-regularization, is the most common approach for regularization. However, it is possible to use other types of

penalties on the parameters. A common approach is L_1-regularization in which the squared penalty is replaced with a penalty on the sum of the absolute magnitudes of the coefficients. Therefore, the new objective function is as follows:

$$L = \sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2 + \lambda \cdot \sum_{i=0}^{d} |w_i|_1$$

The main problem with this objective function is that it contains the term $|w_i|$, which is not differentiable when w_i is exactly equal to 0. This requires some modifications to the gradient-descent method when w_i is 0. For the case when w_i is non-zero, one can use the straightforward update obtained by computing the partial derivative. By differentiating the above objective function, we can define the update equation at least for the case when w_i is different than 0:

$$w_i \Leftarrow w_i - \alpha\lambda s_i - \alpha\frac{\partial L}{\partial w_i}$$

The value of s_i, which is the partial derivative of $|w_i|$ (with respect to w_i), is as follows:

$$s_i = \begin{cases} -1 & w_i < 0 \\ +1 & w_i > 0 \end{cases}$$

However, we also need to set the partial derivative of $|w_i|$ for cases in which the value of w_i is exactly 0. One possibility is to use the *subgradient* method in which the value of w_i is set stochastically to a value in $\{-1, +1\}$. However, this is not necessary in practice. Computers are of finite-precision, and the computational errors will rarely cause w_i to be *exactly* 0. Therefore, the computational errors will often perform the task that would otherwise be achieved by stochastic sampling. Furthermore, for the rare cases in which the value w_i is exactly 0, one can omit the regularization and simply set s_i to 0. This type of approximation to the subgradient method works reasonably well in many settings.

One difference between the update equations for L_1-regularization and those in L_2-regularization is that L_2-regularization uses multiplicative decay as a forgetting mechanism, whereas L_1-regularization uses additive updates as a forgetting mechanism. In both cases, the regularization portions of the updates tend to move the coefficients closer to 0. However, there are some differences in the types of solutions found in the two cases, which are discussed in the next section.

4.4.3 L_1- or L_2-Regularization?

A question arises as to whether L_1- or L_2-regularization is desirable. From an accuracy point of view, L_2-regularization usually outperforms L_1-regularization. This is the reason that L_2-regularization is almost always preferred over L_1-regularization is most implementations. The performance gap is small when the number of inputs and units is large.

However, L_1-regularization does have specific applications from an interpretability point of view. An interesting property of L_1-regularization is that it creates *sparse* solutions in which the vast majority of the values of w_i are 0s (after ignoring[1] computational errors). If the value of w_i is zero for a connection incident on the input layer, then that particular input has no effect on the final prediction. In other words, such an input can be *dropped*,

[1]Computational errors can be ignored by requiring that $|w_i|$ should be at least 10^{-6} in order for w_i to be considered truly non-zero.

and the L_1-regularizer acts as a feature selector. Therefore, one can use L_1-regularization to estimate which features are predictive to the application at hand.

What about the connections in the hidden layers whose weights are set to 0? These connections can be dropped, which results in a sparse neural network. Such sparse neural networks can be useful in cases where one repeatedly performs training on the same type of data set, but the nature and broader characteristics of the data set do not change significantly with time. Since the sparse neural network will contain only a small fraction of the connections in the original neural network, it can be retrained much more efficiently whenever more training data is received.

4.4.4 Penalizing Hidden Units: Learning Sparse Representations

The penalty-based methods, which have been discussed so far, penalize the *parameters* of the neural network. A different approach is to penalize the *activations* of the neural network, so that only a small subset of the neurons are activated for any given data instance. In other words, even though the neural network might be large and complex only a small part of it is used for predicting any given data instance.

The simplest way to achieve sparsity is to impose an L_1-penalty on the hidden units. Therefore, the original loss function L is modified to the regularized loss function L' as follows:

$$L' = L + \lambda \sum_{i=1}^{M} |h_i| \tag{4.8}$$

Here, M is the total number of units in the network, and h_i is the value of the ith hidden unit. Furthermore, the regularization parameter is denoted by λ. In many cases, a single *layer* of the network is regularized, so that a sparse feature representation can be extracted from the activations of that particular layer.

How does this change to the objective function affect the backpropagation algorithm? The main difference is that the loss function is aggregated not only over nodes in the output layer, but also over nodes in the hidden layer. At a fundamental level, this change does not affect the overall dynamics and principles of backpropagation. This situation is discussed in Section 3.2.7 of Chapter 3.

The backpropagation algorithm needs to be modified so that the regularization penalty contributed by a hidden unit is incorporated into the backwards gradient flow of all connections incoming into that node. Let $N(h)$ be the set of nodes reachable from any particular node h in the computational graph (including itself). Then, the gradient $\frac{\partial L}{\partial a_h}$ of the loss L also depends on the penalty contributions of the nodes in $N(h)$. Specifically, for any node h_r with pre-activation value a_{h_r}, its gradient flow $\frac{\partial L}{\partial a_{h_r}} = \delta(h_r, N(h_r))$ to the output node is increased by $\lambda \Phi'(a_{h_r}) \text{sign}(h_r)$. Here, the gradient flow $\frac{\partial L}{\partial a_{h_r}} = \delta(h_r, N(h_r)))$ is defined according to the discussion in Section 3.2.7 of Chapter 3. Consider Equation 3.25 of Chapter 3, which computes the backwards gradient flow as follows:

$$\delta(h_r, N(h_r)) = \Phi'(a_{h_r}) \sum_{h: h_r \Rightarrow h} w_{(h_r,h)} \delta(h, N(h)) \tag{4.9}$$

Here, $w_{(h_r,h)}$ is the weight of the edge from h_r to h. Immediately after making this update, the value of $\delta(h_r, N(h_r))$ is adjusted to account for the regularization term at that node as follows:

$$\delta(h_r, N(h_r)) \Leftarrow \delta(h_r, N(h_r)) + \lambda \Phi'(a_{h_r}) \cdot \text{sign}(h_r)$$

Note that the above update is based on Equation 3.26 of Chapter 3. Once the value of $\delta(h_r, N(h_r))$ is modified at a given node h_r, the changes will automatically be backpropagated to all nodes that reach h_r. This is the only change that is required in order to enforce L_1-regularization of the hidden units. In a sense, incorporating penalties on nodes in intermediate layers does not change the backpropagation algorithm in a fundamental way, except that hidden nodes are now also treated as output nodes in terms of contributing to the gradient flow.

4.5 Ensemble Methods

Ensemble methods derive their inspiration from the bias-variance trade-off. One way of reducing the error of a classifier is to find a way to reduce either its bias or the variance without affecting the other component. Ensemble methods are used commonly in machine learning, and two examples of such methods are *bagging* and *boosting*. The former is a method for variance reduction, whereas the latter is a method for bias reduction.

Most ensemble methods in neural networks are focused on variance reduction. This is because neural networks are valued for their ability to build arbitrarily complex models in which the bias is relatively low. However, operating at the complex end of the bias-variance trade-off almost always leads to higher variance, which is manifested as overfitting. Therefore, the goal of most ensemble methods in the neural network setting is variance reduction (i.e., better generalization). This section will focus on such methods.

4.5.1 Bagging and Subsampling

Imagine that you had an infinite resource of training data available to you, where you could generate as many training points as you wanted from a base distribution. How can one use this unusually generous resource of data to get rid of variance? If a sufficient number of samples is available, after all, the variance of most types of statistical estimates can by asymptotically reduced to 0.

A natural approach for reducing the variance in this case would be to repeatedly create different training data sets and predict the same test instance using these data sets. The prediction across different data sets can then be averaged to yield the final prediction. If a sufficient number of training data sets is used, the variance of the prediction will be reduced to 0, although the bias will still remain depending on the choice of model.

The approach described above can be used only when an infinite resource of data is available. However, in practice, we only have a single finite instance of the data available to us. In such cases, one obviously cannot implement the above methodology. However, it turns out that an imperfect simulation of the above methodology still has better variance characteristics than a single execution of the model on the entire training data set. The basic idea is to generate new training data sets from the single instance of the base data by sampling. The sampling can be performed with or without replacement. The predictions on a particular test instance, which are obtained from the models built with different training sets, are then averaged to create the final prediction. One can average either the real-valued predictions (e.g., probability estimates of class labels) or the discrete predictions. In the case of real-valued predictions, better results are sometimes obtained by using the median of the values.

It is common to use the softmax to yield probabilistic predictions of discrete outputs. If probabilistic predictions are averaged, it is common to average the *logarithms* of these

values. This is the equivalent of using the *geometric* means of the probabilities. For discrete predictions, arithmetically averaged voting is used. This distinction between the handling of discrete and probabilistic predictions is carried over to other types of ensemble methods that require averaging of the predictions. This is because the logarithms of the probabilities have a log-likelihood interpretation, and log-likelihoods are inherently additive.

The main difference between bagging and subsampling is in terms of whether or not replacement is used in the creation of the sampled training data sets. We summarize these methods as follows:

1. *Bagging:* In bagging, the training data is sampled with replacement. The sample size s may be different from the size of the training data size n, although it is common to set s to n. In the latter case, the resampled data will contain duplicates, and about a fraction $(1-1/n)^n \approx 1/e$ of the original data set will not be included at all. Here, the notation e denotes the base of the natural logarithm. A model is constructed on the resampled training data set, and each test instance is predicted with the resampled data. The entire process of resampling and model building is repeated m times. For a given test instance, each of these m models is applied to the test data. The predictions from different models are then averaged to yield a single robust prediction. Although it is customary to choose $s = n$ in bagging, the best results are often obtained by choosing values of s much less than n.

2. Subsampling is similar to bagging, except that the different models are constructed on the samples of the data created *without* replacement. The predictions from the different models are averaged. In this case, it is essential to choose $s < n$, because choosing $s = n$ yields the same training data set and identical results across different ensemble components.

When a sufficient training data are available, subsampling is often preferable to bagging. However, using bagging makes sense when the amount of available data is limited.

It is noteworthy that all the variance cannot be removed by using bagging or subsampling, because the different training samples will have overlaps in the included points. Therefore, the predictions of test instances from different samples will be positively correlated. The average of a set of random variables that are positively correlated will always have a variance that is proportional to the level of correlation. As a result, there will always be a residual variance in the predictions. This residual variance is a consequence of the fact that bagging and subsampling are imperfect simulations of drawing the training data from a base distribution. Nevertheless, the variance of this approach is still lower than that of constructing a single model on the entire training data set. The main challenge in directly using bagging for neural networks is that one must construct multiple training models, which is highly inefficient. However, the construction of different models can be fully parallelized, and therefore this type of setting is a perfect candidate for training on multiple GPU processors.

4.5.2 Parametric Model Selection and Averaging

One challenge in the case of neural network construction is the selection of a large number of hyperparameters like the depth of the network and the number of neurons in each layer. Furthermore, the choice of the activation function also has an effect on performance, depending on the application at hand. The presence of a large number of parameters creates problems in model construction, because the performance might be sensitive to the particular configuration used. One possibility is to hold out a portion of the training data and

try different combinations of parameters and model choices. The selection that provides the highest accuracy on the held-out portion of the training data is then used for prediction. This is, of course, the standard approach used for parameter tuning in all machine learning models, and is also referred to as *model selection*. In a sense, model selection is inherently an ensemble-centric approach, where the best out of bucket of models is selected. Therefore, the approach is also sometimes referred to as the *bucket-of-models* technique.

The main problem in deep learning settings is that the number of possible configurations is rather large. For example, one might need to select the number of layers, the number of units in each layer, and the activation function. The combination of these possibilities is rather large. Therefore, one is often forced to try only a limited number of possibilities to choose the configuration. An additional approach that can be used to reduce the variance, is to select the k best configurations and then average the predictions of these configurations. Such an approach leads to more robust predictions, especially if the configurations are very different from one another. Even though each individual configuration might be suboptimal, the overall prediction will still be quite robust. However, such an approach cannot be used in very large-scale settings because each execution might require on the order of a few weeks. Therefore, one is often reduced to leveraging the single best configuration based on the approach in Section 3.3.1 of Chapter 3. As in the case of bagging, the use of multiple configurations is often feasible only when multiple GPUs are available for training.

4.5.3 Randomized Connection Dropping

The random dropping of connections between different layers in a multilayer neural network often leads to diverse models in which different combinations of features are used to construct the hidden variables. The dropping of connections between layers does tend to create less powerful models because of the addition of constraints to the model-building process. However, since different random connections are dropped from different models, the predictions from different models are very diverse. The averaged prediction from these different models is often highly accurate. It is noteworthy that the weights of different models are not shared in this approach, which is different from another technique called *Dropout*.

Randomized connection dropping can be used for any type of predictive problem and not just classification. For example, the approach has been used for outlier detection with autoencoder ensembles [64]. As discussed in Section 2.5.4 of Chapter 2, autoencoders can be used for outlier detection by estimating the reconstruction error of each data point. The work in [64] uses multiple autoencoders with randomized connections, and then aggregates the outlier scores from these different components in order to create the score of a single data point. However, the use of the median is preferred to the mean in [64]. It has been shown in [64] that such an approach improves the overall accuracy of outlier detection. It is noteworthy that this approach might seem superficially similar to *Dropout* and *DropConnect*, although it is quite different. This is because methods like *Dropout* and *DropConnect* share weights between different ensemble components, whereas this approach does not share any weights between ensemble components.

4.5.4 Dropout

Dropout is a method that uses node sampling instead of edge sampling in order to create a neural network ensemble. If a node is dropped, then all incoming and outgoing connections from that node need to be dropped as well. The nodes are sampled only from the input and hidden layers of the network. Note that sampling the output node(s) would make it

impossible to provide a prediction and compute the loss function. In some cases, the input nodes are sampled with a different probability than the hidden nodes. Therefore, if the full neural network contains M nodes, then the total number of possible sampled networks is 2^M.

A key point that is different from the connection sampling approach discussed in the previous section is that *weights of the different sampled networks are shared*. Therefore, *Dropout* combines node sampling with weight sharing. The training process then uses a single sampled example in order to update the weights of the sampled network using back-propagation. The training process proceeds using the following steps, which are repeated again and again in order to cycle through all of the training points in the network:

1. Sample a neural network from the base network. The input nodes are each sampled with probability p_i, and the hidden nodes are each sampled with probability p_h. Furthermore, all samples are independent of one another. When a node is removed from the network, all its incident edges are removed as well.

2. Sample a single training instance or a mini-batch of training instances.

3. Update the weights of the retained edges in the network using backpropagation on the sampled training instance or the mini-batch of training instances.

It is common to exclude nodes with probability between 20% and 50%. Large learning rates are often used with momentum, which are tempered with a max-norm constraint on the weights. In other words, the L_2-norm of the weights entering each node is constrained to be no larger than a small constant such as 3 or 4.

It is noteworthy that a different neural network is used for every small mini-batch of training examples. Therefore, the number of neural networks sampled is rather large, and depends on the size of the training data set. This is different from most other ensemble methods like bagging in which the number of ensemble components is rarely larger than 25. In the *Dropout* method, thousands of neural networks are sampled with shared weights, and a tiny training data set is used to update the weights in each case. Even though a large number of neural networks is sampled, the *fraction* of neural networks sampled out of the base number of possibilities is still minuscule. Another assumption that is used in this class of neural networks is that the output is in the form of a probability. This assumption has a bearing on the way in which the predictions of the different neural networks are combined.

How can one use the ensemble of neural networks to create a prediction for an unseen test instance? One possibility is to predict the test instance using all the neural networks that were sampled, and then use the geometric mean of the probabilities that are predicted by the different networks. The geometric mean is used rather than the arithmetic mean, because the assumption is that the output of the network is a probability and the geometric mean is equivalent to averaging log-likelihoods. For example, if the neural network has k probabilistic outputs corresponding to the k classes, and the jth ensemble yields an output of $p_i^{(j)}$ for the ith class, then the ensemble estimate for the ith class is computed as follows:

$$p_i^{Ens} = \left[\prod_{j=1}^{m} p_i^{(j)}\right]^{1/m} \tag{4.10}$$

Here, m is the total number of ensemble components, which can be rather large in the case of the *Dropout* method. One problem with this estimation is that the use of geometric

means results in a situation where the probabilities over the different classes do not sum to 1. Therefore, the values of the probabilities are re-normalized so that they sum to 1:

$$p_i^{Ens} \Leftarrow \frac{p_i^{Ens}}{\sum_{i=1}^{k} p_i^{Ens}} \tag{4.11}$$

The main problem with this approach is that the number of ensemble components is too large, which makes the approach inefficient.

A key insight of the *Dropout* method is that it is not necessary to evaluate the prediction on all ensemble components. Rather, one can perform forward propagation on only the base network (with no dropping) after re-scaling the weights. The basic idea is to multiply the weights going out of each unit with the probability of sampling that unit. By using this approach, the expected output of that unit from a sampled network is captured. This rule is referred to as the *weight scaling inference rule*. Using this rule also ensures that the input going into a unit is also the same as the expected input that would occur in a sampled network.

The weight scaling inference rule is exact for many types of networks with linear activations, although the rule is not exactly true for networks with nonlinearities. In practice, the rule tends to work well across a broad variety of networks. Since most practical neural networks have nonlinear activations, the weight scaling inference rule of *Dropout* should be viewed as a heuristic rather than a theoretically justified result. *Dropout* has been used with a wide variety of models that use a distributed representation; it has been used with feed-forward networks, Restricted Boltzmann machines, and recurrent neural networks.

The main effect of *Dropout* is to incorporate regularization into the learning procedure. By dropping both input units and hidden units, *Dropout* effectively incorporates noise into both the input data and the hidden representations. The nature of this noise can be viewed as a kind of masking noise in which some inputs and hidden units are set to 0. Noise addition is a form of regularization. It has been shown in the original paper [467] on *Dropout* that this approach works better than other regularizers such as weight decay. *Dropout* prevents a phenomenon referred to as *feature co-adaptation* from occurring between hidden units. Since the effect of *Dropout* is a masking noise that removes some of the hidden units, this approach forces a certain level of redundancy between the features learned at the different hidden units. This type of redundancy leads to increased robustness.

Dropout is efficient because each of the sampled subnetworks is trained with a small set of sampled instances. Therefore, only the work of sampling the hidden units needs to be done additionally. However, since *Dropout* is a regularization method, it reduces the expressive power of the network. Therefore, one needs to use larger models and more units in order to gain the full advantages of *Dropout*. This results in a hidden computational overhead. Furthermore, if the original training data set is already large enough to reduce the likelihood of overfitting, the additional computational advantages of *Dropout* may be small but still perceptible. For example, many of the convolutional neural networks trained on large data repositories like *ImageNet* [255] report consistently improved results of about 2% with *Dropout*. A variation of *Dropout* is *DropConnect*, which applies a similar approach to the weights rather than to the neural network nodes [511].

A Note on Feature Co-adaptation

In order to understand why *Dropout* works, it is useful to understand the notion of feature co-adaptation. Ideally, it is useful for the hidden layers of the neural network to create features that reflect important classification characteristics of the input without having complex

dependencies on other features, unless these other features are truly useful. To understand this point, consider a situation in which all edges incident on 50% of the nodes in each layer are fixed at their initial random values, and are not *updated* during backpropagation (even though all gradients are *computed* in the normal fashion). Interestingly, even in this case, it will often be possible for the neural network to provide reasonably good results by adapting the other weights and features to the effect of these randomly fixed subsets of weights (and corresponding activations). Of course, this is not a desirable situation because the goal of features working together is to combine the powers held by each essential feature rather than merely having some features adjust to the detrimental effects of others. Even in the normal training of a neural network (where all weights are updated), this type of co-adaptation can occur. For example, if the updates in some parts of the neural network are not fast enough, some of the features will not be useful and other features will adapt to these less-than-useful features. This situation is very likely in neural network training, because different parts of the neural network do tend to learn at different rates. An even more troubling scenario arises when the co-adapted features work well in predicting training points by picking up on complex dependencies in the training points, which do not generalize well to out-of-sample test points. *Dropout* prevents this type of co-adaptation by forcing the neural network to make predictions using only a subset of the inputs and activations. This forces the network to be able to make predictions with a certain level of redundancy while also encouraging smaller subsets of learned features to have predictive power. In other words, co-adaptation occurs only when it is truly essential for modeling instead of learning random nuances of the training data. This is, of course, a form of regularization. Furthermore, by learning redundant features, *Dropout* averages over the predictions of redundant features, which is similar to what is done in bagging.

4.5.5 Data Perturbation Ensembles

Most of the ensemble techniques discussed so far are either sampling-based ensembles or model-centric ensembles. *Dropout* can be considered an ensemble that adds noise to the data in an indirect way. It is also possible to use explicit data perturbation methods.

In the simplest case, a small amount of noise can be added to the input data, and the weights can be learned on the perturbed data. This process can be repeated with multiple such additions, and the predictions of the test point from different ensemble components can be averaged. This type of approach is a generic ensemble method, which is not specific to neural networks. As discussed in Section 4.10, this approach is used commonly in the unsupervised setting with *de-noising autoencoders*.

It is also possible to add noise to the hidden layer. However, in this case, the noise has to be carefully calibrated [382]. It is noteworthy that the *Dropout* method indirectly adds noise to the hidden layer by dropping nodes randomly. A dropped node is similar to masking noise in which the activation of that node is set to 0.

One can also perform other types of data set augmentation. For example, an image instance can be rotated or translated in order to add to the data set. Carefully designed data augmentation schemes can often greatly improve the accuracy of a learner by increasing its generalization power. However, strictly speaking such schemes are not perturbation schemes because the augmented examples are created with a calibrated procedure and an understanding of the domain at hand. Such methods are used commonly in convolutional neural networks (cf. Section 8.3.4 of Chapter 8).

4.6 Early Stopping

Neural networks are trained using variations of gradient-descent methods. In most optimization models, gradient-descent methods are executed to convergence. However, executing gradient descent to convergence optimizes the loss on the training data, but not necessarily on the out-of-sample test data. This is because the final few steps often overfit to the specific nuances of the training data, which might not generalize well to the test data.

A natural solution to this dilemma is to use *early stopping*. In this method, a portion of the training data is held out as a validation set. The backpropagation-based training is only applied to the portion of the training data that does not include the validation set. At the same time, the error of the model on the validation set is continuously monitored. At some point, this error begins to rise on the validation set, even though it continues to reduce on the training set. This is the point at which further training causes overfitting. Therefore, this point can be chosen for termination. It is important to keep track of the best solution achieved so far in the learning process (as computed on the validation data). This is because one does not perform early stopping after tiny increases in the out-of-sample error (which might be caused by noisy variations), but it is advisable to continue to train to check if the error continues to rise. In other words, the termination point is chosen in hindsight after the error on the validation set continues to rise, and all hope is lost of improving the error performance on the validation set.

Even though the removal of the validation set does lose some training points, the effect of data loss is often quite small. This is because neural networks are often trained on extremely large data sets of the order of tens of millions of points. A validation set does not need a large number of points. For example, the use of a sample of 10,000 points for validation might be tiny compared to the full data size. Although one can often include the validation set within the training data to retrain the network for the same number of steps (as was obtained at the early stopping point), the effect of this approach can sometimes be unpredictable. It can also lead to a doubling of computational costs, because the neural network needs to be trained all over again.

One advantage of early stopping is that it can be easily added to neural network training without significantly changing the training procedure. Furthermore, methods like weight decay require us to try different values of the regularization parameter, λ, which can be expensive. Because of the ease in combining it with existing algorithms, early stopping can be used in combination with other regularizers in a relatively straightforward way. Therefore, early stopping is almost always used, because one does not lose much by adding it to the learning procedure.

One can view early stopping as a kind of constraint on the optimization process. By restricting the number of steps in the gradient descent, one is effectively restricting the distance of the final solution from the initialization point. Adding constraints to the model of a machine learning problem is often a form of regularization.

4.6.1 Understanding Early Stopping from the Variance Perspective

One way of understanding the bias-variance trade-off is that the true loss function of an optimization problem can only be constructed if we have infinite data. If we have a finite amount of data, the loss function constructed from the training data does not reflect the true loss function. Illustrative examples of the contours of the true loss function and its shifted counterpart on the training data are illustrated in Figure 4.6. This shifting is an

indirect manifestation of the variance in prediction created by a particular training data set. Different training data sets will shift the loss function in different and unpredictable ways.

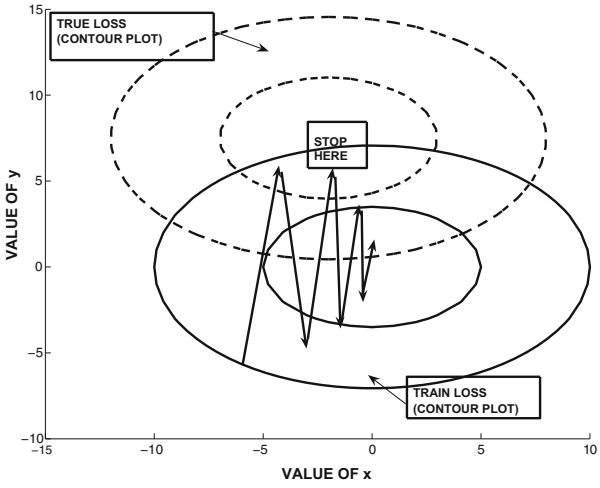


Figure 4.6: Shift in loss function caused by variance effects and the effect of early stopping. Because of the differences in the true loss function and that on the training data, the error will begin to rise if gradient descent is continued beyond a certain point. Here, we have shown a similar shape of the true and training loss functions for simplicity, although this might not be the case in practice.

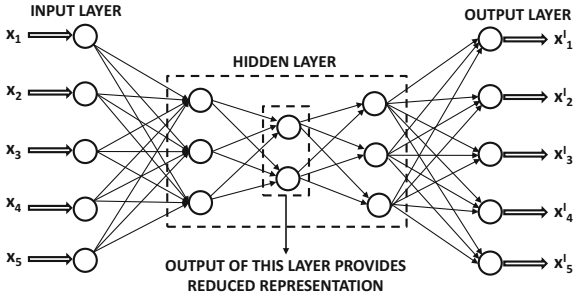
Unfortunately, the learning procedure can perform the gradient-descent only on the loss function defined on the training data set, because the true loss function is unknown. However, if the training data is representative of the true loss function, the optimum solutions in the two cases will be reasonably close as shown in Figure 4.6. As discussed in Chapter 3, most gradient-descent procedures take a circuitous and oscillatory route to the optimal solution. During the final stage of convergence to the optimal solution (on the training data), the gradient descent will often encounter better solutions with respect to the true loss function before it converges to the best solution with respect to the training data. These solutions will be detected by the improved accuracy on the validation set, and therefore provide good termination points. An example of a good early stopping point is shown in Figure 4.6.

4.7 Unsupervised Pretraining

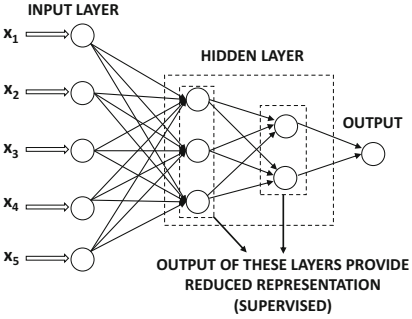
Deep networks are inherently hard to train because of a number of different characteristics discussed in the previous chapter. One issue is the exploding and vanishing gradient problem, because of which the different layers of the neural network do not get trained at the same rate. The multiple layers of the neural network cause distortions in the gradient, which make them hard to train.

Although the depth of the neural network causes challenges, the problems associated with depth are also heavily dependent on how the network is initialized. A good initialization point can often solve many of the problems associated with reaching good solutions. A ground-breaking break-through in this context was the use of unsupervised pretraining

in order to provide robust initializations [196]. This initialization is achieved by training the network greedily in layer-wise fashion. The approach was originally proposed in the context of deep belief networks, but it was later extended to other types of models such as autoencoders [386, 506]. In this chapter, we will study the autoencoder approach because of its simplicity. First, we will start with the dimensionality reduction application, because the application is unsupervised and it is easy to show how to use unsupervised pretraining in this case. However, unsupervised pretraining can also be used for supervised applications like classification with minor modifications.

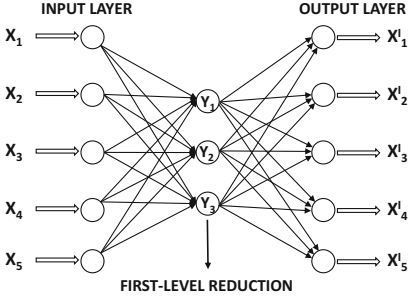


(a) Multilayer autoencoder

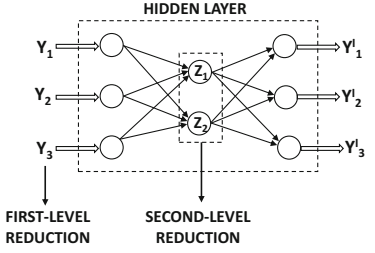


(b) Multilayer classifier

Figure 4.7: Both the multilayer classifier and the multilayer autoencoder use a similar pretraining procedure.



(a) Pretraining first-level reduction and outer weights



(b) Pretraining second-level reduction and inner weights

Figure 4.8: Pretraining a neural network

In pretraining, a greedy approach is used to train the network one layer at a time by learning the weights of the outer hidden layers first and then learning the weights of the inner hidden layers. The resulting weights are used as starting points for a final phase of traditional neural network backpropagation in order to fine-tune them.

Consider the autoencoder and classifier architectures shown in Figure 4.7. Since these architectures have multiple layers, randomized initialization can sometimes cause challenges. However, it is possible to create a good initialization by setting the initial weights layer by layer in a greedy fashion. First, we describe the process in the context of the autoencoder shown in Figure 4.7(a), although an almost identical procedure is relevant to the classifier of Figure 4.7(b). We have intentionally chosen neural architectures in the two cases so that the hidden layers have similar numbers of nodes.

The pretraining process is shown in Figure 4.8. The basic idea is to assume that the two (symmetric) outer hidden layers contain a first-level reduced representation of larger dimensionality, and the inner hidden layer contains a second-level reduced representation of smaller dimensionality. Therefore, the first step is to learn the first-level reduced representation and the corresponding weights associated with the outer hidden layers using the simplified network of Figure 4.8(a). In this network, the middle hidden layer is missing and the two outer hidden layers are collapsed into a single hidden layer. The assumption is that the two outer hidden layers are related to one another in a symmetric way like a smaller autoencoder. In the second step, the reduced representation in the first step is used to learn the second-level reduced representation (and weights) of the inner hidden layers. Therefore, the inner portion of the neural network is treated as a smaller autoencoder in its own right. Since each of these pretrained subnetworks is much smaller, the weights can be learned more easily. This initial set of weights is then used to train the entire neural network with backpropagation. Note that this process can be performed in layerwise fashion for a deep neural network containing any number of hidden layers.

So far, we have only discussed how we can use unsupervised pretraining for unsupervised applications. A natural question arises as to how one can use pretraining for supervised applications. Consider a multilayer classification architecture with a single output layer and k hidden layers. During the pretraining stage, the output layer is removed, and the representation of the final hidden layer is learned in an unsupervised way. This is achieved by creating an autoencoder with $2 \cdot k - 1$ hidden layers, where the middle layer is the final hidden layer of the supervised setting. For example, the relevant autoencoder for Figure 4.7(b) is shown in Figure 4.7(a). Therefore, an additional $(k - 1)$ hidden layers are added, each of which has a symmetric counterpart in the original network. This network is trained in exactly the same layer-wise fashion as discussed above for the autoencoder architecture. The weights of only the encoder portion of this autoencoder are used for initialization of the weights entering into all hidden layers. The weights between the final hidden layer and the output layer can also be initialized by treating the final hidden layer and output nodes as a single-layer network. This single-layer network is fed with the reduced representations of the final hidden layer (based on the autoencoder learned in pretraining). After the weights of all the layers have been learned, the output nodes are re-attached to the final hidden layer. The backpropagation algorithm is applied to this initialized network in order to fine-tune the weights from the pretrained stage. Note that this approach learns all the initial hidden representations in an unsupervised way, and only the weights entering into the output layer are initialized using the labels. Therefore, the pretraining can still be considered to be largely unsupervised.

During the early years, pretraining was often seen as a more stable way to train a deep network in which the different layers have a better chance of being initialized in an equally effective way. Although this issue does play a role in explaining the improvements of pretraining, the problem is often manifested as overfitting. As discussed in Chapter 3, the (finally converged) weights in the early layers may not change much from their random initializations, when the network exhibits the vanishing gradient problem. Even when the connection weights in the first few layers are random (as a result of poor training), it is possible for the later layers to adapt their weights sufficiently so as to give zero error on the *training* data. In this case, the random connections in the early layers provide near-random transformations to the later layers, but the later layers are still able to overfit to these features in order to provide very low training error. In other words, the features in later layers *adapt* to those in early layers as a result of training inefficiencies. Any kind of feature co-adaptation caused by training inefficiencies almost always leads to overfitting. Therefore, when the approach is applied to unseen test data, the overfitting becomes apparent because the various layers are not specifically adapted to these unseen test instances. In this sense, pretraining is an unusual form of regularization.

Incidentally, unsupervised pretraining helps even in cases where the amount of training data is very large. It is likely that this behavior is caused by the fact that pretraining helps in issues beyond model generalization. One evidence of this fact is that in larger data sets, even the error on the training data seems to be high, when methods like pretraining are not used. In these cases, the weights of the early layers often do not change much from their initializations, and one is using only a small number of later layers on a random transformation of the data (defined by the random initialization of the early layers). As a result, the trained portion of the network is rather shallow, with some additional loss caused by the random transformation. In such cases, pretraining also helps a model realize the full benefits of depth, thereby facilitating the improvement of prediction accuracy on larger data sets.

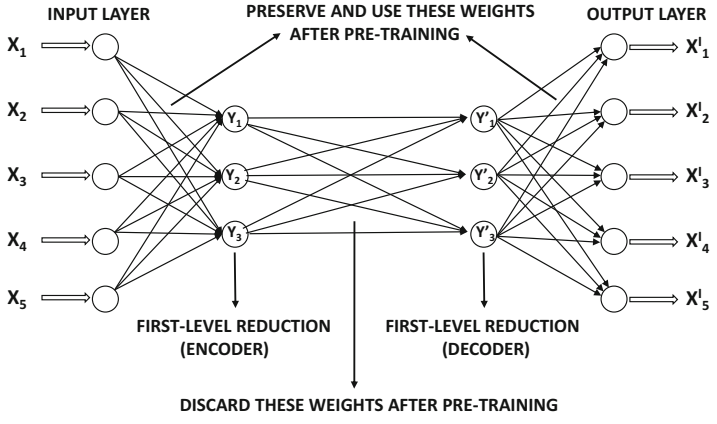


Figure 4.9: This architecture allows the first-level representations in the encoder and decoder to be significantly different. It is helpful to compare this architecture with that in Figure 4.8(a).

Another way of understanding pretraining is that it provides insights into the repeated patterns in the data, which are the features learned from the training data points. For example, an autoencoder might learn that many digits have loops in them, and certain

digits have strokes that are curved in a particular way. The decoder reconstructs the digits by putting together these frequent shapes. However, these shapes also have discriminative power with respect to recognizing digits. Expressing the data in terms of a few features then helps in recognizing how these features are related to the class labels. This principle is summarized by Geoff Hinton [192] in the context of image classification as follows: "*To recognize shapes, first learn to generate images.*" This type of regularization preconditions the training process in a semantically relevant region of the parameter space, where several important features have already been learned, and further training can fine-tune and combine them for prediction.

4.7.1 Variations of Unsupervised Pretraining

There are many different ways in which one can introduce variations to the procedure of unsupervised pretraining. For example, multiple layers can be trained at one time instead of performing pretraining only one layer at a time. A particular case in point is *VGG* (cf. Section 8.4.3 of Chapter 8) in which as many as eleven layers of an even deeper architecture were trained together. Indeed, there are some advantages in grouping as many layers as possible within the pretraining because a (successful) training procedure with larger pieces of the neural network leads to more powerful initializations. On the other hand, grouping too many layers together within each pretraining component can lead to problems (such as the vanishing and exploding gradient problems) within each component.

A second point is that the pretraining procedure of Figure 4.8 assumes that the autoencoder works in a completely symmetric way in which the reduction in the kth layer of the encoder is approximately similar to the reduction in its mirror layer in the decoder. This might be a restrictive assumption in practice, if different types of activation functions are used in different layers. For example, a sigmoid activation function in a particular layer of the encoder will create only nonnegative values, whereas a tanh activation in the matching layer of the decoder might create both positive and negative values. Another approach is to use a relaxed pretraining architecture in which we learn separate reductions for the kth level reduction in the encoder and its mirror image in the decoder. This allows the corresponding reductions in the encoder and the decoder to be different. An additional layer of weights must be added between the two layers to allow for the differences between the two reductions. This additional layer of weights is discarded after the reduction, and only the encoder-decoder weights are preserved. The only location at which an additional set of weights is not used for pretraining is in the innermost reduction, which proceeds in a similar manner to that discussed in the earlier section (cf. Figure 4.8(b)). An example of such an architecture for the first-level reduction of Figure 4.8(a) is shown in Figure 4.9. Note that the first-level representations for the encoder and decoder layers can be quite different in this case, which provides some flexibility during the pretraining process. When the approach is used for classification, only the weights in the encoder can be used, and the final reduced code can be capped with a classification layer for learning.

4.7.2 What About Supervised Pretraining?

So far, we have only discussed *unsupervised* pretraining, whether the base application is supervised or unsupervised. Even in the case where the base application is supervised, the initialization was done using an unsupervised autoencoder architecture. Although it is possible to perform supervised pretraining as well, an interesting and surprising result is that supervised pretraining does not seem to give as good results as unsupervised pretraining

in at least some settings [113, 31]. This does not mean that supervised pretraining is *never* helpful. Indeed, there are cases of networks in which it is hard to train the network itself because of its depth. For example, networks with hundreds of layers are extremely hard to train because of issues associated with convergence and other problems. In such cases, even the error *on the training data* is high, which means that one is unable to make the training algorithm work. *This is a different problem from that of model generalization.* Aside from supervised pretraining, many techniques such as the construction of *highway networks* [161, 470], *gating networks* [204], and *residual networks* [184], can address many of these problems. However, these solutions do not specifically address overfitting, whereas unsupervised pretraining seems to hedge its bets in addressing both issues in at least some types of networks.

In supervised pretraining [31], the autoencoder architecture is not used for learning the weights of connections incident on the hidden layer. In the first iteration, the constructed network contains only the first hidden layer, which is connected to all nodes in the output layer. This step learns the weights of the connections from the input to hidden layer, although the weights of the output layer are discarded. Subsequently, the outputs of the first hidden layer are used as the new representations of the training points. Then, we create another neural network containing the first and second hidden layers and the output layer. The first hidden layer is now treated as an input layer with its inputs as the transformed representations of the training points learned in the previous iteration. These are then used to learn the next layer of weights and their hidden representations. This approach is repeated all the way to the final layer. Although this approach does provide improvements over an approach that does not use pretraining, it does not seem to work as well as unsupervised pretraining in at least some settings. The main difference in performance is on the *generalization error* on unseen test data, whereas the errors on the training data are often similar [31]. This is a near-certain sign of differential levels of overfitting of different methods.

Why does supervised pretraining not help as much as unsupervised pretraining in many settings? A key problem of supervised pretraining is that it is a bit too greedy and the early layers are initialized to representations that are very directly related to the outputs. As a result, the full advantages of depth are not exploited. This is a different type of overfitting. An important explanation for the success of unsupervised pretraining is that the learned representations are often related to the class labels in a gentle way; as a result, further learning is able to isolate and fine-tune the important characteristics of these representations. Therefore, one can view pretraining as an unusual form of *semi-supervised learning* as well, which forces the initial representations of the hidden layers to lie on the low-dimensional manifolds of data instances. The secret to the success of pretraining is that more features on these manifolds are predictive of classification accuracy than the features corresponding to random regions of the data space. After all, class distributions vary smoothly over the underlying data manifolds. The locations of data points on these manifolds are therefore good features in predicting class distributions. Therefore, the final phase of learning only has to fine-tune and enhance these features.

Are there cases in which unsupervised pretraining does not help? The work in [31] provides examples in which the manifold corresponding to the data distribution does not seem to exhibit too much relationship with the target. This tends to occur more often in regression as compared to classification. In such cases, it was shown that adding some supervision to pretraining can indeed help. The first layer of weights (between input and first hidden layer) is trained using a combination of gradient updates from autoencoder-like reconstruction as well as greedy supervised pretraining. Thus, the learning of the weights of the first

layer is partially supervised. Subsequent layers are trained using the autoencoder approach only. The inclusion of supervision in the first level of weights automatically incorporates some level of supervision into the inner layers as well. This approach is used for initializing the weights of the neural network. These weights are then fine tuned using fully supervised backpropagation over the entire network.

4.8 Continuation and Curriculum Learning

The discussions in the previous and current chapter show that the learning of neural network parameters is inherently a complex optimization problem, in which the loss function has a complex topological shape. Furthermore, the loss function on the training data is not exactly the same as the true loss function, which leads to spurious minima. These minima are spurious because they might be near optimal minima on the training data, but they might not be minima at all on unseen test instances. In many cases, optimizing a complex loss function tends to lead to such solutions with little generalization power.

The experience with pretraining shows that simplifying the optimization problem (or providing simple greedy solutions without too much optimization) can often precondition the solution towards the basins of better optima on the test data. In other words, instead of trying to solve a complex problem in one shot, one should first try to solve simplifications, and gradually work one's way towards complex solutions. Two such notions are those of *continuation* and *curriculum* learning:

1. *Continuation learning:* In continuation learning, one starts with a simplified version of the optimization problem and solves it. Starting with this solution, one continues to a more complex refinement of the optimization problem and updates the solution. This process is repeated until the complex optimization problem is solved. Thus, continuation learning leverages a model-centric view of working from simpler to complex problems. For example, if one has a loss function with many local optima, one can smooth it to a loss function with a single global optimum and find the optimal solution. Then, one can gradually work with better and better approximations (with increased complexity) until the exact loss function is used.

2. *Curriculum learning:* In curriculum learning, one starts by training the model on simpler data instances, and then gradually adds more difficult instances to the training data. Therefore, curriculum learning leverages a data-centric view of working from the simple to the complex, whereas continuation methods leverage a model-centric view.

A different view of curriculum and continuation learning may be obtained by examining how humans naturally learn tasks. Humans often learn simple concepts first and then move to the complex. The training of a child is often created using such a *curriculum* in order to accelerate learning. This principle also seems to work well in machine learning. In the following, we will examine both continuation and curriculum learning.

4.8.1 Continuation Learning

In continuation learning, one designs a series of loss functions $L_1 \ldots L_r$, in which the difficulty in optimizing this sequence of loss functions grows from the easy to the difficult. In other words, each L_{i+1} is more difficult to optimize than L_i. All the optimization problems are defined on the same set of parameters, because they are defined on the same neural network. The smoothing of a loss function is a form of regularization. One can view each

L_i as a smoothed version of L_{i+1}. Solving each L_i brings the solution closer to the basin of optimal solutions from the point of view of generalization error.

Continuation loss functions are often constructed by using *blurring*. The idea is to compute the loss function at sampled points in the vicinity of a given point, and then average these values in order to create the new loss function. For example, one could use a normal distribution with standard deviation σ_i for computing the ith loss function L_i. One can view this approach as a type of noise addition to the loss function, which is also a form of regularization. The amount of blurring depends on the size of the locality used for blurring, which is defined by σ_i. If the value of σ_i is set to be too large, then the cost will be very similar at all points, and the loss function will not retain sufficient details about the objective. However, it will often be very simple to optimize. On the other hand, setting σ_i to 0 will retain all the details in the loss function. Therefore, the natural solution is to start with large values of σ_i and then reduce the value over successive loss functions. One can view this approach as that of using an increased amount of noise for regularization in the early iterations, and then reducing the level of regularization as the algorithm nears an attractive solution. Such tricks of adding a varying amount of calibrated noise to enable the avoidance of local optima is a recurring theme in many optimization techniques such as *simulated annealing* [244]. The main problem with continuation methods is that they are expensive due to the need to optimize a series of loss functions.

4.8.2 Curriculum Learning

Curriculum learning methods take a *data-centric* view of the goals that are achieved by the *model-centric* continuation learning methods. The main hypothesis is that different training data sets present different levels of difficulty to a learner. In curriculum methods, easy examples are first presented to the learner. One possible way of defining a difficult example is as one that falls on the wrong side of a decision boundary with a perceptron or an SVM. There are other possibilities, such as the use of a Bayes classifier. The basic idea is that the difficult examples are often noisy or they represent exceptional patterns that confuse the learner. Therefore, it is inadvisable to start training with such examples.

In other words, the initial iterations of stochastic gradient descent use only the easy examples to "pretrain" the learner towards a reasonable parameter setting. Subsequently, difficult examples are included with the easy examples in later iterations. It is important to include both easy and difficult examples in the later phases, or else the learner will overfit to only the difficult examples. In many cases, the difficult examples might be exceptional patterns in particular regions of the space, or they might even be noise. If only the difficult examples are presented to the learner in later phases, the overall accuracy will not be good. The best results are often obtained by using a random mixture of simple and difficult examples in later phases. The proportion of difficult examples are increased over the course of the curriculum until the input represents the true data distribution. This type of *stochastic curriculum* has been shown to be an effective approach.

4.9 Parameter Sharing

A natural form of regularization that reduces the parameter footprint of the model is the sharing of parameters across different connections. Often, this type of parameter sharing is enabled by domain-specific insights. The main insight required to share parameters is that the function computed at two nodes should be related in some way. This type of insight

can be obtained when one has a good idea of how a particular computational node relates to the input data. Examples of such parameter-sharing methods are as follows:

1. *Sharing weights in autoencoders:* The symmetric weights in the encoder and decoder portion of the autoencoder are often shared. Although an autoencoder will work whether or not the weights are shared, doing so improves the regularization properties of the algorithm. In a single-layer autoencoder with linear activation, weight sharing forces orthogonality among the different hidden components of the weight matrix. This provides the same reduction as singular value decomposition.

2. *Recurrent neural networks:* These networks are often used for modeling sequential data, such as time-series, biological sequences, and text. The last of these is the most commonly used application of recurrent neural networks. In recurrent neural networks, a time-layered representation of the network is created in which the neural network is replicated across layers associated with time stamps. Since each time stamp is assumed to use the same model, the parameters are shared between different layers. Recurrent neural networks are discussed in detail in Chapter 7.

3. *Convolutional neural networks:* Convolutional neural networks are used for image recognition and prediction. Correspondingly, the inputs of the network are arranged into a rectangular grid pattern, along with all the layers of the network. Furthermore, the weights across contiguous patches of the network are typically shared. The basic idea is that a rectangular patch of the image corresponds to a portion of the visual field, and it should be interpreted in the same way no matter where it is located. In other words, a carrot means the same thing whether it is at the left or the right of the image. In essence, these methods use semantic insights about the data to reduce the parameter footprint, share weights, and sparsify the connections. Convolutional neural networks are discussed in Chapter 8.

In many of these cases, it is evident that parameter sharing is enabled by the use of domain-specific insights about the training data as well as a good understanding of how the computed function at a node relates to the training data. The modifications to the backpropagation algorithm required for enabling weight sharing are discussed in Section 3.2.9 of Chapter 3.

An additional type of weight sharing is *soft weight sharing* [360]. In soft weight sharing, the parameters are not completely tied, but a penalty is associated with them being different. For example, if one expects the weights w_i and w_j to be similar, the penalty $\lambda(w_i - w_j)^2/2$ might be added to the loss function. In such a case, the quantity $\alpha\lambda(w_j - w_i)$ might be added to the update of w_i, and the quantity $\alpha\lambda(w_i - w_j)$ might be added to the update of w_j. Here, α is the learning rate. These types of changes to the updates tend to pull the weights towards each other.

4.10 Regularization in Unsupervised Applications

Although overfitting does occur in unsupervised applications, it is often less of a problem. In classification, one is trying to learn a single bit of information associated with each example, and therefore using more parameters than the number of examples can cause overfitting. This is not quite the case in unsupervised applications in which a single training example may contain many more bits of information corresponding to the different dimensions. In general, the number of bits of information will depend on the intrinsic dimensionality of the

data set. Therefore, one tends to hear fewer complaints about overfitting in unsupervised applications.

Nevertheless, there are many unsupervised settings in which it is beneficial to use regularization. A common case is one in which we have an *overcomplete* autoencoder, in which the number of hidden units is greater than the number of input units. An important goal of regularization in unsupervised applications is to impose some kind of structure on the learned representations. This approach to regularization can have different application-specific benefits like creating sparse representations or in providing the ability to clean corrupted data. As in the case of supervised models, one can use semantic insights about a problem domain in order to force a solution to have specific types of desired properties. This section will show how different types of penalties and constraints on the hidden units can create hidden/reconstructed representations with useful properties.

4.10.1 Value-Based Penalization: Sparse Autoencoders

The penalizing of sparse hidden units has unsupervised applications such as *sparse autoencoders*. Sparse autoencoders contain a much larger number of hidden units in each layer as compared to the number of input units. However, the values of the hidden units are encouraged to be 0s by either explicit penalization or by constraints. As a result, most of the values in the hidden units will be 0s at convergence. One possible approach is to impose an L_1-penalty on the hidden units in order to create sparse representations. The gradient-descent approach with L_1-penalties on the hidden units is discussed in Section 4.4.4. It is also noteworthy that the use of L_1-regularization seems to be somewhat unusual in the autoencoder literature (although there is no reason not to use it). Other constraint-based methods exist, such as allowing only the top-k hidden units to be activated. In most of these cases, the constraints are chosen in such a way that the backpropagation approach can be modified in a reasonable way. For example, if only the top-k units are selected for activation, then the gradient flows are allowed to backpropagate only through these chosen units. Constraint-based techniques are simply hard variations of penalty-based methods. More details are provided on some of these learning methods in Section 2.5.5.1 of Chapter 2.

4.10.2 Noise Injection: De-noising Autoencoders

As discussed in Section 4.4.1, noise injection is a form of penalty-based regularization of the weights. The use of Gaussian noise in the input is roughly equal to L_2-regularization in single-layer networks with linear activation. The de-noising autoencoder is based on noise injection rather than penalization of the weights or hidden units. However, the goal of the de-noising autoencoder is to reconstruct good examples from corrupted training data. Therefore, the type of noise should be calibrated to the nature of the input. Several different types of noise can be added:

1. *Gaussian noise:* This type of noise is appropriate for real-valued inputs. The added noise has zero mean and variance $\lambda > 0$ for each input. Here, λ is the regularization parameter.

2. *Masking noise:* The basic idea is to set a fraction f of the inputs to zeros in order to corrupt the inputs. This type of approach is particularly useful when working with binary inputs.

3. *Salt-and-pepper noise:* In this case, a fraction f of the inputs are set to either their minimum or maximum possible values according to a fair coin flip. The approach is typically used for binary inputs, for which the minimum and maximum values are 0 and 1, respectively.

De-noising autoencoders are useful when dealing with data that is corrupted. Therefore, the main application of such autoencoders is to reconstruct corrupted data. The inputs to the autoencoder are corrupted training records, and the outputs are the uncorrupted data records. As a result, the autoencoder learns to recognize the fact that the input is corrupted, and the true representation of the input needs to be reconstructed. Therefore, even if there is corruption in the test data (as a result of application-specific reasons), the approach is able to reconstruct clean versions of the test data. Note that the noise in the training data is explicitly added, whereas that in the test data is already present as a result of various application-specific reasons. For example, as shown in the top portion of Figure 4.10, one can use the approach to removing blurring or other noise from images. The nature of the noise added to the input training data should be based on insights about the type of corruption present in the test data. Therefore, one does require uncorrupted examples of the training data for best performance. In most domains, this is not very difficult to achieve. For example, if the goal is to remove noise from images, the training data might contain high-quality images as the output and artificially blurred images as the input. It is common for the de-noising autoencoder to be overcomplete, when it is used for reconstruction from corrupted data. However, this choice also depends on the nature of the input and the amount of noise added. Aside from its use for reconstructing inputs, the addition of noise is also an excellent regularizer that tends to make the approach work better for out-of-sample inputs even when the autoencoder is undercomplete.

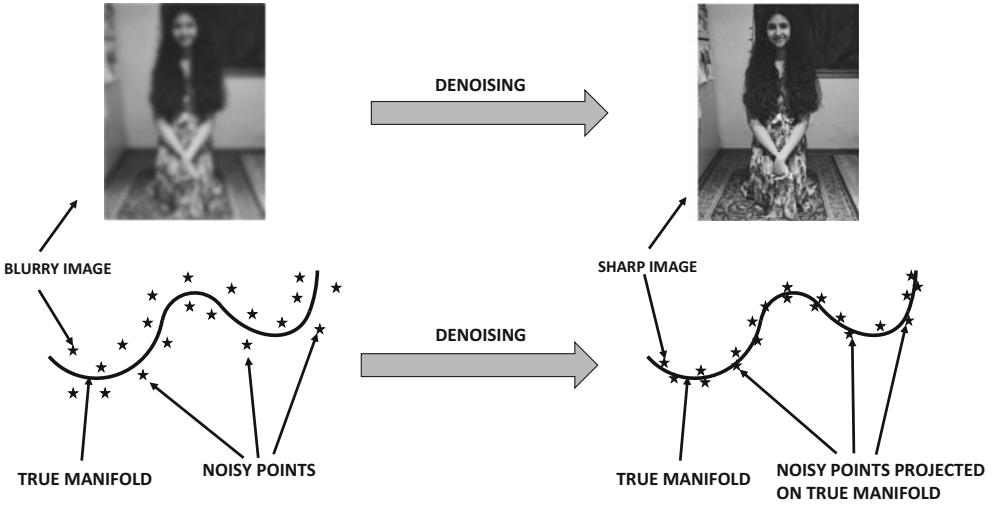


Figure 4.10: The de-noising autoencoder

The way in which the de-noising autoencoder works is that it uses the noise in the input data to learn the true manifold on which the data is embedded. Each corrupted point is projected to its "closest" matching point on the true manifold of the data distribution. The closest matching point is the expected position on the manifold from which the model predicts that the noisy point has originated. This projection is shown in the bottom portion

of Figure 4.10. The true manifold is a more concise representation of the data as compared to the noisy data, and this conciseness is a result of the regularization inherent in the addition of noise to the input. All forms of regularization tend to increase the conciseness of the underlying model.

4.10.3 Gradient-Based Penalization: Contractive Autoencoders

As in the case of the de-noising autoencoder, the hidden representation of the contractive autoencoder is often overcomplete, because the number of hidden units is greater than the number of input units. A contractive autoencoder is a heavily regularized encoder in which we do not want the hidden representation to change very significantly with small changes in input values. Obviously, this will also result in an output that is less sensitive to the input. Trying to create an autoencoder in which the output is less sensitive to changes in the input seems like an odd goal at first sight. After all, an autoencoder is supposed to reconstruct the data exactly. Therefore, the goals of regularization seem to be completely at odds with those of the contractive regularization portion of the loss function.

A key point is that contractive encoders are designed to be robust only to *small* changes in the input data. Furthermore, they tend to be insensitive to those changes that are inconsistent with the manifold structure of the data. In other words, if one makes a small change to the input that does not lie on the manifold structure of the input data, the contractive autoencoder will tend to damp the change in the reconstructed representation. Here, it is important to understand that the vast majority of (randomly chosen) directions in high-dimensional input data (with a much lower-dimensional manifold) tend to be approximately orthogonal to the manifold structure, which has the effect of changing the components of the change on the manifold structure. The damping of the changes in the reconstructive representation based on the local manifold structure is also referred to as the *contractive* property of the autoencoder. As a result, contractive autoencoders tend to remove noise from the input data (like de-noising autoencoders), although the mechanism for doing this is different from that of de-noising autoencoders. As we will see later, contractive autoencoders penalize the gradients of the hidden values with respect to the inputs. When the hidden values have low gradients with respect to the inputs, it means that they are not very sensitive to small changes in the inputs (although larger changes or changes parallel to manifold structure will tend to change the gradients).

For ease in discussion, we will discuss the case where the contractive autoencoder has a single hidden layer. The generalization to multiple hidden layers is straightforward. Let $h_1 \ldots h_k$ be the values of the k hidden units for the input variables $x_1 \ldots x_d$. Let the reconstructed values in the output layer be given by $\hat{x}_1 \ldots \hat{x}_d$. Then, the objective function is given by the weighted sum of the reconstruction loss and the regularization term. The loss L for a single training instance is given by the following:

$$L = \sum_{i=1}^{d} (x_i - \hat{x}_i)^2 \tag{4.12}$$

The regularization term is constructed by using the sum of the squares of the partial derivatives of all hidden variables with respect to all input dimensions. For a problem with k hidden units denoted by $h_1 \ldots h_k$, the regularization term R can be written as follows:

$$R = \frac{1}{2} \sum_{i=1}^{d} \sum_{j=1}^{k} \left(\frac{\partial h_j}{\partial x_i} \right)^2 \tag{4.13}$$

In the original paper [397], the sigmoid nonlinearity is used in the hidden layer, in which case the following can be shown (cf. Section 3.2.5 of Chapter 3):

$$\frac{\partial h_j}{\partial x_i} = w_{ij} h_j (1 - h_j) \ \forall i, j \tag{4.14}$$

Here, w_{ij} is the weight of the input unit i to the hidden unit j.

The overall objective function for a single training instance is given by a weighted sum of the loss and the regularization terms.

$$\begin{aligned} J &= L + \lambda \cdot R \\ &= \sum_{i=1}^{d} (x_i - \hat{x}_i)^2 + \frac{\lambda}{2} \sum_{j=1}^{k} h_j^2 (1 - h_j)^2 \sum_{i=1}^{d} w_{ij}^2 \end{aligned}$$

This objective function contains a combination of weight and hidden unit regularization. Penalties on hidden units can be handled in the same way as discussed in Section 3.2.7 of Chapter 3. Let a_{h_j} be the pre-activation value for the node h_j. The backpropagation updates are traditionally defined in terms of the preactivation values, where the value of $\frac{\partial J}{\partial a_{h_j}}$ is propagated backwards. After $\frac{\partial J}{\partial a_{h_j}}$ is computed using the dynamic programming update of backpropagation from the output layer, one can further update it to incorporate the effect of hidden-layer regularization of h_j:

$$\begin{aligned} \frac{\partial J}{\partial a_{h_j}} &\Leftarrow \frac{\partial J}{\partial a_{h_j}} + \frac{\lambda}{2} \frac{\partial [h_j^2 (1 - h_j)^2]}{\partial a_{h_j}} \sum_{i=1}^{d} w_{ij}^2 \\ &= \frac{\partial J}{\partial a_{h_j}} + \lambda h_j (1 - h_j)(1 - 2h_j) \underbrace{\frac{\partial h_j}{\partial a_{h_j}}}_{h_j (1 - h_j)} \sum_{i=1}^{d} w_{ij}^2 \\ &= \frac{\partial J}{\partial a_{h_j}} + \lambda h_j^2 (1 - h_j)^2 (1 - 2h_j) \sum_{i=1}^{d} w_{ij}^2 \end{aligned}$$

The value of $\frac{\partial h_j}{\partial a_{h_j}}$ is set to $h_j(1 - h_j)$ because the sigmoid activation is assumed, although it would be different for other activations. According to the chain rule, the value of $\frac{\partial J}{\partial a_{h_j}}$ should be multiplied with the value of $\frac{\partial a_{h_j}}{\partial w_{ij}} = x_i$ to obtain the gradient of the loss with respect to w_{ij}. However, according to the *multivariable* chain rule, we also need to directly add the derivative of the regularizer with respect to w_{ij} in order to obtain the full gradient. Therefore, the partial derivative of the hidden-layer regularizer R with respect to the weight is added as follows:

$$\begin{aligned} \frac{\partial J}{\partial w_{ij}} &\Leftarrow \frac{\partial J}{\partial a_{h_j}} \frac{\partial a_{h_j}}{\partial w_{ij}} + \lambda \frac{\partial R}{\partial w_{ij}} \\ &= x_i \frac{\partial J}{\partial a_{h_j}} + \lambda w_{ij} h_j^2 (1 - h_j)^2 \end{aligned}$$

Interestingly, if a linear hidden unit is used instead of the sigmoid, it is easy to see that the objective function will become identical to that of an L_2-regularized autoencoder. Therefore, it makes sense to use this approach only with a nonlinear hidden layer, because a linear hidden layer can be handled in a much simpler way. The weights in the encoder and decoder can be either tied or independent. If the weights are tied then the gradients over both copies of a weight need to be added. The above discussion assumes a single hidden layer, although it is easy to generalize to more hidden layers. The work in [397] showed that better compression can be achieved with the use of deeper variants of the approach.

Some interesting relationships exist between the de-noising autoencoder and the contractive autoencoder. The de-noising autoencoder achieves its goals of robustness stochastically by explicitly adding noise, whereas a contractive autoencoder achieves its goals analytically by adding a regularization term. Adding a small amount of Gaussian noise in a de-noising autoencoder achieves roughly similar goals as a contractive autoencoder, when the hidden layer uses linear activation. When the hidden layer uses linear activation, the partial derivative of the hidden unit with respect to an input is simply the connecting weight, and therefore the objective function of the contractive autoencoder becomes the following:

$$J_{linear} = \sum_{i=1}^{d}(x_i - \hat{x}_i)^2 + \frac{\lambda}{2}\sum_{i=1}^{d}\sum_{j=1}^{k} w_{ij}^2 \tag{4.15}$$

In that case, both the contractive and the de-noising autoencoders become similar to regularized singular value decomposition with L_2-regularization. The difference between the de-noising autoencoder and the contractive autoencoder is visually illustrated in Figure 4.11. In the case of the de-noising autoencoder on the left, the autoencoder learns the directions along the true manifold of uncorrupted data by using the relationship between the corrupted data in the output and the true data in the input. This goal is achieved analytically in the contractive autoencoder, because the vast majority of random perturbations are roughly orthogonal to the manifold when the dimensionality of the manifold is much smaller than the input data dimensionality. In such a case, perturbing the data point slightly does not change the hidden representation along the manifold very much. Penalizing the partial derivative of the hidden layer equally along all directions ensures that the partial derivative is significant only along the small number of directions along the true manifold, and the partial derivatives along the vast majority of orthogonal directions are close to 0. In other words, the variations that are not meaningful to the distribution of the specific training data set at hand are damped, and only the meaningful variations are kept.

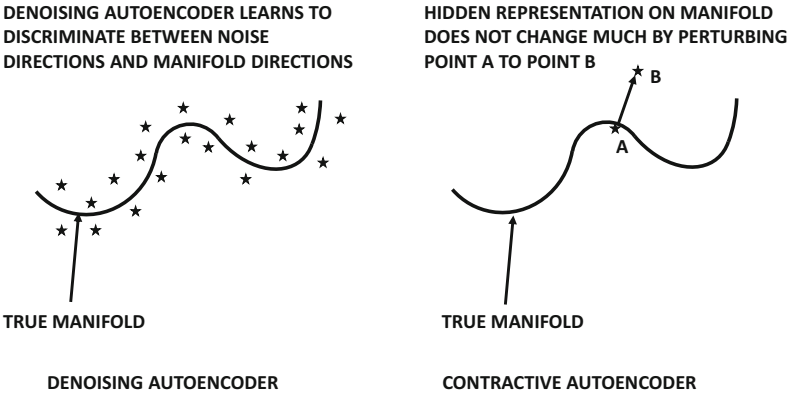


Figure 4.11: The difference between the de-noising and the contractive autoencoder

Another difference between the two methods is that the de-noising autoencoder shares the responsibility for regularization between the encoder and decoder, whereas the contractive autoencoder places this responsibility only on the encoder. Only the encoder portion is used in feature extraction; therefore, contractive autoencoders are more useful for feature engineering.

In a contractive autoencoder, the gradients are deterministic, and therefore it is also easier to use second-order learning methods as compared to the de-noising autoencoder. On the other hand, the de-noising autoencoder is easier to construct (with small changes to the code of an unregularized autoencoder), if first-order learning methods are used.

4.10.4 Hidden Probabilistic Structure: Variational Autoencoders

Just as sparse encoders impose a sparsity constraint on the hidden units, variational encoders impose a specific probabilistic structure on the hidden units. The simplest constraint is that the activations in the hidden units over the whole data should be drawn from the standard Gaussian distribution (i.e., zero mean and unit variance in each direction). By imposing this type of constraint, one advantage is that we can throw away the encoder after training, and simply feed samples from the standard normal distribution to the decoder in order to generate samples of the training data. However, if every object is generated from an identical distribution, then it would be impossible to either differentiate the various objects or to reconstruct them from a given input. Therefore, the *conditional* distribution of the activations in the hidden layer (with respect to a specific input object) would have a different distribution from the standard normal distribution. Even though a regularization term would try to pull even the conditional distribution towards the standard normal distribution, this goal would only be achieved over the distribution of hidden samples from the whole data rather than the hidden samples from a single object.

Imposing a constraint on the probabilistic distribution of hidden variables is more complicated than the other regularizers discussed so far. However, the key is to use a re-parametrization approach in which the encoder creates the k-dimensional mean and standard deviations vector of the conditional Gaussian distribution, and the hidden vector is sampled from this distribution as shown in Figure 4.12(a). Unfortunately, this network still has a sampling component. The weights of such a network cannot be learned by backpropagation because the stochastic portions of the computations are not differentiable, and therefore backpropagation cannot be not used. Therefore, the stochastic part of it can be addressed by the user explicitly generating k-dimensional samples in which each component is drawn from the standard normal distribution. The mean and standard deviation output by the encoder are used to scale and translate the input sample from the Gaussian distribution. This architecture is shown in Figure 4.12(b). By generating the stochastic potion explicitly as a part of the input, the resulting architecture is now fully deterministic, and its weights can be learned by backpropagation. Furthermore, the values of the generated samples from the standard normal distribution will need to be used in the backpropagation updates.

For each object $\overline{X}$, separate hidden activations for the mean and standard deviation are created by the encoder. The k-dimensional activations for the mean and standard deviation are denoted by $\overline{\mu}(\overline{X})$ and $\overline{\sigma}(\overline{X})$, respectively. In addition, a k-dimensional sample $\overline{z}$ is generated from $\mathcal{N}(0, I)$, where I is the identity matrix, and treated as an input into the hidden layer by the user. The hidden representation $\overline{h}(\overline{X})$ is created by scaling this random input vector $\overline{z}$ with the mean and standard deviation as follows:

$$\overline{h}(\overline{X}) = \overline{z} \odot \overline{\sigma}(\overline{X}) + \overline{\mu}(\overline{X}) \tag{4.16}$$

Here, $\odot$ indicates element-wise multiplication. These operations are shown in Figure 4.12(b) with the little circles containing the multiplication and addition operators. The elements of the vector $\overline{h}(\overline{X})$ for a particular object will obviously diverge from the standard normal distribution unless the vectors $\overline{\mu}(\overline{X})$ and $\overline{\sigma}(\overline{X})$ contain only 0s and 1s, respectively. This will not be the case because of the reconstruction component of the loss, which forces the conditional distributions of the hidden representations of particular points to have different means

and lower standard deviations than that of the standard normal distribution (which is like a prior distribution). The distribution of the hidden representation of a particular point is a posterior distribution (conditional on the specific training data point), and therefore it will differ from the Gaussian prior. The overall loss function is expressed as a weighted sum of the reconstruction loss and the regularization loss. One can use a variety of choices for the reconstruction error, and for simplicity we will use the squared loss, which is defined as follows:

$$L = ||\overline{X} - \overline{X}'||^2 \tag{4.17}$$

Here, $\overline{X}'$ is the reconstruction of the input point $\overline{X}$ from the decoder. The regularization loss R is simply the Kullback-Leibler (KL)-divergence measure of the conditional hidden distribution with parameters $(\overline{\mu}(\overline{X})), \overline{\sigma}(\overline{X}))$ with respect to the k-dimensional Gaussian distribution with parameters $(0, I)$. This value is defined as follows:

$$R = \frac{1}{2}\left(\underbrace{||\overline{\mu}(\overline{X})||^2}_{\overline{\mu}(\overline{X})_i \Rightarrow 0} + \underbrace{||\overline{\sigma}(\overline{X})||^2 - 2\sum_{i=1}^{k} \ln(\overline{\sigma}(\overline{X})_i) - k}_{\overline{\sigma}(\overline{X})_i \Rightarrow 1} \right) \tag{4.18}$$

Below some of the terms, we have annotated the specific effects of these terms in pushing parameters in particular directions. The constant term does not really do anything but it is a part of the KL-divergence function. Including the constant term does have the cosmetically satisfying effect that the regularization portion of the objective function reduces to 0, if the parameters $(\overline{\mu}(\overline{X})), \overline{\sigma}(\overline{X}))$ are the same as those of the isotropic Gaussian distribution with zero mean and unit variance in all directions. However, this will not be the case for any specific data point because of the effect of the reconstruction portion of the objective function. Over all training data points, the distribution of the hidden representation will, however, move closer to the standardized Gaussian because of the regularization term. The overall objective function J for the data point $\overline{X}$ is defined as the weighted sum of the reconstruction loss and the regularization loss:

$$J = L + \lambda R \tag{4.19}$$

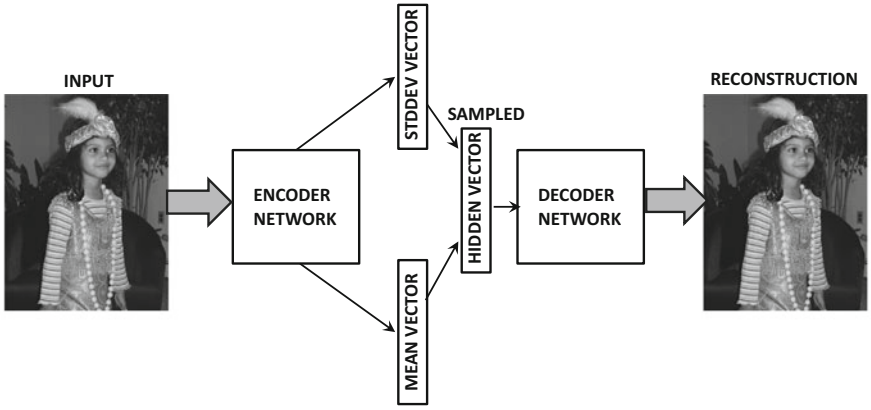
Here, $\lambda > 0$ is the regularization parameter. Small values of λ will favor exact reconstruction, and the approach will behave like a traditional autoencoder. The regularization term forces the hidden representations to be stochastic, so that multiple hidden representations generate almost the same point. This increases generalization power because it is easier to model a new image that is like (but not an exact likeness of) an image in the training data within the stochastic range of hidden values. However, since there will be overlaps among the distributions of the hidden representations of similar points, it has some undesirable side effects. For example, the reconstructions tend to be blurry, when using the approach to reconstruct images. This is caused by an averaging effect over somewhat similar points. In the extreme case, if the value of λ is chosen to be exceedingly large, then all points will have the same hidden distribution (which is an isotropic Gaussian distribution with zero mean and unit variance). The reconstruction might provide a gross averaging over large numbers of training points, which will not be meaningful. The blurriness of the reconstructions of the variational autoencoder is an undesirable property of this class of models in comparison with several other related models for generative modeling.

Training the Variational Autoencoder

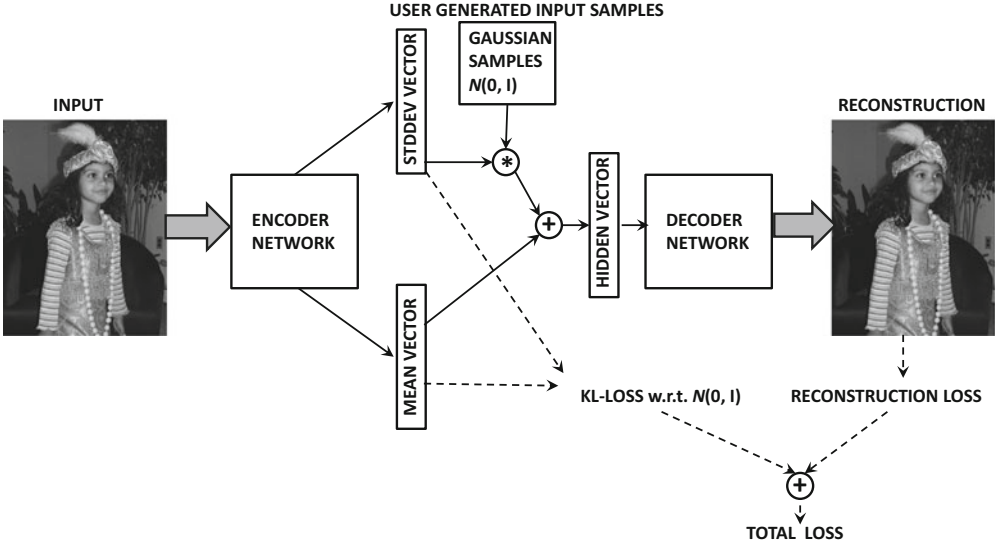
The training of a variational autoencoder is relatively straightforward because the stochasticity has been pulled out as an additional input. One can backpropagate as in any

traditional neural network. The only difference is that one needs to backpropagate across the unusual form of Equation 4.16. Furthermore, one needs to account for the penalties of the hidden layer during backpropagation.

First, one can backpropagate the loss L up to the hidden state $\overline{h}(\overline{X}) = (h_1 \ldots h_k)$ using traditional methods. Let $\overline{z} = (z_1 \ldots z_k)$ be the k random samples from $\mathcal{N}(0, 1)$, which are used in the current iteration. In order to backpropagate from $\overline{h}(\overline{X})$ to $\overline{\mu}(\overline{X}) = (\mu_1 \ldots \mu_k)$ and $\overline{\sigma}(\overline{X}) = (\sigma_1 \ldots \sigma_k)$, one can use the following relationship:



(a) Point-specific Gaussian distribution (stochastic and non-differentiable loss)



(b) Point-specific Gaussian distribution (deterministic and differentiable loss)

Figure 4.12: Re-parameterizing a variational autoencoder

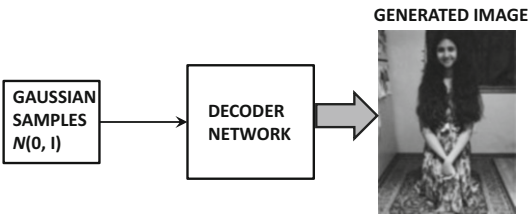


Figure 4.13: Generating samples from the variational autoencoder. The images are illustrative only.

$$J = L + \lambda R \tag{4.20}$$

$$\frac{\partial J}{\partial \mu_i} = \frac{\partial L}{\partial h_i} \underbrace{\frac{\partial h_i}{\partial \mu_i}}_{=1} + \lambda \frac{\partial R}{\partial \mu_i} \tag{4.21}$$

$$\frac{\partial J}{\partial \sigma_i} = \frac{\partial L}{\partial h_i} \underbrace{\frac{\partial h_i}{\partial \sigma_i}}_{=z_i} + \lambda \frac{\partial R}{\partial \sigma_i} \tag{4.22}$$

The values below the under-braces show the evaluations of partial derivatives of h_i with respect to μ_i and σ_i, respectively. Note that the values of $\frac{\partial h_i}{\partial \mu_i} = 1$ and $\frac{\partial h_i}{\partial \sigma_i} = z_i$ are obtained by differentiating Equation 4.16 with respect to μ_i and σ_i, respectively. The value of $\frac{\partial L}{\partial h_i}$ on the right-hand side is available from backpropagation. The values of $\frac{\partial R}{\partial \mu_i}$ and $\frac{\partial R}{\partial \sigma_i}$ are straightforward derivatives of the KL-divergence in Equation 4.18. Subsequent error propagation from the activations for $\overline{\mu}(\overline{X})$ and $\overline{\sigma}(\overline{X})$ can proceed in a similar way to the normal workings of the backpropagation algorithm.

The architecture of the variational autoencoder is considered fundamentally different from other types of autoencoders because it models the hidden variables in a stochastic way. However, there are still some interesting connections. In the de-noising autoencoder, one adds noise to the input; however, there is no constraint on the shape of the hidden distribution. In the variational autoencoder, one works with a stochastic hidden representation, although the stochasticity is pulled out by using it as an additional input during training. In other words, noise is added to the hidden representation rather than the input data. The variational approach improves generalization, because it encourages each input to map to its own stochastic region in the hidden space rather than mapping it to a single point. Small changes in the hidden representation, therefore, do not change the reconstruction too much. This assertion would also be true with a contractive autoencoder. However, constraining the shape of the hidden distribution to be Gaussian is a more fundamental difference of the variational autoencoder from other types of transformations.

4.10.4.1 Reconstruction and Generative Sampling

The approach can be used for creating the reduced representations as well as generating samples. In the case of data reduction, a Gaussian distribution with mean $\overline{\mu}(\overline{X})$ and standard deviation $\overline{\sigma}(\overline{X})$ is obtained, which represents the distribution of the hidden representation.

However, a particularly interesting application of the variational autoencoder is to generate samples from the underlying data distribution. Just as feature engineering methods

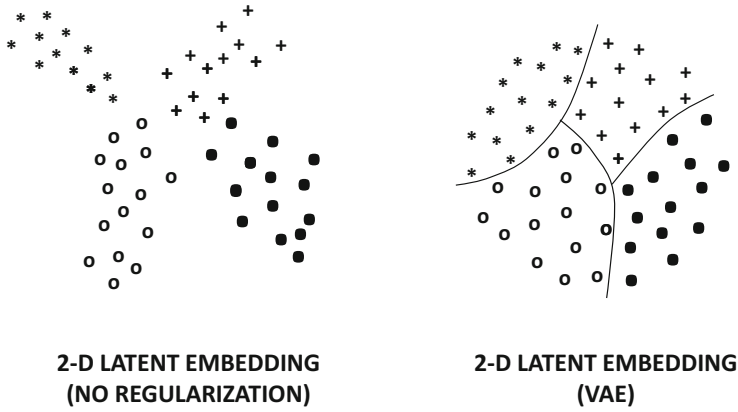


Figure 4.14: Illustrations of the embeddings created by a variational autoencoder in relation to the unregularized version. The unregularized version has large discontinuities in the latent space, which might not correspond to meaningful points. The Gaussian embedding of the points in the variational autoencoder makes sampling possible.

use only the encoder portion of the autoencoder (once training is done), variational autoencoders use only the decoder portion. The basic idea is to repeatedly draw a point from the Gaussian distribution and feed it to the hidden units in the decoder. The resulting "reconstruction" output of the decoder will be a point satisfying a similar distribution as the original data. As a result, the generated point will be a realistic sample from the original data. The architecture for sample generation is shown in Figure 4.13. The shown image is illustrative only, and does not reflect the actual output of a variational autoencoder (which is generally of somewhat lower quality). To understand why a variational autoencoder can generate images in this way, it is helpful to view the typical types of embeddings an unregularized autoencoder would create versus a method like the variational autoencoder. In the left side of Figure 4.14, we have shown an example of the 2-dimensional embeddings of the training data created by an unregularized autoencoder of a four-class distribution (e.g., four digits of MNIST). It is evident that there are large discontinuities in particular regions of the latent space, and that these sparse regions may not correspond to meaningful points. On the other hand, the regularization term in the variational autoencoder encourages the training points to be (roughly) distributed in a Gaussian distribution, and there are far fewer discontinuities in the embedding on the right-hand side of Figure 4.14. Consequently, sampling from any point in the latent space will yield meaningful reconstructions of one of the four classes (i.e., one of the digits of MNIST). Furthermore, "walking" from one point in the latent space to another along a straight line in the second case will result in a smooth transformation across classes. For example, walking from a region containing instances of '4' to a region containing instances of '7' in the latent space of the MNIST data set would result in a slow change in the style of the digit '4' until a transition point, where the handwritten digit could be interpreted either as a '4' or a '7'. This situation does occur in real settings as well because such types of confusing handwritten digits do occur in the MNIST data set. Furthermore, the placement of different digits within the embedding would be such that digit pairs with smooth transitions at confusion points (e.g., $[4, 7]$ or $[5, 6]$) are placed adjacent to one another in the latent space.

It is important to understand that the generated objects are often similar to but not exactly the same as those drawn from the training data. Because of its stochastic nature, the variational autoencoder has the ability to explore different modes of the generation process, which leads to a certain level of creativity in the face of ambiguity. This property can be put to good use by conditioning the approach on another object.

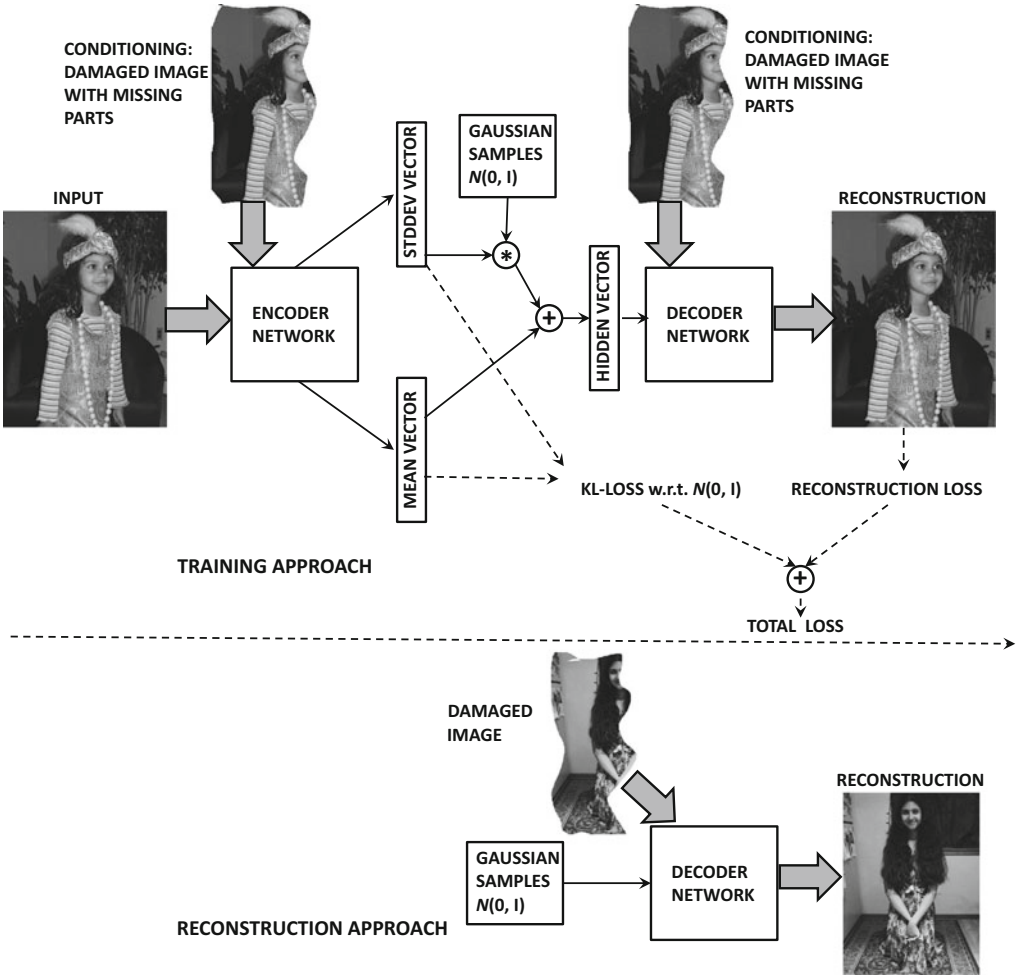


Figure 4.15: Reconstructing damaged images with the conditional variational autoencoder. The images are illustrative only.

4.10.4.2 Conditional Variational Autoencoders

One can apply conditioning to variational autoencoders in order to obtain some interesting results [510, 463]. The basic idea in conditional variational autoencoders is to add an additional conditional input, which typically provides a related context. For example, the context might be a damaged image with missing holes, and the job of the autoencoder is to reconstruct it. Predictive models will generally perform poorly in this type of setting because the level of ambiguity may be too large, and an averaged reconstruction across all images might not be useful. During the training phase, pairs of damaged and original

images are needed, and therefore the encoder and decoder are able to learn how the context relates to the images being generated from the training data. The architecture of the training phase is illustrated in the upper part of Figure 4.15. The training is otherwise similar to the unconditional variational autoencoder. During the testing phase, the context is provided as an additional input, and the autoencoder reconstructs the missing portions in a reasonable way based on the model learned in the training phase. The architecture of the reconstruction phase is illustrated in the lower part of Figure 4.15. The simplicity of this architecture is particularly notable. The shown images are only illustrative; in actual executions on image data, the generated images are often blurry, especially in the missing portions. This is a type of image-to-image translation approach, which will be revisited in Chapter 10 under the context of a discussion on *generative adversarial networks*.

4.10.4.3 Relationship with Generative Adversarial Networks

Variational autoencoders are closely related to another class of models, referred to as generative adversarial networks. However, there are some key differences as well. Like variational autoencoders, generative adversarial networks can be used to create images that are similar to a base training data set. Furthermore, conditional variants of both models are useful for completing missing data, especially in cases where the ambiguity is large enough to require a certain level of creativity from the generative process. However, the results of generative adversarial networks are often more realistic because the decoders are explicitly trained to create good counterfeits. This is achieved by having a discriminator as a judge of the quality of the generated objects. Furthermore, the objects are also generated in a more creative way because the generator is never shown the original objects in the training data set, but is only given guidance to fool the discriminator. As a result, generative adversarial networks learn to create creative counterfeits. In certain domains such as image and video data, this approach can have remarkable results; unlike variational autoencoders, the quality of the images is not blurry. One can create vivid images and videos with an artistic flavor, that give the impression of dreaming. These techniques can also be used in numerous applications like text-to-image or image-to-image translation. For example, one can specify a text description, and then obtain a fantasy image that matches the description [392]. Generative adversarial networks are discussed in Section 10.4 of Chapter 10.

4.11 Summary

Neural networks often contain a large number of parameters, which causes overfitting. One solution is to restrict the size of the networks up front. However, such an approach often provides suboptimal solutions when the model is complex and sufficient data are available. A more flexible approach is to use tunable regularization, in which a large number of parameters are allowed. In such cases, the regularization restricts the size of the parameter space in a soft way. The most common form of regularization is penalty-based regularization. It is common to use penalties on the parameters, although it is also possible to use penalties on the activations of the hidden units. The latter approach leads to sparse representations in the hidden units. Ensemble learning is a common approach to reduce variance, and some ensemble methods like *Dropout* are specifically designed for neural networks. Other common regularization methods include early stopping and pretraining. Pretraining acts as a regularizer by acting as a form of semi-supervised learning, which works from the simple to the complex by initializing with a simple heuristic and using backpropagation to discover

refined solutions. Other related techniques include curriculum and continuation methods, which also work from the simple to the complex in order to provide solutions with low generalization error. Although overfitting is often a less serious problem in unsupervised settings, different types of regularization are used to impose structure on the learned models.

4.12 Bibliographic Notes

A detailed discussion of the bias-variance trade-off may be found in [177]. The bias-variance trade-off originated in the field of statistics, where it was proposed in the context of the regression problem. The generalization to the case of binary loss functions in classification was proposed in [247, 252]. Early methods for reducing overfitting were proposed in [175, 282] in which unimportant weights were removed from a network to reduce its parameter footprint. It was shown that this type of pruning had significant benefits in terms of generalization. The early work also showed [450] that deep and narrow networks tended to generalize better than broad and shallow networks. This is primarily because the depth imposes a structure on the data, and can represent the data in a fewer number of parameters. A recent study of model generalization in neural networks is provided in [557].

The use of L_2-regularization in regression dates back to Tikhonov-Arsenin's seminal work [499]. The equivalence of Tikhonov regularization and training with noise was shown by Bishop [44]. The use of L_1-regularization is studied in detail in [179]. Several regularization methods have also been proposed that are specifically designed for neural architectures. For example, the work in [201] proposes a regularization technique that constrains the norm of each layer in a neural network. Sparse representations of the data are explored in [67, 273, 274, 284, 354].

Detailed discussions of ensemble methods for classification may be found in [438, 566]. The bagging and subsampling methods are discussed in [50, 56]. The work in [515] proposes an ensemble architecture that is inspired by a random forest. This architecture is illustrated in Figure 1.16 of Chapter 1. This type of ensemble is particularly well suited for problems with small data sets, where a random forest is known to work well. The approach for random edge dropping was introduced in the context of outlier detection [64], whereas the *Dropout* approach was presented in [467]. The work in [567] discusses the notion that it is better to combine the results of the top-performing ensemble components rather than combining all of them. Most ensemble methods are designed for variance reduction, although a few techniques like *boosting* [122] are also designed for bias reduction. Boosting has also been used in the context of neural network learning [435]. However, the use of boosting in neural networks is generally restricted to the incremental addition of hidden units based on error characteristics. A key point about boosting is that it tends to overfit the data, and is therefore suitable for high-bias learners but not high-variance learners. Neural networks are inherently high-variance learners. The relationship between boosting and certain types of neural architectures is pointed out in [32]. Data perturbation methods for classification are discussed in [63], although this method primarily seems to be about increasing the amount of available data of a minority class, and does not discuss variance reduction methods. A later book [5] discusses how this approach can be combined with a variance reduction method. Ensemble methods for neural networks are proposed in [170].

Different types of pretraining have been explored in the context of neural networks [31, 113, 196, 386, 506]. The earliest methods for unsupervised pretraining were proposed in [196]. The original work of pretraining [196] was based on probabilistic graphical models (cf. Section 6.7) and was later extended to conventional autoencoders [386, 506].

Compared to unsupervised pretraining, the effect of supervised pretraining is limited [31]. A detailed discussion of why unsupervised pretraining helps deep learning is provided in [113]. This work posits that unsupervised pretraining implicitly acts as a regularizer, and therefore it improves the generalization power to unseen test instances. This fact is also evidenced by the experimental results in [31], which show that supervised variations of pretraining do not help as much as unsupervised variations of pretraining. In this sense, unsupervised pretraining can be viewed as a type of semi-supervised learning, which restricts the parameter search to specific regions of the parameter space, which depend on the base data distribution at hand. Pretraining also does not seem to help with certain types of tasks [303]. Another form of semi-supervised learning can be performed with *ladder networks* [388, 502], in which skip-connections are used in conjunction with an autoencoder-like architecture.

Curriculum and continuation learning are applications of the principle of moving from simple to complex models. Continuation learning methods are discussed in [339, 536]. A number of methods were proposed in the early years [112, 422, 464] that showed the advantages of curriculum learning. The basic principles of curriculum learning are discussed in [238]. The relationship between curriculum and continuation learning is explored in [33].

Numerous unsupervised methods have been proposed for regularization. A discussion of sparse autoencoders may be found in [354]. De-noising autoencoders are discussed in [506]. The contractive autoencoder is discussed in [397]. The use of de-noising autoencoders in recommender systems is discussed in [472, 535]. The ideas in the contractive autoencoder are reminiscent of *double backpropagation* [107] in which small changes in the input are not allowed to change the output. Related ideas are also discussed in the *tangent classifier* [398].

The variational autoencoder is introduced in [242, 399]. The use of importance weighting to improve over the representations learned by the variational autoencoder is discussed in [58]. Conditional variational autoencoders are discussed in [463, 510]. A tutorial on variational autoencoders is found in [106]. Generative variants of de-noising autoencoders are discussed in [34]. Variational autoencoders are closely related to generative adversarial networks, which are discussed in Chapter 10. Closely related methods for designing adversarial autoencoders are discussed in [311].

4.12.1 Software Resources

Numerous ensemble methods are available from machine learning libraries like *scikit-learn* [587]. Most of the weight-decay and penalty-based methods are available as standardized options in the deep learning libraries. However, techniques like *Dropout* are application-specific and need to be implemented from scratch. Implementations of several different types of autoencoders may be found in [595]. Several implementations of the variational autoencoder may be found in [596, 597, 640].

4.13 Exercises

1. Consider two neural networks used for regression modeling with identical structure of an input layer and 10 hidden layers containing 100 units each. In both cases, the output node is a single unit with linear activation. The only difference is that one of them uses linear activations in the hidden layers and the other uses sigmoid activations. Which model will have higher variance in prediction?

2. Consider a situation in which you have four attributes $x_1 \ldots x_4$, and the dependent variable y is such that $y = 2x_1$. Create a tiny training data set of 5 distinct examples in which a linear regression model without regularization will have an infinite number of coefficient solutions with $w_1 = 0$. Discuss the performance of such a model on out-of-sample data. Why will regularization help?

3. Implement a perceptron with and without regularization. Test the accuracy of both variations of the perceptron on both the training data and the out-of-sample data on the *Ionosphere* data set of the *UCI Machine Learning Repository* [601]. What do you observe about the effect of regularization in the two cases? Repeat the experiment with smaller samples of the *Ionosphere* training data, and report your observations.

4. Implement an autoencoder with a single hidden layer. Reconstruct inputs for the *Ionosphere* data set of the previous exercise with (a) no added noise and weight regularization, (b) added Gaussian noise and no weight regularization.

5. The discussion in the chapter uses an example of sigmoid activation for the contractive autoencoder. Consider a contractive autoencoder with a single hidden layer and ReLU activation. Discuss how the updates change when ReLU activation is used.

6. Suppose that you have a model that provides around 80% accuracy on the training as well as on the out-of-sample test data. Would you recommend increasing the amount of data or adjusting the model to improve accuracy?

7. In the chapter, we showed that adding Gaussian noise to the input features in linear regression is equivalent to L_2-regularization of linear regression. Discuss why adding of Gaussian noise to the input data in a de-noising single-hidden layer autoencoder with linear units is roughly equivalent to L_2-regularized singular value decomposition.

8. Consider a network with a single input layer, two hidden layers, and a single output predicting a binary label. All hidden layers use the sigmoid activation function and no regularization is used. The input layer contains d units, and each hidden layer contains p units. Suppose that you add an additional hidden layer between the two current hidden layers, and this additional hidden layer contains q linear units.

   (a) Even though the number of parameters have increased by adding the hidden layer, discuss why the capacity of this model will decrease when $q < p$.

   (b) Does the capacity of the model increase when $q > p$?

9. Bob divided the labeled classification data into a portion used for model construction and another portion for validation. Bob then tested 1000 neural architectures by learning parameters (backpropagating) on the model-construction portion and testing its accuracy on the validation portion. Discuss why the resulting model is likely to yield poorer accuracy on the out-of-sample test data as compared to the validation data, even though the validation data was not used for learning parameters. Do you have any recommendations for Bob on using the results of his 1000 validations?

10. Does the classification accuracy on the training data generally improve with increasing training data size? How about the point-wise average of the loss on training instances? At what point do training and testing accuracy become similar? Explain your answer.

11. What is the effect of increasing the regularization parameter on the training and testing accuracy? At what point do training and testing accuracy become similar?

Chapter 5

Radial Basis Function Networks

"Two birds disputed about a kernel, when a third swooped down and carried it off."—African Proverb

5.1 Introduction

Radial basis function (RBF) networks represent a fundamentally different architecture from what we have seen in the previous chapters. All the previous chapters use a feed-forward network in which the inputs are transmitted forward from layer to layer in a similar fashion in order to create the final outputs. A feed-forward network might have many layers, and the nonlinearity is typically created by the repeated composition of activation functions. On the other hand, an RBF network typically uses only an input layer, a single hidden layer (with a special type of behavior defined by RBF functions), and an output layer. Although it is possible to replace the output layer with multiple feed-forward layers (like a conventional network), the resulting network is still quite shallow, and its behavior is strongly influenced by the nature of the special hidden layer. For simplicity in discussion, we will work with only a single output layer. As in feed-forward networks, the input layer is not really a computational layer, and it only carries the inputs forward. The nature of the computations in the hidden layer are very different from what we have seen so far in feed-forward networks. In particular, the hidden layer performs a computation based on a comparison with a *prototype vector*, which has no exact counterpart in feed-forward networks. The structure and the computations performed by the special hidden layer is the key to the power of the RBF network.

One can characterize the difference in the functionality of the hidden and output layers as follows:

1. The hidden layer takes the input points, in which the class structure might not be linearly separable, and transforms them into a new space that is (often) linearly separable. The hidden layer often has higher dimensionality than the input layer, because transformation to a higher-dimensional space is often required in order to ensure linear separability. This principle is based on *Cover's theorem on separability of patterns* [84], which states that pattern classification problems are more likely to be linearly separable when cast into a high-dimensional space with a nonlinear transformation. Furthermore, certain types of transformations in which features represent small localities in the space are more likely to lead to linear separability. Although the dimensionality of the hidden layer is typically greater than the input dimensionality, it is always less than or equal to the number of training points. An extreme case in which the dimensionality of the hidden layer is equal to the number of training points can be shown to be roughly equivalent to *kernel learners*. Examples of such models include kernel regression and kernel support vector machines.

2. The output layer uses linear classification or regression modeling with respect to the inputs from the hidden layer. The connections from the hidden to the output layer have weights attached to them. The computations in the output layer are performed in the same way as in a standard feed-forward network. Although it is also possible to replace the output layer with multiple feed-forward layers, we will consider only the case of a single feed-forward layer for simplicity.

Just as the perceptron is a variant of the linear support vector machine, the RBF network is a generalization of kernel classification and regression. Special cases of the RBF network can be used to implement kernel regression, least-squares kernel classification, and the kernel support-vector machine. The differences among these special cases is in terms of how the output layer and the loss function is structured. In feed-forward networks, increasing nonlinearity is obtained by increasing depth. However, in an RBF network, a single hidden layer is usually sufficient to achieve the required level of nonlinearity because of its special structure. Like feed-forward networks, RBF networks are universal function approximators.

The layers of the RBF network are designed as follows:

1. The input layer simply transmits from the input features to the hidden layers. Therefore, the number of input units is exactly equal to the dimensionality d of the data. As in the case of feed-forward networks, no computation is performed in the input layers. As in all feed-forward networks, the input units are fully connected to the hidden units and carry their input forward.

2. The computations in the hidden layers are based on comparisons with *prototype vectors*. Each hidden unit contains a d-dimensional prototype vector. Let the prototype vector of the ith hidden unit be denoted by $\overline{\mu}_i$. In addition, the ith hidden unit contains a bandwidth denoted by σ_i. Although the prototype vectors are always specific to particular units, the bandwidths of different units σ_i are often set to the same value σ. The prototype vectors and bandwidth(s) are usually learned either in an unsupervised way, or with the use of mild supervision.

Then, for any input training point $\overline{X}$, the activation $\Phi_i(\overline{X})$ of the ith hidden unit is defined as follows:

$$h_i = \Phi_i(\overline{X}) = \exp\left(-\frac{||\overline{X} - \overline{\mu}_i||^2}{2 \cdot \sigma_i^2}\right) \quad \forall i \in \{1, \ldots, m\} \tag{5.1}$$

The total number of hidden units is denoted by m. Each of these m units is designed to have a high level of influence on the particular cluster of points that is closest to its prototype vector. Therefore, one can view m as the number of clusters used for modeling, and it represents an important hyper-parameter available to the algorithm. For low-dimensional inputs, it is typical for the value of m to be larger than the input dimensionality d, but smaller than the number of training points n.

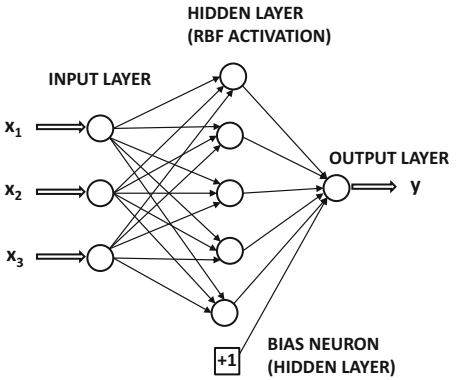


Figure 5.1: An RBF network: Note that the hidden layer is broader than the input layer, which is typical (but not mandatory).

3. For any particular training point $\overline{X}$, let h_i be the output of the ith hidden unit, as defined by Equation 5.1. The weights of the connections from the hidden to the output nodes are set to w_i. Then, the prediction $\hat{y}$ of the RBF network in the output layer is defined as follows:

$$\hat{y} = \sum_{i=1}^{m} w_i h_i = \sum_{i=1}^{m} w_i \Phi_i(\overline{X}) = \sum_{i=1}^{m} w_i \exp\left(-\frac{||\overline{X} - \overline{\mu}_i||^2}{2 \cdot \sigma_i^2}\right)$$

The variable $\hat{y}$ has a circumflex on top to indicate the fact that it is a predicted value rather than observed value. If the observed target is real-valued, then one can set up a least-squares loss function, which is much like that in a feed-forward network. The values of the weights $w_1 \ldots w_m$ need to be learned in a supervised way.

An additional detail is that the hidden layer of the neural network contains bias neurons. Note that the bias neuron can be implemented by a single hidden unit in the output layer, which is always on. One can also implement this type of neuron by creating a hidden unit in which the value of σ_i is ∞. In either case, it will be assumed throughout the discussions in this chapter that this special hidden unit is absorbed among the m hidden units. Therefore, it is not treated in any special way. An example of an RBF network is illustrated in Figure 5.1.

In the RBF network, there are two sets of computations corresponding to the hidden layer and the output layer. The parameters $\overline{\mu}_i$ and σ_i of the hidden layer are learned in

an unsupervised way, whereas those of the output layer are learned in a supervised way with gradient descent. The latter is similar to the case of the feed-forward network. The prototypes $\overline{\mu}_i$ may either be sampled from the data, or be set to be the m centroids of an m-way clustering algorithm. The latter solution is used frequently. The different ways of training the neural network are discussed in Section 5.2.

RBF networks can be shown to be direct generalizations of the class of kernel methods. This is primarily because the prediction in the output node can be shown to be equivalent to a weighted nearest neighbor estimator, where the weights are products of the coefficients w_i and Gaussian RBF similarities to *prototypes*. The prediction function in almost all kernel methods can also be shown to be equivalent to a weighted nearest neighbor estimator, where the weights are learned in a supervised way. Therefore, kernel methods represent a special case of RBF methods in which the number of hidden nodes is equal to the number of training points, the prototypes are set to the training points, and each σ_i has the same value. This suggests that RBF networks have greater power and flexibility than do kernel methods; this relationship will be discussed in detail in Section 5.4.

When to Use RBF Networks

A key point is that the hidden layer of the RBF network is created in an unsupervised way, tending to make it robust to all types of noise (including adversarial noise). Indeed, this property of RBF networks is shared by support vector machines. At the same time, there are limitations with respect to how much structure in the data an RBF network can learn. Deep feed-forward networks are effective at learning from data with a rich structure because the multiple layers of nonlinear activations force the data to follow specific types of patterns. Furthermore, by adjusting the structure of connections, one can incorporate domain-specific insights in feed-forward networks. Examples of such settings include recurrent and convolutional neural networks. The single layer of an RBF network limits the amount of structure that one can learn. Although both RBF networks and deep feed-forward networks are known to be universal function approximators, there are differences in terms of their generalization performance on different types of data sets.

Chapter Organization

This chapter is organized as follows. The next section discusses the various training methods for RBF networks. The use of RBF networks in classification and interpolation is discussed in Section 5.3. The relationship of the RBF method to kernel regression and classification is discussed in Section 5.4. A summary is provided in Section 5.5.

5.2 Training an RBF Network

The training of an RBF network is very different from that of a feed-forward network, which is fully integrated across different layers. In an RBF network, the training of the hidden layer is typically done in an unsupervised manner. While it is possible, in principle, to train the prototype vectors and the bandwidths using backpropagation, the problem is that there are more local minima on the loss surface of RBF networks compared to feed-forward networks. Therefore, the supervision in the hidden layer (when used) is often relatively gentle, or it is restricted only to fine-tuning weights that have already been learned. Nevertheless, since overfitting seems to be a pervasive problem with the supervised training of the hidden

layer, our discussion will be restricted to unsupervised methods. In the following, we will first discuss the training of the hidden layer of an RBF network, and then discuss the training of the output layer.

5.2.1 Training the Hidden Layer

The hidden layer of the RBF network contains several parameters, including the prototype vectors $\overline{\mu}_1 \ldots \overline{\mu}_m$, and the bandwidths $\sigma_1 \ldots \sigma_m$. The hyperparameter m controls the number of hidden units. In practice, a separate value of σ_i is not set for each unit, and all units have the same bandwidth σ. However, the mean values $\overline{\mu}_i$ for the various hidden units are different because they define the all-important prototype vectors. The complexity of the model is regulated by the number of hidden units and the bandwidth. The combination of a small bandwidth and a large number of hidden units increases the model complexity, and is a useful setting when the amount of data is large. Smaller data sets require fewer units and larger bandwidths to avoid overfitting. The value of m is typically larger than the input data dimensionality, but it is never larger than the number of training points. Setting the value of m equal to the number of training points, and using each training point as a prototype in a hidden node, makes the approach equivalent to traditional kernel methods.

The bandwidth also depends on the chosen prototype vectors $\overline{\mu}_1 \ldots \overline{\mu}_m$. Ideally, the bandwidths should be set in a way that each point should be (significantly) influenced by only a small number of prototype vectors, which correspond to its closest clusters. Setting the bandwidth too large or too small compared to the inter-prototype distance will lead to under-fitting and over-fitting, respectively. Let d_{max} be maximum distance between pairs of prototype centers, and d_{ave} be the average distance between them. Then, two heuristic ways of setting the bandwidth are as follows:

$$\sigma = \frac{d_{max}}{\sqrt{m}}$$
$$\sigma = 2 \cdot d_{ave}$$

One problem with this choice of σ is that the optimal value of the bandwidth might vary in different parts of the input space. For example, the bandwidth in a dense region in the data space should be smaller than the bandwidth in a sparse region of the space. The bandwidth should also depend on how the prototype vectors are distributed in the space. Therefore, one possible solution is to choose the bandwidth σ_i of the ith prototype vector to be equal to its distance to its rth nearest neighbor among the prototypes. Here, r is a small value like 5 or 10.

However, these are only heuristic rules. It is possible to fine-tune these values by using a held-out portion of data set. In other words, candidate values of σ are generated in the neighborhood of the above recommended values of σ (as an initial reference point). Then, multiple models are constructed using these candidate values of σ (including the training of the output layer). The choice of σ that provides the least error on the held-out portion of the training data set is used. This type of approach does use a certain level of supervision in the selection of the bandwidth, without getting stuck in local minima. However, the nature of the supervision is quite gentle, which is particularly important when dealing with the parameters of the first layer in an RBF network. It is noteworthy that this type of tuning of the bandwidth is also performed when using the Gaussian kernel with a kernel support-vector machine. This similarity is not a coincidence because the kernel support-vector machine is a special case of RBF networks (cf. Section 5.4).

The selection of the prototype vectors is somewhat more complex. In particular, the following choices are often made:

1. The prototype vectors can be randomly sampled from the n training points. A total of $m < n$ training points are sampled in order to create the prototype vectors. The main problem with this approach is that it will over-represent prototypes from dense regions of the data, whereas sparse regions might get few or no prototypes. As a result, the prediction accuracy in such regions will suffer.

2. A k-means clustering algorithm can be used in order to create m clusters. The centroid of each of these m clusters can be used as a prototype. The use of the k-means algorithm is the most common choice for learning the prototype vectors.

3. Variants of clustering algorithms that partition the data *space* (rather than the points) are also used. A specific example is the use of decision trees to create the prototypes.

4. An alternative method for training the hidden layer is by using the *orthogonal least-squares algorithm*. This approach uses a certain level of supervision. In this approach, the prototype vectors are selected one by one from the training data in order to minimize the residual error of prediction on an out-of-sample test set. Since this approach requires understanding of the training of the output layer, its discussion will be deferred to a later section.

In the following, we briefly describe the k-means algorithm for creating the prototypes because it is the most common choice in real implementations. The k-means algorithm is a classical technique in the clustering literature. It uses the cluster prototypes as the prototypes for the hidden layer in the RBF method. Broadly speaking, the k-means algorithm proceeds as follows. At initialization, the m cluster prototypes are set to m random training points. Subsequently, each of the n data points is assigned to the prototype to which it has the smallest Euclidean distance. The assigned points of each prototype are averaged in order to create a new cluster center. In other words, the centroid of the created clusters is used to replace its old prototype with a new prototype. This process is repeated iteratively to convergence. Convergence is reached when the cluster assignments do not change significantly from one iteration to the next.

5.2.2 Training the Output Layer

The output layer is trained after the hidden layer has been trained. The training of the output layer is quite straightforward, because it uses only a single layer with linear activation. For ease in discussion, we will first consider the case in which the target of the output layer is real-valued. Later, we will discuss other settings. The output layer contains an m-dimensional vector of weights $\overline{W} = [w_1 \ldots w_m]$ that needs to be learned. Assume that the vector $\overline{W}$ is a row vector.

Consider a situation in which the training data set contains n points $\overline{X_1} \ldots \overline{X_n}$, which create the representations $\overline{H_1} \ldots \overline{H_n}$ in the hidden layer. Therefore, each $\overline{H_i}$ is an m-dimensional row vector. One could stack these n row vectors on top of one another to create an $n \times m$ matrix H. Furthermore, the observed targets of the n training points are denoted by $y_1, y_2, \ldots y_n$, which can be written as the n-dimensional column vector $\overline{y} = [y_1 \ldots y_n]^T$.

The predictions of the n training points are given by the elements of the n-dimensional column vector $H\overline{W}^T$. Ideally, we would like these predictions to be as close to the observed vector $\overline{y}$ as possible. Therefore, the loss function L for learning the output-layer weights is as follows:

$$L = \frac{1}{2}||H\overline{W}^T - \overline{y}||^2$$

In order to reduce overfitting, one can add Tikhonov regularization to the objective function:

$$L = \frac{1}{2}||H\overline{W}^T - \overline{y}||^2 + \frac{\lambda}{2}||\overline{W}||^2 \tag{5.2}$$

Here, $\lambda > 0$ is the regularization parameter. By computing the partial derivative of L with respect to the elements of the weight vector, we obtain the following:

$$\frac{\partial L}{\partial \overline{W}} = H^T(H\overline{W}^T - \overline{y}) + \lambda \overline{W}^T = 0$$

The above derivative is written in matrix calculus notation where $\frac{\partial L}{\partial \overline{W}}$ refers to the following:

$$\frac{\partial L}{\partial \overline{W}} = \left(\frac{\partial L}{\partial w_1} \cdots \frac{\partial L}{\partial w_d}\right)^T \tag{5.3}$$

By re-adjusting the above condition, we obtain the following:

$$(H^T H + \lambda I)\overline{W}^T = H^T \overline{y}$$

When $\lambda > 0$, the matrix $H^T H + \lambda I$ is positive-definite and is therefore invertible. In other words, one obtains a simple solution for the weight vector in closed form:

$$\overline{W}^T = (H^T H + \lambda I)^{-1} H^T \overline{y} \tag{5.4}$$

Therefore, a simple matrix inversion is sufficient to find the weight vector, and backpropagation is completely unnecessary.

However, the reality is that the use of a closed-form solution is not viable in practice because the size of the matrix $H^T H$ is $m \times m$, which can be large. For example, in kernel methods, we set $m = n$, in which the matrix is too large to even materialize, let alone invert. Therefore, one uses stochastic gradient descent to update the weight vector in practice. In such a case, the gradient-descent updates (with all training points) are as follows:

$$\begin{aligned}\overline{W}^T &\Leftarrow \overline{W}^T - \alpha \frac{\partial L}{\partial \overline{W}} \\ &= \overline{W}^T(1 - \alpha\lambda) - \alpha H^T \underbrace{(H\overline{W}^T - \overline{y})}_{\text{Current Errors}}\end{aligned}$$

One can also choose to use mini-batch gradient descent in which the matrix H in the above update can be replaced with a random subset of rows H_r from H, corresponding to the mini-batch. This approach is equivalent to what would normally be used in a traditional neural network with mini-batch stochastic gradient descent. However, it is applied only to the weights of the connections incident on the output layer in this case.

5.2.2.1 Expression with Pseudo-Inverse

In the case in which the regularization parameter λ is set to 0, the weight vector $\overline{W}$ is defined as follows:

$$\overline{W}^T = (H^T H)^{-1} H^T \overline{y} \tag{5.5}$$

The matrix $(H^T H)^{-1} H^T$ is said to be the *pseudo-inverse* of the matrix H. The pseudo-inverse of the matrix H is denoted by H^+. Therefore, one can write the weight vector $\overline{W}^T$ as follows:

$$\overline{W}^T = H^+ \overline{y} \tag{5.6}$$

The pseudo-inverse is a generalization of the notion of an inverse for non-singular or rectangular matrices. In this particular case, $H^T H$ is assumed to be invertible, although the pseudo-inverse of H can be computed even in cases where $H^T H$ is not invertible. In the case where H is square and invertible, the pseudo-inverse is the same as its inverse.

5.2.3 Orthogonal Least-Squares Algorithm

We revisit the training phase of the hidden layer. The training approach discussed in this section will use the predictions of the output layer in choosing the prototypes. Therefore, the training process of the hidden layer is supervised, although the supervision is restricted to iterative selections from the original training points. The orthogonal least-squares algorithm chooses the prototype vector one by one from the training points in order to minimize the error of prediction.

The algorithm starts by building an RBF network with a single hidden node and trying each possible training point as a prototype in order to compute the prediction error. One then selects the prototype from the training points that minimizes the error of prediction. In the next iteration, one more prototype is added to the selected prototype in order to build an RBF network with two prototypes. As in the previous iteration, all $(n-1)$ remaining training points are tried as possible prototypes in order to add to the current bag of prototypes, and the criterion for adding to the bag is the minimization of prediction error. In the $(r+1)$th iteration, one tries all the $(n-r)$ remaining training points, and adds one of them to the bag of prototypes so that the prediction error is minimized. Some of the training points in the data are held out, and are not used in the computations of the predictions or as candidates for prototypes. These out-of-sample points are used in order to test the effect of adding a prototype to the error. At some point, the error on this held-out set begins to rise as more prototypes as added. An increase in error on the held-out test set is a sign of the fact that further increase in prototypes will increase overfitting. This is the point at which one terminates the algorithm.

The main problem with this approach is that it is extremely inefficient. In each iteration, one must run n training procedures, which is computationally prohibitive for large training data sets. An interesting procedure in this respect is the *orthogonal least-squares algorithm* [65], which is known to be efficient. This algorithm is similar to the one described above in the sense that the prototype vectors are added iteratively from the original training

data set. However, the procedure with which the prototype is added is far more efficient. A set of orthogonal vectors are constructed in the space spanned by the hidden unit activations from the training data set. These orthogonal vectors can be used to directly compute which prototype should be selected from the training data set.

5.2.4 Fully Supervised Learning

The orthogonal least-squares algorithm represents a type of mild supervision in which the prototype vector is selected from one of the training points based on the effect to the overall prediction error. It is also possible to perform stronger types of supervision in which one can backpropagate in order to update the prototype vectors and the bandwidth. Consider the loss function L over the various training points:

$$L = \frac{1}{2}\sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i)^2 \tag{5.7}$$

Here, $\overline{H_i}$ represents the m-dimensional vector of activations in the hidden layer for the ith training point $\overline{X_i}$.

The partial derivative with respect to each bandwidth σ_j can be computed as follows:

$$\begin{aligned}\frac{\partial L}{\partial \sigma_j} &= \sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i)w_j \frac{\partial \Phi_j(\overline{X_i})}{\partial \sigma_j} \\ &= \sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i)\, w_j \Phi_j(\overline{X_i})\, \frac{||\overline{X_i} - \overline{\mu}_j||^2}{\sigma_j^3}\end{aligned}$$

If all bandwidths σ_j are fixed to the same value σ, as is common in RBF networks, then the derivative can be computed using the same trick commonly used for handling shared weights:

$$\begin{aligned}\frac{\partial L}{\partial \sigma} &= \sum_{j=1}^{m}\frac{\partial L}{\partial \sigma_j}\cdot\underbrace{\frac{\partial \sigma_j}{\partial \sigma}}_{=1} \\ &= \sum_{j=1}^{m}\frac{\partial L}{\partial \sigma_j} \\ &= \sum_{j=1}^{m}\sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i)\, w_j \Phi_j(\overline{X_i})\, \frac{||\overline{X_i} - \overline{\mu}_j||^2}{\sigma^3}\end{aligned}$$

One can also compute a partial derivative with respect to each element of the prototype vector. Let μ_{jk} represent the kth element of $\overline{\mu}_j$. Similarly, let x_{ik} represent the kth element of the ith training point $\overline{X_i}$. The partial derivative with respect to μ_{jk} is computed as follows:

$$\frac{\partial L}{\partial \mu_{jk}} = \sum_{i=1}^{n}(\overline{H_i} \cdot \overline{W} - y_i)\, w_j \Phi_j(\overline{X_i})\, \frac{(x_{ik} - \mu_{jk})}{\sigma_j^2} \tag{5.8}$$

Using these partial derivatives, one can update the bandwidth and the prototype vectors together with the weights. Unfortunately, this type of strong approach to supervision does not seem to work very well. There are two main drawbacks with this approach:

1. An attractive characteristic of RBFs is that they are efficient to train, if unsupervised methods are used. Even the orthogonal least-squares method can be run in a reasonably amount of time. However, this advantage is lost, if one resorts to full back-propagation. In general, the two-stage training of RBF is an efficiency feature of RBF networks.

2. The loss surface of RBFs has many local minima. This type of approach tends to get stuck in local minima from the point of view of generalization error.

Because of these characteristics of RBF networks, supervised training is rarely used. In fact, it has been shown in [342] that supervised training tends to increase the bandwidths and encourage generalized responses. When supervision is used, it should be used in a very controlled way by repeatedly testing performance on out-of-sample data in order to reduce the risk of overfitting.

5.3 Variations and Special Cases of RBF Networks

The above discussion only considers the case in which the supervised training is designed for numeric target variables. In practice, it is possible for the target variables to be binary. One possibility is to treat binary class labels in $\{-1, +1\}$ as numeric responses, and use the same approach of setting the weight vector according to Equation 5.4:

$$\overline{W}^T = (H^T H + \lambda I)^{-1} H^T \overline{y}$$

As discussed in Section 2.2.2.1 of Chapter 2, this solution is also equivalent to the Fisher discriminant and the Widrow-Hoff method. The main difference is that these methods are being applied on a hidden layer of increased dimensionality, which promotes better results in more complex distributions. It is also helpful to examine other loss functions that are commonly used in feed-forward neural networks for classification.

5.3.1 Classification with Perceptron Criterion

Using the notations introduced in the previous section, the prediction of the ith training instance is given by $\overline{W} \cdot \overline{H_i}$. Here, $\overline{H_i}$ represents the m-dimensional vector of activations in the hidden layer for the ith training instance $\overline{X_i}$. Then, as discussed in Section 1.2.1.1 of Chapter 1, the perceptron criterion corresponds to the following loss function:

$$L = \max\{-y_i(\overline{W} \cdot \overline{H_i}), 0\} \tag{5.9}$$

In addition, a Tikhonov regularization term with parameter $\lambda > 0$ is often added to the loss function.

Then, for each mini-batch S of training instances, let S^+ represent the misclassified instances. The misclassified instances are defined as those for which the loss L is non-zero. For such instances, applying the sign function to $\overline{H_i} \cdot \overline{W}$ will yield a prediction with opposite sign to the observed label y_i.

Then, for each mini-batch S of training instances, the following updates are used for the misclassified instances in S^+:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{(\overline{H_i}, y_i) \in S^+} y_i \overline{H_i} \tag{5.10}$$

Here, $\alpha > 0$ is the learning rate.

5.3.2 Classification with Hinge Loss

The hinge loss is used frequently in the support vector machine. Indeed, the use of hinge loss in the Gaussian RBF network can be viewed as a generalization of the support-vector machine. The hinge loss is a shifted version of the perceptron criterion:

$$L = \max\{1 - y_i(\overline{W} \cdot \overline{H_i}), 0\} \tag{5.11}$$

Because of the similarity in loss functions between the hinge loss and the perceptron criterion, the updates are also very similar. The main difference is that S^+ includes only misclassified points in the case of the perceptron criterion, whereas S^+ includes both misclassified points and marginally classified points in the case of hinge loss. This is because S^+ is defined by the set of points for which the loss function is non-zero, but (unlike the perceptron criterion) the hinge loss function is non-zero even for marginally classified points. Therefore, with this modified definition of S^+, the following updates are used:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{(\overline{H_i}, y_i) \in S^+} y_i \overline{H_i} \tag{5.12}$$

Here, $\alpha > 0$ is the learning rate, and $\lambda > 0$ is the regularization parameter. Note that one can easily define similar updates for the logistic loss function (cf. Exercise 2).

5.3.3 Example of Linear Separability Promoted by RBF

The main goal of the hidden layer is to perform a transformation that promotes linear separability, so that even linear classifiers work well on the transformed data. Both the perceptron and the linear support vector machine with hinge loss are known to perform poorly when the classes are not linearly separable. The Gaussian RBF classifier is able to separate out classes that are not linearly separable in the input space when loss functions such as the perceptron criterion and hinge loss are used. The key to this separability is the local transformation created by the hidden layer. An important point is that a Gaussian kernel with a small bandwidth often results in a situation where only a small number of hidden units in particular local regions get activated to significant non-zero values, whereas the other values are almost zeros. This is because of the exponentially decaying nature of the Gaussian function, which takes on near-zero values outside a particular locality. The identification of prototypes with cluster centers often divides the space into local regions, in which significant non-zero activation is achieved only in small portions of the space. As a practical matter, each local region of the space is assigned its own feature, corresponding to the hidden unit that is activated most strongly by it.

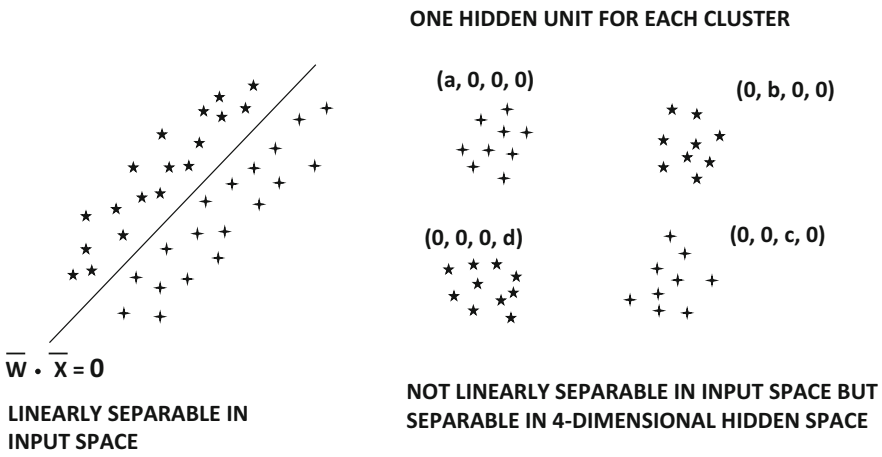


Figure 5.2: Revisiting Figure 1.4: The Gaussian RBF promotes separability because of the transformation to the hidden layer.

Examples of two data sets are illustrated in Figure 5.2. These data sets were introduced in Chapter 1 to illustrate cases that the (traditional) perceptron can or cannot solve. The traditional perceptron of Chapter 1 is able to find a solution for the data set on the left, but does not work well for the data set on the right. However, the transformation used by the Gaussian RBF method is able to address this issue of separability for the clustered data set on the right. Consider a case in which each of the centroids of the four clusters in Figure 5.2 is used as a prototype. This will result in a 4-dimensional hidden representation of the data. Note that the hidden dimensionality is higher than the input dimensionality, which is common in these settings. With appropriate choice of bandwidth, only one hidden unit will be activated strongly corresponding to the cluster identifier to which the point belongs. The other hidden units will be activated quite weakly, and will be close to 0. This will result in a rather sparse representation, as shown in Figure 5.2. We have shown the approximate 4-dimensional representations for the points in each cluster. The values of a, b, c, and d in Figure 5.2 will vary over the different points in the corresponding cluster, although they will always be strongly non-zero compared to the other coordinates. Note that one of the classes is defined by strongly non-zero values in the first and third dimensions, whereas the second class is defined by strongly non-zero values in the second and fourth dimensions. As a result, the weight vector $\overline{W} = [1, -1, 1, -1]$ will provide excellent non-linear separation between the two classes. The key point to understand is that the Gaussian RBF creates *local* features that result in separable distributions of the classes. This is exactly how a kernel support-vector machine achieves linear separability.

5.3.4 Application to Interpolation

One of the earliest applications of the Gaussian RBF was its use in interpolation of the value of a function over a set of points. The goal here is to perform *exact* interpolation of the provided points, so that the resulting function passes through all the input points. One can view interpolation as a special case of regression in which each training point is a prototype, and therefore the number of weights m in $\overline{W}$ is exactly equal to the number of training examples n. In such cases, it is possible to find a n-dimensional weight vector

$\overline{W}$ with zero error. In such a case, the activations $\overline{H_1} \ldots \overline{H_n}$ represent n-dimensional row vectors. Therefore, the matrix H obtained by stacking these row vectors on top of each other has a size of $n \times n$. Let $\overline{y} = [y_1, y_2, \ldots y_n]^T$ be the n-dimensional column vector of observed variables.

In linear regression, one attempts to minimize the loss function $||H\overline{W}^T - \overline{y}||^2$ in order to determine $\overline{W}$. This is because the matrix H is not square, and the system of equations $H\overline{W}^T = \overline{y}$ is over-complete. However, in the case of linear interpolation, the matrix H is square, and the system of equations is no longer over-complete. Therefore, it is possible to find an exact solution (with zero loss) satisfying the following system of equations:

$$H\overline{W}^T = \overline{y} \tag{5.13}$$

It can be shown that this system of equations has a unique solution when the training points are distinct from one another [323]. The value of the weight vector $\overline{W}^T$ can then be computed as follows:

$$\overline{W}^T = H^{-1}\overline{y} \tag{5.14}$$

It is noteworthy that this equation is a special case of Equation 5.6 because the pseudo-inverse of a square and non-singular matrix is the same as its inverse. In the case where the matrix H is non-singular, one can simplify the pseudo-inverse as follows:

$$\begin{aligned} H^+ &= (H^T H)^{-1} H^T \\ &= H^{-1} \underbrace{(H^T)^{-1} H^T}_{I} \\ &= H^{-1} \end{aligned}$$

Therefore, the case of linear interpolation is a special case of least-squares regression. Stated in another way, least-squares regression is a form of noisy interpolation, where it is impossible to fit the function through all the training points because of the limited degrees of freedom in the hidden layer. Relaxing the size of the hidden layer to the training data size allows exact interpolation. Exact interpolation is not necessarily better for computing the function value of out-of-sample points, because it might be the result of overfitting.

5.4 Relationship with Kernel Methods

The RBF network gains its power by mapping the input points into a high-dimensional hidden space in which linear models are sufficient to model nonlinearities. This is the same principle used by kernel methods like kernel regression and kernel SVMs. In fact, it can be shown that certain special cases of the RBF network reduce to kernel regression and kernel SVMs.

5.4.1 Kernel Regression as a Special Case of RBF Networks

The weight vector $\overline{W}$ in RBF networks is trained to minimize the squared loss of the following prediction function:

$$\hat{y}_i = \overline{H_i}\,\overline{W}^T = \sum_{j=1}^{m} w_j \Phi_j(\overline{X_i}) \tag{5.15}$$

Now consider the case in which the prototypes are the same as the training points, and therefore we set $\overline{\mu}_j = \overline{X_j}$ for each $j \in \{1 \ldots n\}$. Note that this approach is the same as that used in function interpolation, in which the prototypes are set to all the training points. Furthermore, each bandwidth σ is set to the same value. In such a case, one can write the above prediction function as follows:

$$\hat{y}_i = \sum_{j=1}^{n} w_j \exp\left(-\frac{||\overline{X_i} - \overline{X_j}||^2}{2\sigma^2}\right) \tag{5.16}$$

The exponentiated term on the right-hand side of Equation 5.16 can be written as the Gaussian kernel similarity between points $\overline{X_i}$ and $\overline{X_j}$. This similarity is denoted by $K(\overline{X_i}, \overline{X_j})$. Therefore, the prediction function becomes the following:

$$\hat{y}_i = \sum_{j=1}^{n} w_j K(\overline{X_i}, \overline{X_j}) \tag{5.17}$$

This prediction function is exactly the same as that used in kernel regression with bandwidth σ, where the prediction function $\hat{y}_i^{kernel}$ is defined[1] in terms of the *Lagrange multipliers* λ_j instead of weight w_j (see, for example, [6]):

$$\hat{y}_i^{kernel} = \sum_{j=1}^{n} \lambda_j y_j K(\overline{X_i}, \overline{X_j}) \tag{5.18}$$

Furthermore, the (squared) loss function is the same in the two cases. Therefore, a one-to-one correspondence will exist between the Gaussian RBF solutions and the kernel regression solutions, so that setting $w_j = \lambda_j y_j$ leads to the same value of the loss function. Therefore, their optimal values will be the same as well. In other words, the Gaussian RBF network provides the same results as kernel regression in the special case where the prototype vectors are set to the training points. However, the RBF network is more powerful and general because it can choose different prototype vectors; therefore, the RBF network can model cases that are not possible with kernel regression. In this sense, it is helpful to view the RBF network as a flexible neural variant of kernel methods.

5.4.2 Kernel SVM as a Special Case of RBF Networks

Like kernel regression, the kernel support vector machine (SVM) is also a special case of RBF networks. As in the case of kernel regression, the prototype vectors are set to the training points, and the bandwidths of all hidden units are set to the same value of σ. Furthermore, the weights w_j are learned in order to minimize the hinge loss of the prediction.

In such a case, it can be shown that the prediction function of the RBF network is as follows:

$$\hat{y}_i = \text{sign}\left\{\sum_{j=1}^{n} w_j \exp\left(-\frac{||\overline{X_i} - \overline{X_j}||^2}{2\sigma^2}\right)\right\} \tag{5.19}$$

$$\hat{y}_i = \text{sign}\left\{\sum_{j=1}^{n} w_j K(\overline{X_i}, \overline{X_j})\right\} \tag{5.20}$$

[1] A full explanation of the kernel regression prediction of Equation 5.18 is beyond the scope of this book. Readers are referred to [6].

It is instructive to compare this prediction function with that used in kernel SVMs (see, for example, [6]) with the Lagrange multipliers λ_j:

$$\hat{y}_i^{kernel} = \text{sign}\left\{\sum_{j=1}^{n} \lambda_j y_j K(\overline{X_i}, \overline{X_j})\right\} \tag{5.21}$$

This prediction function is of a similar form as that used in kernel SVMs, with the exception of a slight difference in the variables used. The hinge-loss is used as the objective function in both cases. By setting $w_j = \lambda_j y_j$ one obtains the same result in both cases in terms of the value of the loss function. Therefore, the optimal solutions in the kernel SVM and the RBF network will also be related according to the condition $w_j = \lambda_j y_j$. In other words, the kernel SVM is also a special case of RBF networks. Note that the weight w_j can also be considered the coefficient of each data point, when the *representer theorem* is used in kernel methods [6].

5.4.3 Observations

One can extend the arguments above to other linear models, such as the kernel Fisher discriminant and kernel logistic regression, by changing the loss function. In fact, the kernel Fisher discriminant can be obtained by simply using the binary variables as the targets and then applying kernel regression technique. However, since the Fisher discriminant works under the assumption of centered data, a bias needs to be added to the output layer to absorb any offsets from uncentered data. Therefore, the RBF network can simulate virtually any kernel method by choosing an appropriate loss function. A key point is that the RBF network provides more flexibility than kernel regression or classification. For example, one has much more flexibility in choosing the number of nodes in the hidden layer, as well as the number of prototypes. Choosing the prototypes wisely in a more economical way helps in both accuracy and efficiency. There are a number of key trade-offs associated with these choices:

1. Increasing the number of hidden units increases the complexity of the modeled function. It can be useful for modeling difficult functions, but it can cause overfitting, if the modeled function is not truly complex.

2. Increasing the number of hidden units increases the complexity of training.

One way of choosing the number of hidden units is to hold out a portion of the data, and estimate the accuracy of the model on the held-out set with different numbers of hidden units. The number of hidden units is then set to a value that optimizes this accuracy.

5.5 Summary

This chapter introduces radial basis function (RBF) networks, which represent a fundamentally different way of using the neural network architecture. Unlike feed-forward networks, the hidden layer and output layer are trained in a somewhat different way. The training of the hidden layer is unsupervised, whereas that of the output layer is supervised. The hidden layer usually has a larger number of nodes than the input layer. The key idea is to transform the data points into high-dimensional space with the use of locality-sensitive transformations, so that the transformed points become linearly separable. The approach

can be used for classification, regression, and linear interpolation by changing the nature of the loss function. In classification, one can use different types of loss functions such as the Widrow-Hoff loss, the hinge loss, and the logistic loss. Special cases of different loss functions specialize to well-known kernel methods such as kernel SVMs and kernel regression. The RBF network has rarely been used in recent years, and it has become a forgotten category of neural architectures. However, it has significant potential to be used in any scenario where kernel methods are used. Furthermore, it is possible to combine this approach with feed-forward architectures by using multi-layered representations following the first hidden layer.

5.6 Bibliographic Notes

RBF networks were proposed by Broomhead and Lowe [51] in the context of function interpolation. The separability of high-dimensional transformations is shown in Cover's work [84]. A review of RBF networks may be found in [363]. The books by Bishop [41] and Haykin [182] also provide good treatments of the topic. An overview of radial basis functions is provided in [57]. The proof of universal function approximation with RBF networks is provided in [173, 365]. An analysis of the approximation properties of RBF networks is provided in [366].

Efficient training algorithms for RBF networks are described in [347, 423]. An algorithm for learning the center locations in RBF networks is proposed in [530]. The use of decision trees to initialize RBF networks is discussed in [256]. The orthogonal least-squares algorithm was proposed in [65]. Early comparisons of supervised and unsupervised training of RBF networks are provided in [342]. According to this analysis, full supervision seems to increase the likelihood of the network getting trapped in local minima. Some ideas on improving the generalization power of RBF networks are provided in [43]. Incremental RBF networks are discussed in [125]. A detailed discussion of the relationship between RBF networks and kernel methods is provided in [430].

5.7 Exercises

Some exercises require additional knowledge about machine learning that is not discussed in this book. Exercises 5, 7, and 8 require additional knowledge of kernel methods, spectral clustering, and outlier detection.

1. Consider the following variant of radial basis function networks in which the hidden units take on either 0 or 1 values. The hidden unit takes on the value of 1, if the distance to a prototype vector is less than σ. Otherwise it takes on the value of 0. Discuss the relationship of this method to RBF networks, and its relative advantages/disadvantages.

2. Suppose that you use the sigmoid activation in the final layer to predict a binary class label as a probability in the output node of an RBF network. Set up a negative log-likelihood loss for this setting. Derive the gradient-descent updates for the weights in the final layer. How does this approach relate to the logistic regression methods discussed in Chapter 2? In which case will this approach perform better than logistic regression?

3. Discuss why an RBF network is a supervised variant of a nearest-neighbor classifier.

4. Discuss how you can extend the three multi-class models discussed in Chapter 2 to RBF networks. In particular discuss the extension of the (a) multi-class perceptron, (b) Weston-Watkins SVM, and (c) softmax classifier with RBF networks. Discuss how these models are more powerful than the ones discussed in Chapter 2.

5. Propose a method to extend RBF networks to unsupervised learning with autoencoders. What will you reconstruct in the output layer? A special case of your approach should be able to roughly simulate kernel singular value decomposition.

6. Suppose that you change your RBF network so that you keep only the top-k activations in the hidden layer, and set the remaining activations to 0. Discuss why such an approach will provide improved classification accuracy with limited data.

7. Combine the top-k method of constructing the RBF layer in Exercise 6 with the RBF autoencoder in Exercise 5 for unsupervised learning. Discuss why this approach will create representations that are better suited to clustering. Discuss the relationship of this method with spectral clustering.

8. The manifold view of outliers is to define them as points that do not naturally fit into the nonlinear manifolds of the training data. Discuss how you can use RBF networks for unsupervised outlier detection.

9. Suppose that instead of using the RBF function in the hidden layer, you use dot products between prototypes and data points for activation. Show that a special case of this setting reduces to a linear perceptron.

10. Discuss how you can modify the RBF autoencoder in Exercise 5 to perform semi-supervised classification, when you have a lot of unlabeled data, and a limited amount of labeled data.

Chapter 6

Restricted Boltzmann Machines

> "Available energy is the main object at stake in the struggle for existence and the evolution of the world."—Ludwig Boltzmann

6.1 Introduction

The restricted Boltzmann machine (RBM) is a fundamentally different model from the feed-forward network. Conventional neural networks are input-output mapping networks where a set of inputs is mapped to a set of outputs. On the other hand, RBMs are networks in which the probabilistic states of a network are learned for a set of inputs, which is useful for *unsupervised* modeling. While a feed-forward network minimizes a loss function of a prediction (computed from *observed* inputs) with respect to an *observed* output, a restricted Boltzmann machine models the joint probability distribution of the observed attributes together with some hidden attributes. Whereas traditional feed-forward networks have *directed* edges corresponding to the flow of computation from input to output, RBMs are *undirected* networks because they are designed to learn probabilistic *relationships* rather than input-output mappings. Restricted Boltzmann machines are probabilistic models that create latent representations of the underlying data points. Although an autoencoder can also be used to construct latent representations, a Boltzmann machine creates a *stochastic* hidden representation of each point. Most autoencoders (except for the variational autoencoder) create *deterministic* hidden representations of the data points. As a result, the RBM requires a fundamentally different way of training and using it.

At their core, RBMs are unsupervised models that generate latent feature representations of the data points; however, the learned representations can be combined with traditional backpropagation in a closely related feed-forward network (to the specific RBM at hand) for supervised applications. This type of combination of unsupervised and supervised learning is similar to the pretraining that is performed with a traditional autoencoder architecture (cf. Section 4.7 of Chapter 4). In fact, RBMs are credited for the popularization of pretraining in the early years. The idea was soon adapted to autoencoders, which are simpler to train because of their deterministic hidden states.

6.1.1 Historical Perspective

Restricted Boltzmann machines have evolved from a classical model in the neural networks literature, which is referred to as the *Hopfield network*. This network contains nodes containing binary states, which represent binary attribute values in the training data. The Hopfield network creates a *deterministic* model of the relationships among the different attributes by using weighted edges between nodes. Eventually, the Hopfield network evolved into the notion of a Boltzmann machine, which uses *probabilistic* states to represent the Bernoulli distributions of the binary attributes. The Boltzmann machine contains both visible states and hidden states. The visible states model the distributions of the observed data points, whereas the hidden states model the distribution of the latent (hidden) variables. The parameters of the connections among the various states regulate their joint distribution. The goal is to learn the model parameters so that the likelihood of the model is maximized. The Boltzmann machine is a member of the family of (undirected) probabilistic graphical models. Eventually, the Boltzmann machine evolved into the *restricted* Boltzmann Machine (RBM). The main difference between the Boltzmann machine and the restricted Boltzmann machine is that the latter only allows connections between hidden units and visible units. This simplification is very useful from a practical point of view, because it allows the design of more efficient training algorithms. The RBM is a special case of the class of probabilistic graphical models known as *Markov random fields*.

In the initial years, RBMs were considered too slow to train and were therefore not very popular. However, at the turn of the century, faster algorithms were proposed for this class of models. Furthermore, they received some prominence as one of the ensemble components of the entry [414] winning the Netflix prize contest [577]. RBMs are generally used for unsupervised applications like matrix factorization, latent modeling, and dimensionality reduction, although there are many ways of extending them to the supervised case. It is noteworthy that RBMs usually work with binary states in their most natural form, although it is possible to work with other data types. Most of the discussion in this chapter will be restricted to units with binary states. The successful training of deep networks with RBMs preceded successful training experiences with conventional neural networks. In other words, it was shown how multiple RBMs could be stacked to create deep networks and train them effectively, before similar ideas were generalized to conventional networks.

Chapter Organization

This chapter is organized as follows. The next section will introduce Hopfield networks, which was the precursor to the Boltzmann family of models. The Boltzmann machine is introduced in Section 6.3. Restricted Boltzmann machines are introduced in Section 6.4. Applications of restricted Boltzmann machines are discussed in Section 6.5. The use of RBMs for generalized data types beyond binary representations is discussed in Section 6.6.

The process of stacking multiple restricted Boltzmann machines in order to create deep networks is discussed in Section 6.7. A summary is given in Section 6.8.

6.2 Hopfield Networks

Hopfield networks were proposed in 1982 [207] as a model to store memory. A Hopfield network is an undirected network, in which the d units (or neurons) are indexed by values drawn from $\{1 \ldots d\}$. Each connection is of the form (i, j), where each i and j is a neuron drawn from $\{1 \ldots d\}$. Each connection (i, j) is undirected, and is associated with a weight $w_{ij} = w_{ji}$. Although all pairs of nodes are assumed to have connections between them, setting w_{ij} to 0 has the effect of dropping the connection (i, j). The weight w_{ii} is set to 0, and therefore there are no self-loops. Each neuron i is associated with state s_i. An important assumption in the Hopfield network is that each s_i is a binary value drawn from $\{0, 1\}$, although one can use other conventions such as $\{-1, +1\}$. The ith node also has a bias b_i associated with it; large values of b_i encourage the ith state to be 1. The Hopfield network is an undirected model of symmetric relationships between attributes, and therefore the weights always satisfy $w_{ij} = w_{ji}$.

Each binary state in the Hopfield network corresponds to a dimension in the (binary) training data set. Therefore, if a d-dimensional training data set needs to be memorized, we need a Hopfield network with d units. The ith state in the network corresponds to the ith bit in a particular training example. The values of the states represent the binary attribute values from a training example. The weights in the Hopfield network are its parameters; large positive weights between pairs of states are indicative of high degree of positive correlation in state values, whereas large negative weights are indicative of high negative correlation. An example of a Hopfield network with an associated training data set is shown in Figure 6.1. In this case, the Hopfield network is fully connected, and the six visible states correspond to the six binary attributes in the training data.

The Hopfield network uses an optimization model to learn the weight parameters so that the weights can capture that positive and negative relationships among the attributes of the training data set. The objective function of a Hopfield network is also referred to as its *energy function*, which is analogous to the loss function of a traditional feed-forward neural network. The energy function of a Hopfield network is set up in such a way that minimizing this function encourages nodes pairs connected with large positive weights to have similar states, and pairs connected with large negative weights to have different states. The training phase of a Hopfield network, therefore, learns the weights of edges in order to minimize the energy when the states in the Hopfield network are fixed to the binary attribute values in the individual training points. Therefore, learning the weights of the Hopfield network implicitly builds an unsupervised *model* of the training data set. The energy E of a particular combination of states $\overline{s} = (s_1, \ldots s_d)$ of the Hopfield network can be defined as follows:

$$E = -\sum_i b_i s_i - \sum_{i,j:i<j} w_{ij} s_i s_j \tag{6.1}$$

The term $-b_i s_i$ encourages units with large biases to be on. Similarly, the term $-w_{ij} s_i s_j$ encourages s_i and s_j to be similar when $w_{ij} > 0$. In other words, positive weights will cause state "attraction" and negative weights will cause state "repulsion." For a small training data set, this type of modeling results in memorization, which enables one to retrieve training data points from similar, incomplete, or corrupted query points by exploring local minima of the energy function near these query points. In other words, by learning the weights of

a Hopfield network, one is implicitly memorizing the training examples, although there is a relatively conservative limit of the number of examples that can be memorized from a Hopfield network containing d units. This limit is also referred to as the *capacity* of the model.

6.2.1 Optimal State Configurations of a Trained Network

A trained Hopfield contains many local optima, each of which corresponds to either a memorized point from the training data, or a representative point in a dense region of the training data. Before discussing the training of the weights of the Hopfield network, we will discuss the methodology for finding the local energy minimum of a Hopfield network when the trained weights are already given. A local minimum is defined as a combination of states in which flipping any particular bit of the network does not reduce the energy further. The training process sets the weights in such a way that the instances in the training data tend to be local minima in the Hopfield network.

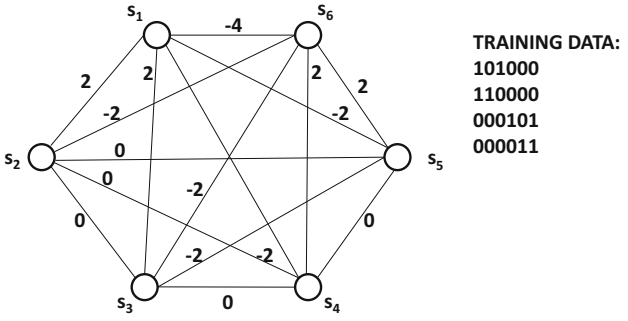


Figure 6.1: A Hopfield network with 6 visible states corresponding to 6-dimensional training data.

Finding the optimal state configuration helps the Hopfield network in recalling memories. The Hopfield network inherently learns *associative memories* because, given an input set of states (i.e., input pattern of bits), it repeatedly flips bits to improve the objective function until it finds a pattern where one cannot improve the objective function further. This local minimum (final combination of states) is often only a few bits away from the starting pattern (initial set of states), and therefore one *recalls* a closely related pattern at which a local minimum is found. Furthermore, this final pattern is often a member of the training data set (because the weights were learned using that data). In a sense, Hopfield networks provide a route towards *content-addressable memory*.

Given a starting combination of states, how can one learn the closest local minimum once the weights have already been fixed? One can use a threshold update rule to update each state in the network in order to move it towards the global energy minimum. In order to understand this point, let us compare the energy of the network between the cases when the state s_i is set to 1, and the one in which s_i is set to 0. Therefore, one can substitute two different values of s_i into Equation 6.1 to obtain the following value of the *energy gap*:

$$\Delta E_i = E_{s_i=0} - E_{s_i=1} = b_i + \sum_{j:j \neq i} w_{ij} s_j \tag{6.2}$$

This value must be larger than 0 in order for a flip of state s_i from 0 to 1 to be attractive. Therefore, one obtains the following update rule for each state s_i:

$$s_i = \begin{cases} 1 & \text{if } \sum_{j:j \neq i} w_{ij} s_j + b_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

The above rule is iteratively used to test each state s_i and then flip the state if needed to satisfy the condition. If one is given the weights and the biases in the network at any particular time, it is possible to find a local energy minimum in terms of the states by repeatedly using the update rule above.

The local minima of a Hopfield network depend on its trained weights. Therefore, in order to "recall" a memory, one only has to provide a d-dimensional vector similar to the stored memory, and the Hopfield network will find the local minimum that is similar to this point by using it as a starting state. This type of associative memory recall is also common in humans, who often retrieve memories through a similar process of association. One can also provide a partial vector of initial states and use it to recover other states. Consider the Hopfield network shown in Figure 6.1. Note that the weights are set in such a way that each of the four training vectors in the figure will have low energy. However, there are some spurious minima such as 111000 as well. Therefore, it is not guaranteed that the local minima will always correspond to the points in the training data. However, the local minima do correspond to some key characteristics of the training data. For example, consider the spurious minimum corresponding to 111000. It is noteworthy that the first three bits are positively correlated, whereas the last three bits are also positively correlated. As a result, this minimum value of 111000 does reflect a broad pattern in the underlying data even though it is not explicitly present in the training data. It is also noteworthy that the weights of this network are closely related to the patterns in the training data. For example, the elements within the first three bits and last three bits are each positively correlated within that particular group of three bits. Furthermore, there are negative correlations *across* the two sets of elements. Consequently, the edges *within* each of the sets $\{s_1, s_2, s_3\}$ and $\{s_4, s_5, s_6\}$ tend to be positive, and those *across* these two sets are negative. Setting the weights in this data-specific way is the task of the training phase (cf. Section 6.2.2).

The iterative state update rule will arrive at one of the many local minima of the Hopfield network, depending on the initial state vector. Each of these local minima can be one of the learned "memories" from the training data set, and the closest memory to the initial state vector will be reached. These memories are implicitly stored in the weights learned during the training phase. However, it is possible for the Hopfield network to make mistakes, where closely related training patterns are merged into a single (deeper) minimum. For example, if the training data contains 1110111101 and 1110111110, the Hopfield network might learn 1110111111 as a local minimum. Therefore, in some queries, one might recover a pattern that is a small number of bits away from a pattern actually present in the training data. However, this is only a form of model generalization in which the Hopfield network is storing representative "cluster" centers instead of individual training points. In other words, the model starts generalizing instead of memorizing when the amount of data exceeds the capacity of the model; after all, Hopfield networks build unsupervised models from training data.

The Hopfield network can be used for recalling associative memories, correcting corrupted data, or for attribute completion. The tasks of recalling associative memories and cleaning corrupted data are similar. In both cases, one uses the corrupted input (or target input for associative recall) as the starting state, and uses the final state as the cleaned

output (or recalled output). In attribute completion, the state vector is initialized by setting observed states to their known values and unobserved states randomly. At this point, only the unobserved states are updated to convergence. The bit values of these states at convergence provide the completed representation.

6.2.2 Training a Hopfield Network

For a given training data set, one needs to learn the weights, so that the local minima of this network lie near instances (or dense regions) of the training data set. Hopfield networks are trained with the *Hebbian learning rule*. According to the biological motivation of Hebbian learning, a synapse between two neurons is strengthened when the neurons on either side of the synapse have highly correlated outputs. Let $x_{ij} \in \{0, 1\}$ represent the jth bit of the ith training point. The number of training instances is assumed to be n. The Hebbian learning rule sets the weights of the network as follows:

$$w_{ij} = 4 \frac{\sum_{k=1}^{n} (x_{ki} - 0.5) \cdot (x_{kj} - 0.5)}{n} \tag{6.4}$$

One way of understanding this rule is that if two bits, i and j, in the training data are positively correlated, then the value $(x_{ki} - 0.5) \cdot (x_{kj} - 0.5)$ will usually be positive. As a result, the weights between the corresponding units will also be set to positive values. On the other hand, if two bits generally disagree, then the weights will be set to negative values. One can also use this rule without normalizing the denominator:

$$w_{ij} = 4 \sum_{k=1}^{n} (x_{ki} - 0.5) \cdot (x_{kj} - 0.5) \tag{6.5}$$

In practice, one often wants to develop incremental learning algorithms for point-specific updates. One can update w_{ij} with only the kth training data point as follows:

$$w_{ij} \Leftarrow w_{ij} + 4(x_{ki} - 0.5) \cdot (x_{kj} - 0.5) \quad \forall i, j$$

The bias b_i can be updated by assuming that a single dummy state is always on, and the bias represents the weight between the dummy and the ith state:

$$b_i \Leftarrow b_i + 2(x_{ki} - 0.5) \quad \forall i$$

In cases where the convention is to draw the state vectors from $\{-1, +1\}$, the above rule simplifies to the following:

$$w_{ij} \Leftarrow w_{ij} + x_{ki} x_{kj} \quad \forall i, j$$
$$b_i \Leftarrow b_i + x_{ki} \quad \forall i$$

There are other learning rules, such as the *Storkey learning rule*, that are commonly used. Refer to the bibliographic notes.

Capacity of a Hopfield Network

What is the size of the training data that a Hopfield network with d visible units can store without causing errors in associative recall? It can be shown that the *storage capacity* of a Hopfield network with d units is only about $0.15 \cdot d$ training examples. Since each training

example contains d bits, it follows that the Hopfield network can store only about $0.15\,d^2$ bits. This is not an efficient form of storage because the number of weights in the network is given by $d(d-1)/2 = O(d^2)$. Furthermore, the weights are not binary and they can be shown to require $O(\log(d))$ bits. When the number of training examples is large, many errors will be made (in associative recall). These errors represent the *generalized* predictions from more data. Although it might seem that this type of generalization is useful for machine learning, there are limitations in using Hopfield networks for such applications.

6.2.3 Building a Toy Recommender and Its Limitations

Hopfield networks are often used for memorization-centric applications rather than the typical machine-learning applications requiring generalization. In order to understand the limits of a Hopfield network, we will consider an application associated with binary collaborative filtering. Since Hopfield networks work with binary data, we will assume the case of *implicit feedback* data in which user is associated with a set of binary attributes corresponding to whether or not they have watched the corresponding movies. Consider a situation in which the user Bob has watched movies *Shrek* and *Aladdin*, whereas the user Alice has watched *Gandhi*, *Nero*, and *Terminator*. It is easy to construct a fully connected Hopfield network on the universe of all movies and set the watched states to 1 and all other states to 0. This configuration can be used for each training point in order to update the weights. Of course, this approach can be extremely expensive if the base number of states (movies) is very large. For a database containing 10^6 movies, we would have 10^{12} edges, most of which will connect states containing zero values. This is because such type of implicit feedback data is often sparse, and most states will take on zero values.

One way of addressing this problem is to use *negative sampling*. In this approach, each user has their own Hopfield network containing their watched movies and a small sample of the movies that were not watched by them. For example, one might randomly sample 20 unwatched movies (of Alice) and create a Hopfield network containing $20 + 3 = 23$ states (including the watched movies). Bob's Hopfield network will containing $20 + 2 = 22$ states, and the unwatched samples might also be quite different. However, for pairs of movies that are common between the two networks, the weights will be shared. During training, all edge weights are initialized to 0. One can use repeated iterations of training over the different Hopfield networks to learn their shared weights (with the same algorithm discussed earlier). The main difference is that iterating over the different training points will lead to iterating over different Hopfield networks, each of which contains a small subset of the base network. Typically, only a small subset of the 10^{12} edges will be present in each of these networks, and most edges will never be encountered in any network. Such edges will implicitly be assumed to have weights of zero.

Now imagine a user Mary, who has watched *E.T.* and *Shrek*. We would like to recommend movies to this user. We use the full Hopfield network with only the non-zero edges present. We initialize the states for *E.T.* and *Shrek* to 1, and all other states to 0. Subsequently, we allow the updates of all states (other than *E.T.* and *Shrek*) in order to identify the minimum energy configuration of the Hopfield network. All states that are set to 1 during the updates can be recommended to the user. However, we would ideally like to have an *ordering* of the top recommended movies. One way of providing an ordering of all movies is to use the *energy gap* between the two states of each movie in order to rank the movies. The energy gap is computed only after the minimum energy configuration has been found. This approach is, however, quite naive because the final configuration of the Hopfield network is a deterministic one containing binary values, whereas the extrapolated values can only be estimated in terms of *probabilities*. For example, it would be much more natural to use

some function of the energy gap (e.g., sigmoid) in order to create probabilistic estimations. Furthermore, it would be helpful to be able to capture correlated sets of movies with some notion of latent (or hidden) states. Clearly, we need techniques in order to increase the expressive power of the Hopfield network.

6.2.4 Increasing the Expressive Power of the Hopfield Network

Although it is not standard practice, one can add *hidden units* to a Hopfield network to increase its expressive power. The hidden states serve the purpose of capturing the latent structure of the data. The weights of connections between hidden and visible units will capture the relationship between the latent structure and the training data. In some cases, it is possible to approximately represent the data only in terms of a small number of hidden states. For example, if the data contains two tightly knit clusters, one can capture this setting in two hidden states. Consider the case in which we enhance the Hopfield network of Figure 6.1 and add two hidden units. The resulting network is shown in Figure 6.2. The edges with near-zero weights have been dropped from the figure for clarity. Even though the original data is defined in terms of six bits, the two hidden units provide a *hidden* representation of the data in terms of two bits. This hidden representation is a compressed version of the data, which tells us something about the pattern at hand. In essence, all patterns are compressed to the pattern 10 or 01, depending on whether the first three bits or the last three bits dominate the training pattern. If one fixes the hidden states of the Hopfield network to 10 and randomly initializes the visible states, then one would often obtain the pattern 111000 on repeatedly using the state-wise update rule of Equation 6.3. One also obtains the pattern 0001111 as the final resting point when one starts with the hidden state 01. Notably, the patterns 000111 and 111000 are close approximations of the two types of patterns in the data, which is what one would expect from a compression technique. If we provide an incomplete version of the visible units, and then iteratively update the other states with the update rule of Equation 6.3, one would often arrive at either 000111 and 111000 depending on how the bits in the incomplete representation are distributed. If we add hidden units to a Hopfield network *and* allow the states to be probabilistic (rather than deterministic), we obtain a Boltzmann machine. This is the reason that Boltzmann machines can be viewed as *stochastic Hopfield networks with hidden units*.

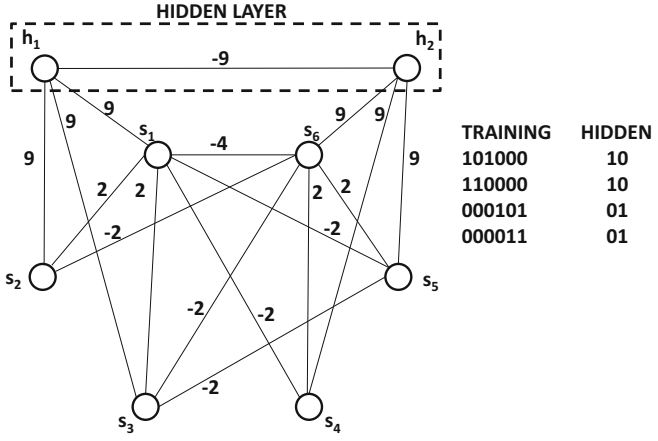


Figure 6.2: The Hopfield network with two hidden nodes

6.3 The Boltzmann Machine

Throughout this section, we assume that the Boltzmann machine contains a total of $q = (m+d)$ states, where d is the number of visible states and m is the number of hidden states. A particular state configuration is defined by the value of the state vector $\overline{s} = (s_1 \ldots s_q)$. If one explicitly wants to demarcate the visible and hidden states in $\overline{s}$, then the state vector $\overline{s}$ can be written as the pair $(\overline{v}, \overline{h})$, where $\overline{v}$ denotes the set of visible units and $\overline{h}$ denotes the set of hidden units. The states in $(\overline{v}, \overline{h})$ represent exactly the same set as $\overline{s} = \{s_1 \ldots s_q\}$, except that the visible and hidden units are explicitly demarcated in the former.

The Boltzmann machine is a probabilistic generalization of a Hopfield network. A Hopfield network *deterministically* sets each state s_i to either 1 or 0, depending on whether the energy gap ΔE_i of the state s_i is positive or negative. Recall that the energy gap of the ith unit is defined as the difference in energy between its two configurations (with other states being fixed to pre-defined values):

$$\Delta E_i = E_{s_i=0} - E_{s_i=1} = b_i + \sum_{j:j \neq i} w_{ij} s_j \tag{6.6}$$

The Hopfield network deterministically sets the value of s_i to 1, when the energy gap is positive. On the other hand, a Boltzmann machine assigns a *probability* to s_i depending on the energy gap. Positive energy gaps are assigned probabilities that are larger than 0.5. The probability of state s_i is defined by applying the sigmoid function to the energy gap:

$$P(s_i = 1 | s_1, \ldots s_{i-1}, s_{i+1}, s_q) = \frac{1}{1 + \exp(-\Delta E_i)} \tag{6.7}$$

Note that the state s_i is now a Bernoulli random variable and a zero energy gap leads to a probability of 0.5 for each binary outcome of the state.

For a particular set of parameters w_{ij} and b_i, the Boltzmann machine defines a probability distribution over various state configurations. The energy of a particular configuration $\overline{s} = (\overline{v}, \overline{h})$ is denoted by $E(\overline{s}) = E([\overline{v}, \overline{h}])$, and is defined in a similar way to the Hopfield network as follows:

$$E(\overline{s}) = -\sum_i b_i s_i - \sum_{i,j:i<j} w_{ij} s_i s_j \tag{6.8}$$

However, these configurations are only probabilistically known in the case of the Boltzmann machine (according to Equation 6.7). The conditional distribution of Equation 6.7 follows from a more fundamental definition of the unconditional probability $P(\overline{s})$ of a particular configuration $\overline{s}$:

$$P(\overline{s}) \propto \exp(-E(\overline{s})) = \frac{1}{Z} \exp(-E(\overline{s})) \tag{6.9}$$

The normalization factor Z is defined so that the probabilities over all possible configurations sum to 1:

$$Z = \sum_{\overline{s}} \exp(-E(\overline{s})) \tag{6.10}$$

The normalization factor Z is also referred to as the *partition function*. In general, the explicit computation of the partition function is hard, because it contains an exponential number of terms corresponding to all possible configurations of states. Because of the intractability of the partition function, exact computation of $P(\overline{s}) = P(\overline{v}, \overline{h})$ is not possible. Nevertheless, the computation of many types of conditional probabilities (e.g., $P(\overline{v}|\overline{h})$) is possible, because such conditional probabilities are ratios and the intractable normalization

factor gets canceled out from the computation. For example, the conditional probability of Equation 6.7 follows from the more fundamental definition of the probability of a configuration (cf. Equation 6.9) as follows:

$$P(s_i = 1|s_1, \ldots s_{i-1}, s_{i+1}, s_q) = \frac{P(s_1, \ldots s_{i-1}, \overbrace{1}^{s_i}, s_{i+1}, s_q)}{P(s_1, \ldots s_{i-1}, \underbrace{1}_{s_i}, s_{i+1}, s_q) + P(s_1, \ldots s_{i-1}, \underbrace{0}_{s_i}, s_{i+1}, s_q)}$$

$$= \frac{\exp(-E_{s_i=1})}{\exp(-E_{s_i=1}) + \exp(-E_{s_i=0})} = \frac{1}{1 + \exp(E_{s_i=1} - E_{s_i=0})}$$

$$= \frac{1}{1 + \exp(-\Delta E_i)} = \text{Sigmoid}(\Delta E_i)$$

This is the same condition as Equation 6.9. One can also see that the logistic sigmoid function finds its roots in notions of energy from statistical physics.

One way of thinking about the benefit of setting these states probabilistically is that we can now sample from these states to create new data points that look like the original data. This makes Boltzmann machines probabilistic models rather than deterministic ones. Many generative models in machine learning (e.g., Gaussian mixture models for clustering) use a sequential process of first sampling the hidden state(s) from a prior, and then generating visible observations conditionally on the hidden state(s). This is not the case in the Boltzmann machine, in which the dependence between all pairs of states is *undirected*; the visible states depend as much on the hidden states as the hidden states depend on visible states. As a result, the generation of data with a Boltzmann machine can be more challenging than in many other generative models.

6.3.1 How a Boltzmann Machine Generates Data

In a Boltzmann machine, the dynamics of the data generation is complicated by the circular dependencies among the states based on Equation 6.7. Therefore, we need an iterative process to generate sample data points from the Boltzmann machine so that Equation 6.7 is satisfied for all states. A Boltzmann machine iteratively samples the states using a conditional distribution generated from the state values in the previous iteration until *thermal equilibrium* is reached. The notion of thermal equilibrium means that we start at a random set of states, use Equation 6.7 to compute their conditional probabilities, and then sample the values of the states again using these probabilities. Note that we can iteratively generate s_i by using $P(s_i|s_1 \ldots s_{i-1}, s_{i+1}, \ldots s_q)$ in Equation 6.7. After running this process for a long time, the sampled values of the visible states provide us with random samples of generated data points. The time required to reach thermal equilibrium is referred to as the *burn-in time* of the procedure. This approach is referred to as *Gibbs sampling* or *Markov Chain Monte Carlo (MCMC) sampling*.

At thermal equilibrium, the generated points will represent the model captured by the Boltzmann machine. Note that the dimensions in the generated data points will be correlated with one another depending on the weights between various states. States with large weights between them will tend to be heavily correlated. For example, in a text-mining application in which the states correspond to the presence of words, there will be correlations among words belonging to a topic. Therefore, if a Boltzmann machine has been trained properly on a text data set, it will generate vectors containing these types of word correlations at thermal equilibrium, even when the states are randomly initialized. It is noticeable that

even generating a set of data points with the Boltzmann machine is a more complicated process compared to many other probabilistic models. For example, generating data points from a Gaussian mixture model only requires to sample points directly from the probability distribution of a sampled mixture component. On the other hand, the undirected nature of the Boltzmann machine forces us to run the process to thermal equilibrium just to generate samples. It is, therefore, an even more difficult to task to learn the weights between states for a given training data set.

6.3.2 Learning the Weights of a Boltzmann Machine

In a Boltzmann machine, we want to learn the weights in such a way so as to maximize the log-likelihood of the specific training data set at hand. The log-likelihoods of individual states are computed by using the logarithm of the probabilities in Equation 6.9. Therefore, by taking the logarithm of Equation 6.9, we obtain the following:

$$\log[P(\overline{s})] = -E(\overline{s}) - \log(Z) \tag{6.11}$$

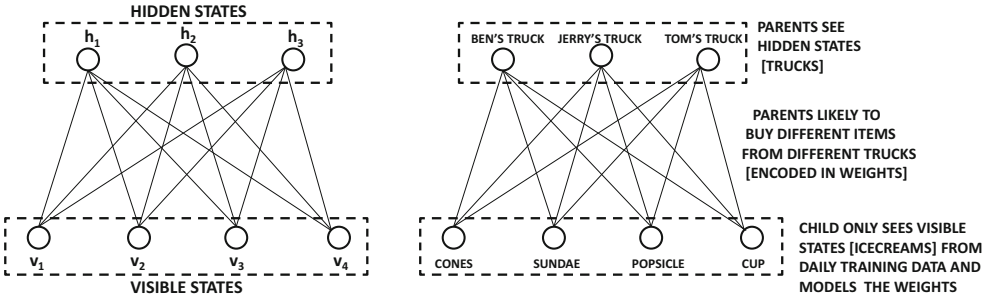
Therefore, computing $\frac{\partial \log[P(\overline{s})]}{\partial w_{ij}}$ requires the computation of the negative derivative of the energy, although we have an additional term involving the partition function. The energy function of Equation 6.8 is linear in the weight w_{ij} with coefficient of $-s_i s_j$. Therefore, the partial derivative of the energy with respect to the weight w_{ij} is $-s_i s_j$. As a result, one can show the following:

$$\frac{\partial \log[P(\overline{s})]}{\partial w_{ij}} = \langle s_i, s_j \rangle_{data} - \langle s_i, s_j \rangle_{model} \tag{6.12}$$

Here, $\langle s_i, s_j \rangle_{data}$ represents the averaged value of $s_i s_j$ obtained by running the generative process of Section 6.3.1, when the visible states are clamped to attribute values in a training point. The averaging is done over a mini-batch of training points. Similarly, $\langle s_i, s_j \rangle_{model}$ represents the averaged value of $s_i s_j$ at thermal equilibrium without fixing visible states to training points and simply running the generative process of Section 6.3.1. In this case, the averaging is done over multiple instances of running the process to thermal equilibrium. Intuitively, we want to strengthen the weights of edges between states, which tend to be *differentially* turned on together (compared to the unrestricted model), when the visible states are fixed to the training data points. This is precisely what is achieved by the update above, which uses the data- and model-centric difference in the value of $\langle s_i, s_j \rangle$. From the above discussion, it is clear that two types of samples need to be generated in order to perform the updates:

1. **Data-centric samples:** The first type of sample fixes the visible states to a randomly chosen vector from the training data set. The hidden states are initialized to random values drawn from Bernoulli distribution with probability 0.5. Then the probability of each hidden state is recomputed according to Equation 6.7. Samples of the hidden states are regenerated from these probabilities. This process is repeated for a while, so that thermal equilibrium is reached. The values of the hidden variables at this point provide the required samples. Note that the visible states are clamped to the corresponding attributes of the relevant training data vector, and therefore they do not need to be sampled.

2. **Model samples:** The second type of sample does not put any constraints on states, and one simply wants samples from the unrestricted model. The approach is the same as discussed above, except that both the visible and hidden states are initialized to random values, and updates are continuously performed until thermal equilibrium is reached.



(a) A Boltzmann machine (b) Alice's ice-cream trucks

Figure 6.3: A Restricted Boltzmann machine. Note the *restriction* of there being no interactions among either visible or hidden units.

These samples help us create an update rule for the weights. From the first type of sample, one can compute $\langle s_i, s_j \rangle_{data}$, which represents the correlations between the states of nodes i and j, when the visible vectors are fixed to a vector in the training data $\mathcal{D}$ and the hidden states are allowed to vary. Since a mini-batch of training vectors is used, one obtains multiple samples of the state vectors. The value of $\langle s_i, s_j \rangle$ is computed as the average product over all such state vectors that are obtained from Gibbs sampling. Similarly, one can estimate the value of $\langle s_i, s_j \rangle_{model}$ using the average product of s_i and s_j from the model-centric samples obtained from Gibbs sampling. Once these values have been computed, the following update is used:

$$w_{ij} \Leftarrow w_{ij} + \alpha \underbrace{(\langle s_i, s_j \rangle_{data} - \langle s_i, s_j \rangle_{model})}_{\text{Partial derivative of log probability}} \tag{6.13}$$

The update rule for the bias is similar, except that the state s_j is set to 1. One can achieve this by using a dummy bias unit that is visible and is connected to all states:

$$b_i \Leftarrow b_i + \alpha (\langle s_i, 1 \rangle_{data} - \langle s_i, 1 \rangle_{model}) \tag{6.14}$$

Note that the value of $\langle s_i, 1 \rangle$ is simply the average of the sampled values of s_i for a mini-batch of training examples from either the data-centric samples or the model-centric samples.

This approach is similar to the Hebbian update rule of a Hopfield net, except that we are also removing the effect of model-centric correlations in the update. The removal of model-centric correlations is required to account for the effect of the partition function within the expression of the log probability in Equation 6.11. The main problem with the aforementioned update rule is that it is slow in practice. This is because of the Monte Carlo sampling procedure, which requires a large number of samples in order to reach thermal equilibrium. There are faster approximations to this tedious process. In the next section, we will discuss this approach in the context of a simplified version of the Boltzmann machine, which is referred to as the *restricted* Boltzmann machine.

6.4 Restricted Boltzmann Machines

In the Boltzmann machine, the connections among hidden and visible units can be arbitrary. For example, two hidden states might contain edges between them, and so might two visible states. This type of generalized assumption creates unnecessary complexity. A natural special case of the Boltzmann machine is the *restricted* Boltzmann machine (RBM), which is bipartite, and the connections are allowed only between hidden and visible units. An example of a restricted Boltzmann machine is shown in Figure 6.3(a). In this particular example, there are three hidden nodes and four visible nodes. Each hidden state is connected to one or more visible states, although there are no connections between pairs of hidden states, and between pairs of visible states. The restricted Boltzmann machine is also referred to as a *harmonium* [457].

We assume that the hidden units are $h_1 \ldots h_m$ and the visible units are $v_1 \ldots v_d$. The bias associated with the visible node v_i be denoted by $b_i^{(v)}$, and the bias associated with hidden node h_j is denoted by $b_j^{(h)}$. Note the superscripts in order to distinguish between the biases of visible and hidden nodes. The weight of the edge between visible node v_i and hidden node h_j is denoted by w_{ij}. The notations for the weights are also slightly different for the restricted Boltzmann machine (compared to the Boltzmann machine) because the hidden and visible units are indexed separately. For example, we no longer have $w_{ij} = w_{ji}$ because the first index i always belongs to a visible node and the second index j belongs to a hidden node. It is important to keep these notational differences in mind while extrapolating the equations from the previous section.

In order to provide better interpretability, we will use a running example throughout this section, which we refer to as the example of "Alice's ice-cream trucks" based on the Boltzmann machine in Figure 6.3(b). Imagine a situation in which the training data corresponds to four bits representing the ice-creams received by Alice from her parents each day. These represent the visible states in our example. Therefore, Alice can collect 4-dimensional training points, as she receives (between 0 and 4) ice-creams of different types each day. However, the ice-creams are bought for Alice by her parents from one[1] or more of three trucks shown as the hidden states in the same figure. The identity of these trucks is hidden from Alice, although she knows that there are three trucks from which her parents procure the ice-creams (and more than one truck can be used to construct a single day's ice-cream set). Alice's parents are indecisive people, and their decision-making process is unusual because they change their mind about the selected ice-creams after selecting the trucks and vice versa. The likelihood of a particular ice-cream being picked depends on the trucks selected as well as the weights to these trucks. Similarly, the likelihood of a truck being selected depends on the ice-creams that one intends to buy and the same weights. Therefore, Alice's parents can keep changing their mind about selecting ice-creams after selecting trucks and about selecting trucks after selecting ice-creams (for a while) until they reach a final decision each day. As we will see, this *circular* relationship is the characteristic of undirected models, and process used by Alice's parents is similar to Gibb's sampling.

The use of the bipartite restriction greatly simplifies inference algorithms in RBMs, while retaining the application-centric power of the approach. If we know all the values of the visible units (as is common when a training data point is provided), the probabilities of the hidden units can be computed in one step without having to go through the laborious process of Gibbs sampling. For example, the probability of each hidden unit taking on the

[1] This example is tricky in terms of semantic interpretability for the case in which no trucks are selected. Even in that case, the probabilities of various ice-creams turn out to be non-zero depending on the bias. One can explain such cases by adding a dummy truck that is always selected.

value of 1 can be written directly as a logistic function of the values of visible units. In other words, we can apply Equation 6.7 to the restricted Boltzmann machine to obtain the following:

$$P(h_j = 1|\overline{v}) = \frac{1}{1 + \exp(-b_j^{(h)} - \sum_{i=1}^{d} v_i w_{ij})} \tag{6.15}$$

This result follows directly from Equation 6.7, which relates the state probabilities to the energy gap ΔE_j between $h_j = 0$ and $h_j = 1$. The value of ΔE_j is $b_j + \sum_i v_i w_{ij}$ when the visible states are observed. The main difference from an unrestricted Boltzmann machine is that the right-hand side of the above equation does not contain any (unknown) hidden variables and only the hidden variables. This relationship is also useful in creating a reduced representation of each training vector, once the weights have been learned. Specifically, for a Boltzmann machine with m hidden units, one can set the value of the jth hidden value to the probability computed in Equation 6.15. Note that such an approach provides a real-valued reduced representation of the binary data. One can also write the above equation using a sigmoid function:

$$P(h_j = 1|\overline{v}) = \text{Sigmoid}\left(b_j^{(h)} + \sum_{i=1}^{d} v_i w_{ij}\right) \tag{6.16}$$

One can also use a sample of the hidden states to generate the data points in one step. This is because the relationship between the visible units and the hidden units is similar in the undirected and bipartite architecture of the RBM. In other words, we can use Equation 6.7 to obtain the following:

$$P(v_i = 1|\overline{h}) = \frac{1}{1 + \exp(-b_i^{(v)} - \sum_{j=1}^{m} h_j w_{ij})} \tag{6.17}$$

One can also express this probability in terms of the sigmoid function:

$$P(v_i = 1|\overline{h}) = \text{Sigmoid}\left(b_i^{(v)} + \sum_{j=1}^{m} h_j w_{ij}\right) \tag{6.18}$$

One nice consequence of using the sigmoid is that it is often possible to create a closely related feed-forward network with sigmoid activation units in which the weights learned by the Boltzmann machine are leveraged in a directed computation with input-output mappings. The weights of this network are then fine-tuned with backpropagation. We will give examples of this approach in the application section.

Note that the weights encode the affinities between the visible and hidden states. A large positive weight implies that the two states are likely to be on together. For example, in Figure 6.3(b), it might be possible that the parents are more likely to buy cones and sundae from Ben's truck, whereas they are more likely to buy popsicles and cups from Tom's truck. These propensities are encoded in the weights, which regulate both visible state selection and hidden state selection in a circular way. The *circular* nature of the relationship creates challenges, because the relationship between ice-cream choice and truck choice runs both ways; it is the raison d'etre for Gibb's sampling. Although Alice might not know which trucks the ice-creams are coming from, she will notice the resulting correlations among the bits in the training data. In fact, if the weights of the RBM are known by Alice, she can use Gibb's sampling to generate 4-bit points representing "typical" examples of

ice-creams she will receive on future days. Even the weights of the model can be learned by Alice from examples, which is the essence of an unsupervised generative model. Given the fact that there are 3 hidden states (trucks) and enough examples of 4-dimensional training data points, Alice can learn the relevant weights and biases between the visible ice-creams and hidden trucks. An algorithm for doing this is discussed in the next section.

6.4.1 Training the RBM

Computation of the weights of the RBM is achieved using a similar type of learning rule as that used for Boltzmann machines. In particular, it is possible to create an efficient algorithm based on mini-batches. The weights w_{ij} are initialized to small values. For the current set of weights w_{ij}, they are updated as follows:

- *Positive phase:* The algorithm uses a mini-batch of training instances, and computes the probability of the state of each hidden unit in exactly one step using Equation 6.15. Then a single sample of the state of each hidden unit is generated from this probability. This process is repeated for each element in a mini-batch of training instances. The correlation between these different training instances of v_i and generated instances of h_j is computed; it is denoted by $\langle v_i, h_j \rangle_{pos}$. This correlation is essentially the average product between each such pair of visible and hidden units.

- *Negative phase:* In the negative phase, the algorithm starts with a mini-batch of training instances. Then, for each training instance, it goes through a phase of Gibbs sampling after starting with randomly initialized states. This is achieved by repeatedly using Equations 6.15 and 6.17 to compute the probabilities of the visible and hidden units, and using these probabilities to draw samples. The values of v_i and h_j at thermal equilibrium are used to compute $\langle v_i, h_j \rangle_{neg}$ in the same way as the positive phase.

- One can then use the same type of update as is used in Boltzmann machines:

$$w_{ij} \Leftarrow w_{ij} + \alpha \left(\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg} \right)$$
$$b_i^{(v)} \Leftarrow b_i^{(v)} + \alpha \left(\langle v_i, 1 \rangle_{pos} - \langle v_i, 1 \rangle_{neg} \right)$$
$$b_j^{(h)} \Leftarrow b_j^{(h)} + \alpha \left(\langle 1, h_j \rangle_{pos} - \langle 1, h_j \rangle_{neg} \right)$$

  Here, $\alpha > 0$ denotes the learning rate. Each $\langle v_i, h_j \rangle$ is estimated by averaging the product of v_i and h_j over the mini-batch, although the values of v_i and h_j are computed in different ways in the positive and negative phases, respectively. Furthermore, $\langle v_i, 1 \rangle$ represents the average value of v_i in the mini-batch, and $\langle 1, h_j \rangle$ represents the average value of h_j in the mini-batch.

It is helpful to interpret the updates above in terms of Alice's trucks in Figure 6.3(b). When the weights of certain visible bits (e.g., cones and sundae) are highly correlated, the above updates will tend to push the weights in directions that these correlations can be explained by the weights between the trucks and the ice-creams. For example, if the cones and sundae are highly correlated but all other correlations are very weak, it can be explained by high weights between each of these two types of ice-creams and a single truck. In practice, the correlations will be far more complex, as will the patterns of the underlying weights.

An issue with the above approach is that one would need to run the Monte Carlo sampling for a while in order to obtain thermal equilibrium and generate the negative

samples. However, it turns out that it is possible to run the Monte Carlo sampling for only a short time *starting by fixing the visible states to a training data point from the mini-batch* and still obtain a good approximation of the gradient.

6.4.2 Contrastive Divergence Algorithm

The fastest variant of the contrastive divergence approach uses a *single* additional iteration of Monte Carlo sampling (over what is done in the positive phase) in order to generate the samples of the hidden and visible states. First, the hidden states are generated by fixing the visible units to a training point (which is already accomplished in the positive phase), and then the visible units are generated again (exactly once) from these hidden states using Monte Carlo sampling. The values of the visible units are used as the sampled states in lieu of the ones obtained at thermal equilibrium. The hidden units are generated again using these visible units. Thus, the main difference between the positive and negative phase is only of the number of iterations that one runs the approach starting with the same initialization of visible states to training points. In the positive phase, we use only half an iteration of simply computing the hidden states. In the negative phase, we use at least one *additional* iteration (so that visible states are recomputed from hidden states and hidden states generated again). This difference in the number of iterations is what causes the contrastive divergence between the state distributions in the two cases. The intuition is that an increased number of iterations causes the distribution to move away (i.e., diverge) from the data-conditioned states to what is proposed by the current weight vector. Therefore, the value of $(\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg})$ in the update quantifies the amount of contrastive divergence. This fastest variant of the contrastive divergence algorithm is referred to as CD_1 because it uses a single (additional) iteration in order to generate the negative samples. Of course, using such an approach is only an approximation to the true gradient. One can improve the accuracy of contrastive divergence by increasing the number of additional iterations to k, in which the data is reconstructed k times. This approach is referred to as CD_k. Increased values of k lead to better gradients at the expense of speed.

In the early iterations, using CD_1 is good enough, although it might not be helpful in later phases. Therefore, a natural approach is to progressively increase the value of k, while applying CD_k in training. One can summarize this process as follows:

1. In the early phase of gradient-descent, the weights are initialized to small values. In each iteration, only one additional step of contrastive divergence is used. One step is sufficient at this point because the difference between the weights are very inexact in early iterations and only a rough direction of descent is needed. Therefore, even if CD_1 is executed, one will be able to obtain a good direction in most cases.

2. As the gradient descent nears a better solution, higher accuracy is needed. Therefore, two or three steps of contrastive divergence are used (i.e., CD_2 or CD_3). In general, one can double the number of Markov chain steps after a fixed number of gradient descent steps. Another approach advocated in [469] is to create the value of k in CD_k by 1 after every 10,000 steps. The maximum value of k used in [469] was 20.

The contrastive divergence algorithm can be extended to many other variations of the RBM. An excellent practical guide for training restricted Boltzmann machines may be found in [193]. This guide discusses several practical issues such as initialization, tuning, and updates. In the following, we provide a brief overview of some of these practical issues.

6.4.3 Practical Issues and Improvisations

There are several practical issues in training the RBM with contrastive divergence. Although we have always assumed that the Monte Carlo sampling procedure generates binary samples, this is not quite the case. Some of the iterations of the Monte Carlo sampling directly use *computed* probabilities (cf. Equations 6.15 and 6.17), rather than *sampled* binary values. This is done in order to reduce the noise in training, because probability values retain more information than binary samples. However, there are some differences between how hidden states and visible states are treated:

- *Improvisations in sampling hidden states:* The final iteration of CD_k computes hidden states as probability values according to Equation 6.15 for positive and negative samples. Therefore, the value of h_j used for computing $\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg}$ would always be a real value for both positive and negative samples. This real value is a fraction because of the use of the sigmoid function in Equation 6.15.

- *Improvisations in sampling visible states:* Therefore, the improvisations for Monte Carlo sampling of visible states are always associated with the computation of $\langle v_i, h_j \rangle_{neg}$ rather than $\langle v_i, h_j \rangle_{pos}$ because visible states are always fixed to the training data. For the negative samples, the Monte Carlo procedure *always* computes probability values of visible states according to Equation 6.17 over *all* iterations rather than using 0-1 values. This is not the case for the hidden states, which are always binary until the very last iteration.

Using probability values iteratively rather than sampled binary values is technically incorrect, and does not reach correct thermal equilibrium. However, the contrastive divergence algorithm is an approximation anyway, and this type of approach reduces significant noise at the expense of some theoretical incorrectness. Noise reduction is a result of the fact that the probabilistic outputs are closer to expected values.

The weights can be initialized from a Gaussian distribution with zero mean and a standard deviation of 0.01. Large values of the initial weights can speed up the learning, but might lead to a model that is slightly worse in the end. The visible biases are initialized to $\log(p_i/(1-p_i))$, where p_i is the fraction of data points in which the ith dimension takes on the value of 1. The values of the hidden biases are initialized to 0.

The size of the mini-batch should be somewhere between 10 and 100. The order of the examples should be randomized. For cases in which class labels are associated with examples, the mini-batch should be selected in such a way that the proportion of labels in the batch is approximately the same as the whole data.

6.5 Applications of Restricted Boltzmann Machines

In this section, we will study several applications of restricted Boltzmann machines. These methods have been very successful for a variety of unsupervised applications, although they are also used for supervised applications. When using an RBM in a real-world application, a mapping from input to output is often required, whereas a vanilla RBM is only designed to learn probability distributions. The input-to-output mapping is often achieved by constructing a feed-forward network with weights derived from the learned RBM. In other words, one can often derive a traditional neural network that is *associated* with the original RBM.

Here, we will like to discuss the differences between the notions of the *state* of a node in the RBM, and the *activation* of that node in the associated neural network. The state of a

node is a binary value sampled from the Bernoulli probabilities defined by Equations 6.15 and 6.17. On the other hand, the activation of a node in the associated neural network is the probability value derived from the use of the sigmoid function in Equations 6.15 and 6.17. Many applications use the activations in the nodes of the associated neural network, rather than the states in the original RBM after the training. Note that the final step in the contrastive divergence algorithm also leverages the activations of the nodes rather than the states while updating the weights. In practical settings, the activations are more information-rich and are therefore useful. The use of activations is consistent with traditional neural network architectures, in which backpropagation can be used. The use of a final phase of backpropagation is crucial in being able to apply the approach to supervised applications. In most cases, the critical role of the RBM is to perform unsupervised feature learning. Therefore, the role of the RBM is often only one of pretraining in the case of supervised learning. In fact, pretraining is one of the important historical contributions of the RBM.

6.5.1 Dimensionality Reduction and Data Reconstruction

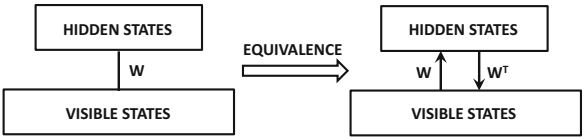
The most basic function of the RBM is that of dimensionality reduction and unsupervised feature engineering. The hidden units of an RBM contain a reduced representation of the data. However, we have not yet discussed how one can reconstruct the original representation of the data with the use of an RBM (much like an autoencoder). In order to understand the reconstruction process, we first need to understand the equivalence of the undirected RBM with directed graphical models [251], in which the computation occurs in a particular direction. Materializing a directed probabilistic graph is the first step towards materializing a traditional neural network (derived from the RBM) in which the discrete probabilistic sampling from the sigmoid can be replaced with real-valued sigmoid activations.

Although an RBM is an undirected graphical model, one can "unfold" an RBM in order to create a directed model in which the inference occurs in a particular direction. In general, an undirected RBM can be shown to be equivalent to a directed graphical model with an infinite number of layers. The unfolding is particularly useful when the visible units are fixed to specific values because the number of layers in the unfolding collapses to exactly twice the number of layers in the original RBM. Furthermore, by replacing the discrete probabilistic sampling with continuous sigmoid units, this directed model functions as a virtual autoencoder, which has both an encoder portion and a decoder portion. Although the weights of an RBM have been trained using discrete probabilistic sampling, they can also be used in this related neural network with some fine tuning. This is a heuristic approach to convert what has been learned from a Boltzmann machine (i.e., the weights) into the initialized weights of a traditional neural network with sigmoid units.

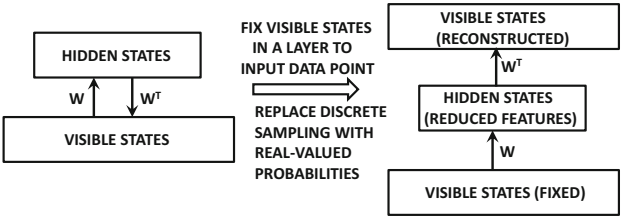
An RBM can be viewed as an undirected graphical model that uses the same weight matrix to learn $\overline{h}$ from $\overline{v}$ as it does from $\overline{v}$ to $\overline{h}$. If one carefully examines Equations 6.15 and 6.17, one can see that they are very similar. The main difference is that these equations uses different biases, and they use the transposes of each other's weight matrices. In other words, one can rewrite Equations 6.15 and 6.17 in the following form for some function $f(\cdot)$:

$$\overline{h} \sim f(\overline{v}, \overline{b}^{(h)}, W)$$
$$\overline{v} \sim f(\overline{h}, \overline{b}^{(v)}, W^T)$$

The function $f(\cdot)$ is typically defined by the sigmoid function in binary RBMs, which constitute the predominant variant of this class of models. Ignoring the biases, one can replace



(a) Equivalence of directed and undirected relationships



(b) Discrete graphical model to approximate real-valued neural network

Figure 6.4: Using trained RBM to approximate trained autoencoder

the undirected graph of the RBM with two directed links, as shown in Figure 6.4(a). Note that the weight matrices in the two directions are W and W^T, respectively. However, if we fix the visible states to the training points, we can perform just two iterations of these operations to reconstruct the visible states with *real-valued* approximations. In other words, we approximate this trained RBM with a traditional neural network by replacing discrete sampling with continuous-valued sigmoid activations (as a heuristic). This conversion is shown in Figure 6.4(b). In other words, instead of using the sampling operation of "$\sim$," we replace the samples with the probability values:

$$\overline{h} = f(\overline{v}, \overline{b}^{(h)}, W)$$
$$\overline{v}' = f(\overline{h}, \overline{b}^{(v)}, W^T)$$

Note that $\overline{v}'$ is the reconstructed version of $\overline{v}$ and it will contain real values (unlike the binary states in $\overline{v}$). In this case, we are working with real-valued activations rather than discrete samples. Because sampling is no longer used and all computations are performed in terms of expectations, we need to perform only one iteration of Equation 6.15 in order to learn the reduced representation. Furthermore, only one iteration of Equation 6.17 is required to learn the reconstructed data. The prediction phase works only in a single direction from the input point to the reconstructed data, and is shown on the right-hand side of Figure 6.4(b). We modify Equations 6.15 and 6.17 to define the states of this traditional neural network as real values:

$$\hat{h}_j = \frac{1}{1 + \exp(-b_j^{(h)} - \sum_{i=1}^{d} v_i w_{ij})} \tag{6.19}$$

For a setting with a total of $m \ll d$ hidden states, the real-valued reduced representation is given by $(\hat{h}_1 \ldots \hat{h}_m)$. This first step of creating the hidden states is equivalent to the encoder portion of an autoencoder, and these values are the expected values of the binary states. One can then apply Equation 6.17 to these *probabilistic values* (without creating Monte-Carlo instantiations) in order to reconstruct the visible states as follows:

$$\hat{v}_i = \frac{1}{1 + \exp(-b_i^{(v)} - \sum_j \hat{h}_j w_{ij})} \tag{6.20}$$

Although $\hat{h}_j$ does represent the expected value of the jth hidden unit, applying the sigmoid function again to this real-valued version of $\hat{h}_j$ only provides a rough approximation to the expected value of v_i. Nevertheless, the real-valued prediction $\hat{v}_i$ is an approximate reconstruction of v_i. Note that in order to perform this reconstruction we have used similar operations as traditional neural networks with sigmoid units rather than the troublesome discrete samples of probabilistic graphical models. Therefore, we can now use this related neural network as a good starting point for fine-tuning the weights with traditional backpropagation. This type of reconstruction is similar to the reconstruction used in the autoencoder architecture discussed in Chapter 2.

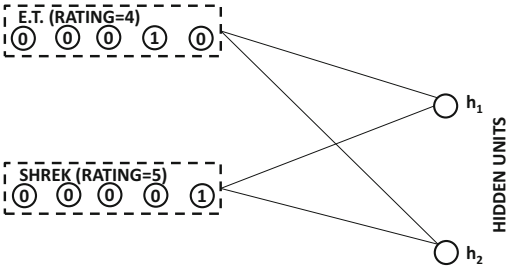
On first impression, it makes little sense to train an RBM when similar goals can be achieved with a traditional autoencoder. However, this broad approach of deriving a traditional neural network with a trained RBM is particularly useful when working with stacked RBMs (cf. Section 6.7). The training of a stacked RBM does not face the same challenges as those associated with deep neural networks, especially the ones related with the vanishing and exploding gradient problems. Just as the simple RBM provides an excellent initialization point for the shallow autoencoder, the stacked RBM also provides an excellent starting point for a deep autoencoder [198]. This principle led to the development of the idea of pretraining with RBMs before conventional pretraining methods were developed without the use of RBMs. As discussed in this section, one can also use RBMs for other reduction-centric applications such as collaborative filtering and topic modeling.

6.5.2 RBMs for Collaborative Filtering

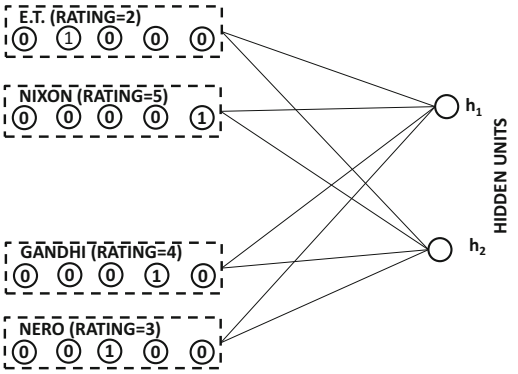
The previous section shows how restricted Boltzmann machines are used as alternatives to the autoencoder for unsupervised modeling and dimensionality reduction. However, as discussed in Section 2.5.7 of Chapter 2, dimensionality reduction methods are also used for a variety of related applications like collaborative filtering. In the following, we will provide an RBM-centric alternative to the recommendation technique described in Section 2.5.7 of Chapter 2. This approach is based on the technique proposed in [414], and it was one of the ensemble components of the winning entry in the Netflix prize contest.

One of the challenges in working with ratings matrices is that they are incompletely specified. This tends to make the design of a neural architecture for collaborative filtering more difficult than traditional dimensionality reduction. Recall from the discussion in Section 2.5.7 that modeling such incomplete matrices with a traditional neural network also faces the same challenge. In that section, it was shown how one could create a different training instance *and* a different neural network for each user, depending on which ratings are observed by that user. All these different neural networks share weights. An exactly similar approach is used with the restricted Boltzmann machine, in which one training case and one RBM is defined for each user. However, in the case of the RBM, one additional problem is that the units are binary, whereas ratings can take on values from 1 to 5. Therefore, we need some way of working with the additional constraint.

In order to address this issue, the hidden units in the RBM are allowed to be 5-way softmax units in order to correspond to rating values from 1 to 5. In other words, the hidden units are defined in the form of a one-hot encoding of the rating. One-hot encodings are naturally modeled with softmax, which defines the probabilities of each possible position. The ith softmax unit corresponds to the ith movie and the probability of a particular rating being given to that movie is defined by the distribution of softmax probabilities. Therefore, if there are d movies, we have a total of d such one-hot encoded ratings. The values of the corresponding binary values of the one-hot encoded visible units are denoted by $v_i^{(1)}, \ldots v_i^{(5)}$.



(a) RBM architecture for user Sayani (Observed Ratings: *E.T.* and *Shrek*)



(b) RBM architecture for user Bob (Observed Ratings: *E.T.*, *Nixon*, *Gandhi*, and *Nero*)

Figure 6.5: The RBM architectures of two users are shown based on their observed ratings. It is instructive to compare this figure with the conventional neural architecture shown in Figure 2.14 in Chapter 2. In both cases, weights are shared by user-specific networks.

Note that only one of the values of $v_i^{(k)}$ can be 1 over fixed i and varying k. The hidden layer is assumed to contain m units. The weight matrix has a separate parameter for each of the multinomial outcomes of the softmax unit. Therefore, the weight between visible unit i and hidden unit j for the outcome k is denoted by $w_{ij}^{(k)}$. In addition, we have 5 biases for the visible unit i, which are denoted by $b_i^{(k)}$ for $k \in \{1, \ldots, 5\}$. The hidden units only have a single bias, and the bias of the jth hidden unit is denoted by b_j (without a superscript). The architecture of the RBM for collaborative filtering is illustrated in Figure 6.5. This example contains $d = 5$ movies and $m = 2$ hidden units. In this case, the RBM architectures of two users, Sayani and Bob, are shown in the figure. In the case of Sayani, she has specified ratings for only two movies. Therefore, a total of $2 \times 2 \times 5 = 20$ connections will be present in her case, even though we have shown only a subset of them to avoid clutter in the figure. In the case of Bob, he has four observed ratings, and therefore his network will contain a total of $4 \times 2 \times 5 = 40$ connections. Note that both Sayani and Bob have rated the movie *E.T.*, and therefore the connections from this movie to the hidden units will share weights between the corresponding RBMs.

The states of the hidden units, which are binary, are defined with the use of the sigmoid function:

$$P(h_j = 1|\overline{v}^{(1)} \ldots \overline{v}^{(5)}) = \frac{1}{1 + \exp(-b_j - \sum_{i,k} v_i^{(k)} w_{ij}^k)} \tag{6.21}$$

The main difference from Equation 6.15 is that the visible units also contain a superscript to correspond to the different rating outcomes. Otherwise, the condition is virtually identical. However, the probabilities of the visible units are defined differently from the traditional RBM model. In this case, the visible units are defined using the softmax function:

$$P(v_i^{(k)} = 1|\overline{h}) = \frac{\exp(b_i^{(k)} + \sum_j h_j w_{ij}^{(k)})}{\sum_{r=1}^{5} \exp(b_i^{(r)} + \sum_j h_j w_{ij}^{(r)})} \tag{6.22}$$

The training is done in a similar way as the unrestricted Boltzmann machine with Monte Carlo sampling. The main difference is that the visible states are generated from a multinomial model. Therefore, the MCMC sampling should also generate the negative samples from the multinomial model of Equation 6.22 to create each $v_i^{(k)}$. The corresponding updates for training the weights are as follows:

$$w_{ij}^{(k)} \Leftarrow w_{ij}^{(k)} + \alpha \left(\langle v_i^{(k)}, h_j \rangle_{pos} - \langle v_i^{(k)}, h_j \rangle_{neg} \right) \quad \forall k \tag{6.23}$$

Note that only the weights of the *observed* visible units to all hidden units are updated for a single training example (i.e., user). In other words, the Boltzmann machine that is used is different for each user in the data, although the weights are shared across the different users. Examples of the Boltzmann machines for two different training examples are illustrated in Figure 6.5, and the architectures for Bob and Sayani are different. However, the weights for the units representing *E.T.* are shared. This type of approach is also used in the traditional neural architecture of Section 2.5.7 in which the neural network used for each training example is different. As discussed in that section, the traditional neural architecture is equivalent to a matrix factorization technique. The Boltzmann machine tends to give somewhat different ratings predictions from matrix factorization techniques, although the accuracy is similar.

Making Predictions

Once the weights have been learned, they can be used for making predictions. However, the predictive phase works with real-valued activations rather than binary states, much like a traditional neural network with sigmoid and softmax units. First, one can use Equation 6.21 in order to learn the probabilities of the hidden units. Let the probability that the jth hidden unit is 1 be denoted by $\hat{p}_j$. Then, the probabilities of *unobserved* visible units are computed using Equation 6.22. The main problem in computing Equation 6.22 is that it is defined in terms of the values of the hidden units, which are only known in the form of probabilities according to Equation 6.21. However, one can simply replace each h_j with $\hat{p}_j$ in Equation 6.22 in order to compute the probabilities of the visible units. Note that these predictions provide the probabilities of each possible rating value of each item. These probabilities can also be used to compute the expected value of the rating if needed. Although this approach is approximate from a theoretical point of view, it works well in practice and is extremely fast. By using these real-valued computations, one is effectively converting the RBM into a traditional neural network architecture with logistic units for hidden layers and

softmax units for the input and output layers. Although the original paper [414] does not mention it, it is even possible to tune the weights of this network with backpropagation (cf. Exercise 1).

The RBM approach works as well as the traditional matrix factorization approach, although it tends to give different types of predictions. This type of diversity is an advantage from the perspective of using an ensemble-centric approach. Therefore, the results can be combined with the matrix factorization approach in order to yield the improvements that are naturally associated with an ensemble method. Ensemble methods generally show better improvements when diverse methods of similar accuracy are combined.

Conditional Factoring: A Neat Regularization Trick

A neat regularization trick is buried inside the RBM-based collaborative filtering work of [414]. This trick is not specific to the collaborative filtering application, but can be used in any application of an RBM. This approach is not necessary in traditional neural networks, where it can be simulated by incorporating an additional hidden layer, but it is particularly useful for RBMs. Here, we describe this trick in a more general way, without its specific modifications for the collaborative filtering application. In some applications with a large number of hidden units and visible units, the size of the parameter matrix $W = [w_{ij}]$ might be large. For example, in a matrix with $d = 10^5$ visible units, and $m = 100$ hidden units, we will have ten million parameters. Therefore, more than ten million training points will be required to avoid overfitting. A natural approach is to assume a low-rank parameter structure of the weight matrix, which is a form of regularization. The idea is to assume that the matrix W can be expressed as the product of two low-rank factors U and V, which are of sizes $d \times k$ and $m \times k$, respectively. Therefore, we have the following:

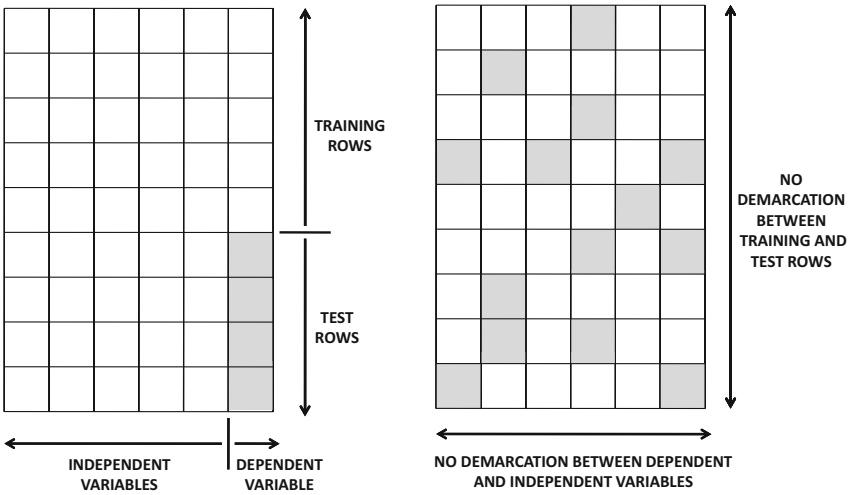
$$W = UV^T \tag{6.24}$$

Here, k is the rank of the factorization, which is typically much less than both d and m. Then, instead of learning the parameters of the matrix W, one can learn the parameters of U and V, respectively. This type of trick is used often in various machine learning applications, where parameters are represented as a matrix. A specific example is that of factorization machines, which are also used for collaborative filtering [396]. This type of approach is not required in traditional neural networks, because one can simulate it by incorporating an additional linear layer with k units between two layers with a weight matrix of W between them. The weight matrices of the two layers will be U and V^T, respectively.

6.5.3 Using RBMs for Classification

The most common way to use RBMs for classification is as a pretraining procedure. In other words, a Boltzmann machine is first used to perform unsupervised feature engineering. The RBM is then unrolled into a related encoder-decoder architecture according to the approach described in Section 6.5.1. This is a traditional neural network with sigmoid units, whose weights are derived from the unsupervised RBM rather than backpropagation. The encoder portion of this neural network is topped with an output layer for class prediction. The weights of this neural network are then fine-tuned with backpropagation. Such an approach can even be used with *stacked RBMs* (cf. Section 6.7) to yield a deep classifier. This methodology of initializing a (conventional) deep neural network with an RBM was one of the first approaches for pretraining deep networks.

There is, however, another alternative approach to perform classification with the RBM, which integrates RBM training and inference more tightly with the classification process. This approach is somewhat similar to the collaborative filtering methodology discussed in the previous section. The collaborative-filtering problem is also referred to as *matrix completion* because the missing entries of an incompletely specified matrix are predicted. The use of RBMs for recommender systems provides some useful hints about their use in classification. This is because classification can be viewed as a simplified version of the matrix completion problem in which we create a single matrix out of both the training and test rows, and the missing values belong to a particular column of the matrix. This column corresponds to the class variable. Furthermore, all the missing values are present in the test rows in the case of classification, whereas the missing values could be present anywhere in the matrix in the case of recommender systems. This relationship between classification and the generic matrix completion problem is illustrated in Figure 6.6. In classification, all features are observed for the rows corresponding to training points, which simplifies the modeling (compared to collaborative filtering in which a complete set of features is typically not observed for any row).



(a) Classification (b) Matrix completion

Figure 6.6: The classification problem is a special case of matrix completion. Shaded entries are missing and need to be predicted.

We assume that the input data contains d binary features. The class label has k discrete values, which corresponds to the multiway classification problem. The classification problem can be modeled by the RBM by defining the hidden and visible features as follows:

1. The visible layer contains two types of nodes corresponding to the features and the class label, respectively. There are d binary units corresponding to features, and there are k binary units corresponding to the class label. However, only one of these k binary units can take on the value of 1, which corresponds to a one-hot encoding of the class labels. This encoding of the class label is similar to the approach used for encoding the ratings in the collaborative-filtering application. The visible units for the features are denoted by $v_1^{(f)} \ldots v_d^{(f)}$, whereas the visible units for the class labels are denoted by $v_1^{(c)} \ldots v_k^{(c)}$. Note that the symbolic superscripts denote whether the visible units corresponds to a feature or a class label.

2. The hidden layer contains m binary units. The hidden units are denoted by $h_1 \ldots h_m$.

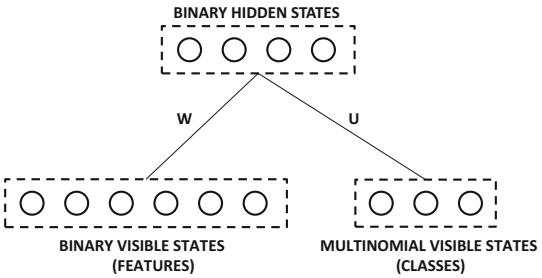


Figure 6.7: The RBM architecture for classification

The weight of the connection between the ith feature-specific visible unit $v_i^{(f)}$ and the jth hidden unit h_j is given by w_{ij}. This results in a $d \times m$ connection matrix $W = [w_{ij}]$. The weight of the connection between the ith class-specific visible unit $v_i^{(c)}$ and the jth hidden unit h_j is given by u_{ij}. This results in a $k \times m$ connection matrix $U = [u_{ij}]$. The relationships between different types of nodes and matrices for $d = 6$ features, $k = 3$ classes, and $m = 5$ hidden features is shown in Figure 6.7. The bias for the ith feature-specific visible node is denoted by $b_i^{(f)}$, and the bias for the ith class-specific visible node is denoted by $b_i^{(c)}$. The bias for the jth hidden node is denoted by b_j (with no superscript). The states of the hidden nodes are defined in terms of all visible nodes using the sigmoid function:

$$P(h_j = 1|\overline{v}^{(f)}, \overline{v}^{(c)}) = \frac{1}{1 + \exp(-b_j - \sum_{i=1}^{d} v_i^{(f)} w_{ij} - \sum_{i=1}^{k} v_i^{(c)} u_{ij})} \tag{6.25}$$

Note that this is the standard way in which the probabilities of hidden units are defined in a Boltzmann machine. There are, however, some differences between how the probabilities of the feature-specific visible units and the class-specific visible units are defined. In the case of the feature-specific visible units, the relationship is not very different from a standard Boltzmann machine:

$$P(v_i^{(f)} = 1|\overline{h}) = \frac{1}{1 + \exp(-b_i^{(f)} - \sum_{j=1}^{m} h_j w_{ij})} \tag{6.26}$$

The case of the class units is, however, slightly different because we must use the softmax function instead of the sigmoid. This is because of the one-hot encoding of the class. Therefore, we have the following:

$$P(v_i^{(c)} = 1|\overline{h}) = \frac{\exp(b_i^{(c)} + \sum_j h_j u_{ij})}{\sum_{l=1}^{k} \exp(b_l^{(c)} + \sum_j h_j u_{lj})} \tag{6.27}$$

A naive approach to training the Boltzmann machine would use a similar generative model to previous sections. The multinomial model is used to generate the visible states $v_i^{(c)}$ for the classes. The corresponding updates of the contrastive divergence algorithm are as follows:

$$w_{ij} \Leftarrow w_{ij} + \alpha \left(\langle v_i^{(f)}, h_j \rangle_{pos} - \langle v_i^{(f)}, h_j \rangle_{neg} \right) \quad \text{if } i \text{ is feature unit}$$

$$u_{ij} \Leftarrow u_{ij} + \alpha \left(\langle v_i^{(c)}, h_j \rangle_{pos} - \langle v_i^{(c)}, h_j \rangle_{neg} \right) \quad \text{if } i \text{ is class unit}$$

This approach is a direct extension from collaborative filtering. However, the main problem is that this *generative* approach does not fully optimize for classification accuracy. To provide an analogy with autoencoders, one would not necessarily perform significantly better dimensionality reduction (in a supervised sense) by simply including the class variable among the inputs. The reduction would often be dominated by the unsupervised relationships among the features. Rather, the *entire focus* of the learning should be on optimizing the accuracy of classification. Therefore, a *discriminative* approach to training the RBM is often used in which the weights are learned to maximize the conditional class likelihood of the true class. Note that it is easy to set up the conditional probability of the class variable, given the visible states by using the probabilistic dependencies between the hidden features and classes/features. For example, in the traditional form of a restrictive Boltzmann machine, we are maximizing the *joint* probability of the feature variables $v_i^{(f)}$ and the class variables v_i^c. However, in the discriminative variant, the objective function is set up to maximize the *conditional* probability of the class variable $y \in \{1 \ldots k\}$ $P(v_y^{(c)} = 1|\overline{v}^{(f)})$. Such an approach has a more focused effect of maximizing classification accuracy. Although it is possible to train a discriminative restricted Boltzmann machine using contrastive divergence, the problem is simplified because one can estimate $P(v_y^{(c)} = 1|\overline{v}^{(f)})$ in closed form without having to use an iterative approach. This form can be shown to be the following [263, 414]:

$$P(v_y^{(c)} = 1|\overline{v}^{(f)}) = \frac{\exp(b_y^{(c)}) \prod_{j=1}^{m} [1 + \exp(b_j^{(h)} + u_{yj} + \sum_i w_{ij} v_i^{(f)})]}{\sum_{l=1}^{k} \exp(b_l^{(c)}) \prod_{j=1}^{m} [1 + \exp(b_j^{(h)} + u_{lj} + \sum_i w_{ij} v_i^{(f)})]} \tag{6.28}$$

With this differentiable closed form, it is a simple matter to differentiate the negative logarithm of the above expression for stochastic gradient descent. If $\mathcal{L}$ is the negative logarithm of the above expression and θ is any particular parameter (e.g., weight or bias) of the Boltzmann machine, one can show the following:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{j=1}^{m} \text{Sigmoid}(o_{yj}) \frac{\partial o_{yj}}{\partial \theta} - \sum_{l=1}^{k} \sum_{j=1}^{m} \text{Sigmoid}(o_{lj}) \frac{\partial o_{lj}}{\partial \theta} \tag{6.29}$$

Here, we have $o_{yj} = b_j^{(h)} + u_{yj} + \sum_i w_{ij} v_i^{(f)}$. The above expression can be easily computed for each training point and for each parameter in order to perform the stochastic gradient descent process. It is a relatively simple matter to make probabilistic predictions for unseen test instances using Equation 6.28. More details and extensions are discussed in [263].

6.5.4 Topic Models with RBMs

Topic modeling is a form of dimensionality reduction that is specific to text data. The earliest topic models, which correspond to Probabilistic Latent Semantic Analysis (PLSA), were proposed in [206]. In PLSA, the basis vectors are not orthogonal to one another, as is the case with SVD. On the other hand, both the basis vectors and the transformed representations are constrained to be nonnegative values. The nonnegativity in the value of each transformed feature is semantically useful, because it represents the strength of a topic in a particular document. In the context of the RBM, this strength corresponds to the probability that a particular hidden unit takes on the value of 1, given that the words in a particular document have been observed. Therefore, one can use the vector of conditional probabilities of the hidden states (when visible states are fixed to document words) in order to create a reduced representation of each document. It is assumed that the lexicon size is d, whereas the number of hidden units is $m \ll d$.

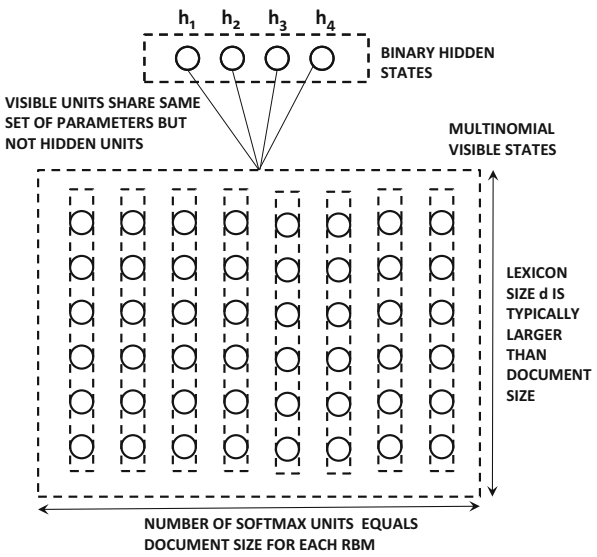


Figure 6.8: The RBM for each document is illustrated. The number of visible units is equal to the number of words in each document

This approach shares some similarities with the technique used for collaborative filtering in which a single RBM is created for each user (row of the matrix). In this case, a single RBM is created for each document. A group of visible units is created for each word, and therefore the number of groups of visible units is equal to the number of words in the document. In the following, we will concretely define how the visible and hidden states of the RBM are fixed in order to describe the concrete workings of the model:

1. For the tth document containing n_t words, a total of n_t softmax groups are retained. Each softmax group contains d nodes corresponding to the d words in the lexicon. Therefore, the RBM for each document is different, because the number of units depends on the length of the document. However, all the softmax groups within a document and across multiple documents share weights of their connections to the hidden units. The ith position in the document corresponds to the ith group of visible softmax units. The ith group of visible units is denoted by $v_i^{(1)} \ldots v_i^{(d)}$. The bias associated with $v_i^{(k)}$ is $b^{(k)}$. Note that the bias of the ith visible node depends only on k (word identity) and not on i (position of word in document). This is because the model uses a bag-of-words approach in which the positions of the words are irrelevant.

2. There are m hidden units denoted by $h_1 \ldots h_m$. The bias of the jth hidden unit is b_j.

3. Each hidden unit is connected to each of the $n_t \times d$ visible units. All softmax groups within a single RBM as well as across different RBMs (corresponding to different documents) share the same set of d weights. The kth hidden unit is connected to a group of d softmax units with a vector of d weights denoted by $\overline{W}^{(k)} = (w_1^{(k)} \ldots w_d^{(k)})$. In other words, the kth hidden unit *is connected to each of the n_t groups of d softmax units with the same set of weights $\overline{W}^{(k)}$.*

The architecture of the RBM is illustrated in Figure 6.8. Based on the architecture of the RBM, one can express the probabilities associated with the states of the hidden units with the use of the sigmoid function:

$$P(h_j = 1|\overline{v}^{(1)}, \ldots \overline{v}^{(d)}) = \frac{1}{1 + \exp(-b_j - \sum_{i=1}^{n_t} \sum_{k=1}^{d} v_i^{(k)} w_j^{(k)})} \tag{6.30}$$

One can also express the visible states with the use of the multinomial model:

$$P(v_i^{(k)} = 1|\overline{h}) = \frac{\exp(b^{(k)} + \sum_{j=1}^{m} w_j^{(k)} h_j)}{\sum_{l=1}^{d} \exp(b^{(l)} + \sum_{j=1}^{m} w_j^{(l)} h_j)} \tag{6.31}$$

The normalization factor in the denominator ensures that the sum of the probabilities of visible units over all the words always sums to 1. Furthermore, the right-hand side of the above equation is independent of the index i of the visible unit. This is because this model does not depend on the position of words in the document, and the modeling treats a document as a bag of words.

With these relationships, one can apply MCMC sampling to generate samples of the hidden and visible states for the contrastive divergence algorithm. Note that the RBMs are different for different documents, although these RBMs share weights. As in the case of the collaborative filtering application, each RBM is associated with only a single training example corresponding to the relevant document. The weight update used for gradient descent is the same as used for the traditional RBM. The only difference is that the weights across different visible units are shared. This approach is similar to what is performed in collaborative filtering. We leave the derivation of the weight updates as an exercise for the reader (see Exercise 5).

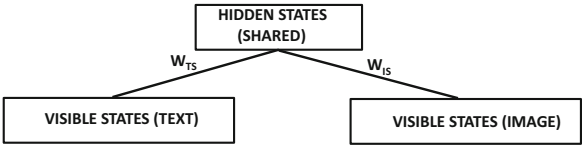
After the training has been performed, the reduced representation of each document is computed by applying Equation 6.30 to the words of a document. The real-valued value of the probabilities of the hidden units provides the m-dimensional reduced representation of the document. The approach described in this section is a simplification of a multilayer approach described in the original work [469].

6.5.5 RBMs for Machine Learning with Multimodal Data

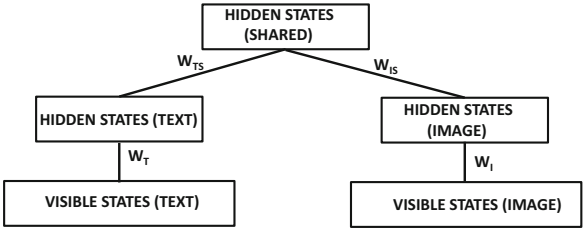
Boltzmann machines can also be used for machine learning with *multimodal* data. Multimodal data refers to a setting in which one is trying to extract information from data points with multiple modalities. For example, an image with a text description can be considered multimodal data. This is because this data object has both image and text modalities.

The main challenge in processing multimodal data is that it is often difficult to use machine learning algorithms on such heterogeneous features. Multimodal data is often processed by using a shared representation in which the two modes are mapped into a joint space. A common approach for this goal is *shared* matrix factorization. Numerous methods for using shared matrix factorization with text and image data are discussed in [6]. Since RBMs provide alternative representations to matrix factorization methods in many settings, it is natural to explore whether one can use this architecture to create a shared latent representation of the data.

An example [468] of an architecture for multimodal modeling is shown in Figure 6.9(a). In this example, it is assumed that the two modes correspond to text and image data. The image and the text data are used to create hidden states that are specific to images and



(a) A simple RBM for multimodal data



(b) A multimodal RBM with an added hidden layer

Figure 6.9: RBM architecture for multimodal data processing

text, respectively. These hidden states then feed into a single shared representation. The similarity of this architecture with the classification architecture of Figure 6.7 is striking. This is because both architectures try to map two types of features into a set of shared hidden states. These hidden states can then be used for different types of inference, such as using the shared representation for classification. As shown in Section 6.7, one can even enhance such unsupervised representations with backpropagation to fine-tune the approach. Missing data modalities can also be generated using this model.

One can optionally improve the expressive power of this model by using depth. An additional hidden layer has been added between the visible states and the shared representation in Figure 6.9(b). Note that one can add multiple hidden layers in order to create a deep network. However, we have not yet described how one can actually train a multilayer RBM. This issue is discussed in Section 6.7.

An additional challenge with the use of multimodal data is that the features are often not binary. There are several solutions to this issue. In the case of text (or data modalities with small cardinalities of discrete attributes), one can use a similar approach as used in the RBM for topic modeling where the count c of a discrete attribute is used to create c instances of the one-hot encoded attribute. The issue becomes more challenging when the data contains arbitrary real values. One solution is to discretize the data, although such an approach can lose useful information about the data. Another solution is to make changes to the energy function of the Boltzmann machine. A discussion of some of these issues is provided in the next section.

6.6 Using RBMs Beyond Binary Data Types

The entire discussion so far in this chapter has focussed on the use of RBMs for binary data types. Indeed the vast majority of RBMs are designed for binary data types. For some types of data, such as categorical data or ordinal data (e.g., ratings), one can use the softmax approach described in Section 6.5.2. For example, the use of softmax units for word-count data is discussed in Section 6.5.4. One can make the softmax approach work

with an ordered attribute, when the number of discrete values of that attribute is small. However, these methods are not quite as effective for real-valued data. One possibility is to use discretization in order to convert real-valued data into discrete data, which can be handled with softmax units. Using such an approach does have the disadvantage of losing a certain amount of representational accuracy.

The approach described in Section 6.5.2 does provide some hints about how different data types can be addressed. For example, categorical or ordinal data is handled *by changing the probability distribution* of visible units to be more appropriate to the problem at hand. In general, one might need to change the distribution of not only the visible units, but also the hidden units. This is because the nature of the hidden units is dependent on the visible units.

For real-valued data, a natural solution is to use Gaussian visible units. Furthermore, the hidden units are real-valued as well, and are assumed to contain a ReLU activation function. The energy for a particular combination $(\overline{v}, \overline{h})$ of visible and hidden units is given by the following:

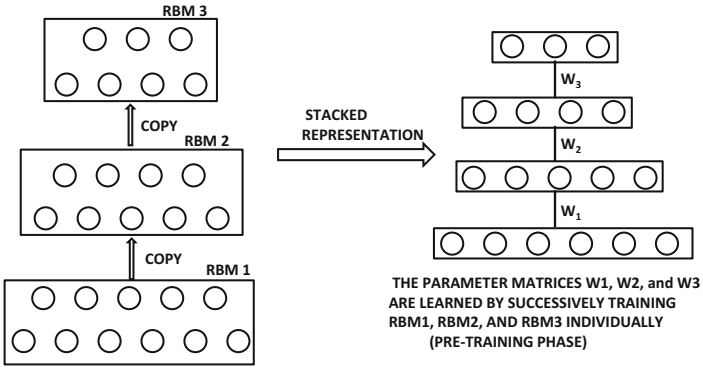
$$E(\overline{v}, \overline{h}) = \underbrace{\sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2}}_{\text{Containment function}} - \sum_j b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij} \tag{6.32}$$

Note that the energy contribution of the bias of visible units is given by a *parabolic containment function*. The effect of using this containment function is to keep the value of the ith visible unit close to b_i. As is the case for other types of Boltzmann machines, the derivatives of the energy function with respect to the different variables also provide the derivatives of the log-likelihoods. This is because the probabilities are always defined by exponentiating the energy function.

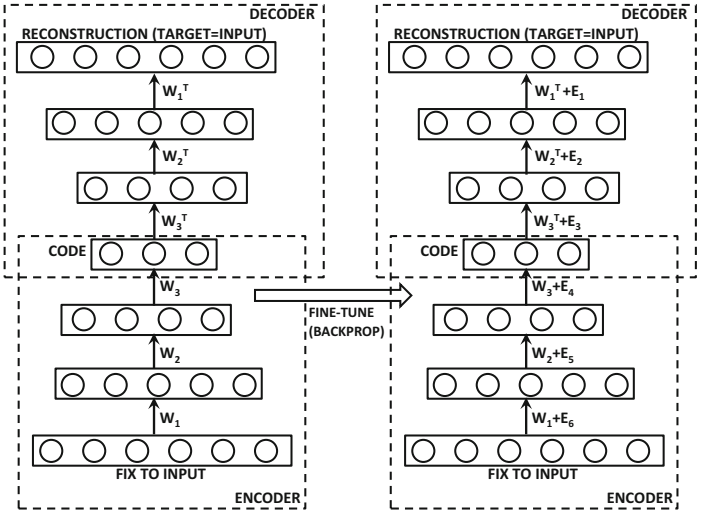
There are several challenges associated with the use of this approach. An important issue is that the approach is rather unstable with respect to the choice of the variance parameter σ. In particular, updates to the visible layer tend to be too small, whereas updates to the hidden layer tend to be too large. One natural solution to this dilemma is to use more hidden units than visible units. It is also common to normalize the input data to unit variance so that the standard deviation σ of the visible units can be set to 1. The ReLU units are modified to create a noisy version. Specifically, Gaussian noise with zero mean and variance $\log(1+\exp(v))$ is added to the value of the unit before thresholding it to nonnegative values. The motivation behind using such an unusual activation function is that it can be shown to be equivalent to a *binomial unit* [348, 495], which encodes more information than the binary unit that is normally used. It is important to enable this ability when working with real-valued data. The Gibbs sampling of the real-valued RBM is similar to a binary RBM, as are the updates to the weights once the MCMC samples are generated. It is important to keep the learning rates low to prevent instability.

6.7 Stacking Restricted Boltzmann Machines

Most of the power of conventional neural architectures arises from having multiple layers of units. Deeper networks are known to be more powerful, and can model more complex functions at the expense of fewer parameters. A natural question arises concerning whether similar goals can be achieved by putting together multiple RBMs. It turns out that the RBM is well suited to creating deep networks, and was used *earlier* than conventional neural networks for creating deep models with pretraining. In other words, the RBM is trained



(a) The stacked RBMs are trained sequentially in pretraining



(b) Pretraining is followed by fine-tuning with backpropagation

Figure 6.10: Training a multi-layer RBM

with Gibbs sampling, and the resulting weights are grandfathered into a conventional neural network with continuous sigmoid activations (instead of sigmoid-based discrete sampling). Why should one go through the trouble to train an RBM in order to train a conventional network at all? This is because of the fact that Boltzmann machines are trained in a fundamentally different way from the backpropagation approach in conventional neural networks. The contrastive divergence approach tends to train all layers jointly, which does not cause the same problems with the vanishing and exploding gradient problems, as is the case in conventional neural networks.

At first glance, the goal of creating deep networks from RBMs seems rather difficult. First, RBMs are not quite like feed-forward units that perform the computation in a particular direction. RBMs are symmetric models in which the visible units and hidden units are connected in the form of an undirected graphical model. Therefore, one needs to define a concrete way in which multiple RBMs interact with one another. In this context, a useful observation is that even though RBMs are symmetric and discrete models, the learned

weights can be used to define a related neural network that performs directed computation in the continuous space of activations. These weights are already quite close to the final solution because of how they have been learned with discrete sampling. Therefore, these weights can be fine-tuned with a relatively modest effort of traditional backpropagation. In order to understand this point, consider the single-layer RBM illustrated in Figure 6.4, which shows that even the single-layer RBM is equivalent to a directed graphical model of infinite length. However, once the visible states have been fixed, it suffices to keep only three layers of this computational graph, and perform the computations with the continuous values derived from sigmoid activations. This approach already provides a good approximate solution. The resulting network is a traditional autoencoder, although its weights have been (approximately) learned in a rather unconventional way. This section will show how this type of approach can also be applied to stacked RBMs.

What is a stacked set of RBMs? Consider a data set with d dimensions, for which the goal is to create a reduced representation with m_1 dimensions. One can achieve this goal with an RBM containing d visible units and m_1 hidden units. By training this RBM, one will obtain an m_1-dimensional representation of the data set. Now consider a second RBM that has m_1 visible units and m_2 hidden units. We can simply *copy* the m_1 outputs of the first RBM as the inputs to the second RBM, which has $m_1 \times m_2$ weights. As a result, one can train this new RBM to create an m_2-dimensional representation by using the outputs from the first RBM as its inputs. Note that we can repeat this process for k times, so that the last RBM is of size $m_{k-1} \times m_k$. Therefore, we *sequentially* train each of these RBMs by copying the output of one RBM into the input of another.

An example of a stacked RBM is shown on the left-hand side of Figure 6.10(a). This type of RBM is often shown with the concise diagram on the right-hand side of Figure 6.10(a). Note that the copying between two RBMs is a simple one-to-one copying between corresponding nodes, because the output layer of the rth RBM has exactly the same number of nodes as the input layer of the $(r+1)$th RBM. The resulting representations are *unsupervised* because they do not depend on a specific target. Another point is that the Boltzmann machine is an undirected model. However, by stacking the Boltzmann machine, we no longer have an undirected model because the upper layers receive feedback from the lower layers, but not vice versa. In fact, one can treat each Boltzmann machine as a single computational unit with many inputs and outputs, and the copying from one machine to another as the data transmission between two computational units. From this particular view of the stack of Boltzmann machines as a computational graph, it is even possible to perform backpropagation if one reverts to using the sigmoid units to create real-valued activations rather than to create the parameters needed for drawing binary samples. Although the use of real-valued activations is only an approximation, it already provides an excellent approximation because of the way in which the Boltzmann machine has been trained. This initial set of weights can be fine-tuned with backpropagation. After all, backpropagation can be performed on any computational graph, irrespective of the nature of the function computed inside the graph as long as a continuous function is computed. The fine tuning of backpropagation approach is particularly essential in the case of supervised learning, because the weights learned from a Boltzmann machine are always unsupervised.

6.7.1 Unsupervised Learning

Even in the case of unsupervised learning, the stacked RBM will generally provide reductions of better quality than a single RBM. However, the training of this RBM has to be performed carefully because results of high quality are not obtained by simply training all the layers

together. Better results are obtained by using a pretraining approach. Each of the three RBMs in Figure 6.10(a) are trained sequentially. First, RBM1 is trained using the provided training data as the values of the visible units. Then, the outputs of the first RBM are used to train RBM2. This approach is repeated to train RBM3. Note that one can greedily train as many layers as desired using this approach. Assume that the weight matrices for the three learned RBMs are W_1, W_2, and W_3, respectively. Once these weight matrices have been learned, one can put together an encoder-decoder pair with these three weight matrices as shown in Figure 6.10(b). The three decoders have weight matrices W_1^T, W_2^T, and W_3^T, because they perform the inverse operations of encoders. As a result, one now has a directed encoder-decoder network that can be trained with backpropagation like any conventional neural network. The states in this network are computed using directed probabilistic operations, rather than sampled with the use of Monte-Carlo methods. One can perform backpropagation through the layers in order to fine-tune the learning. Note that the weight matrices on the right-hand side of Figure 6.10(b) have been adjusted as a result of this fine tuning. Furthermore, the weight matrices of the encoder and the decoder are no longer related in a symmetric way as a result of the fine tuning. Such stacked RBMs provide reductions of higher quality compared to those with shallower RBMs [414], which is analogous to the behavior of conventional neural networks.

6.7.2 Supervised Learning

How can one learn the weights in such a way that the Boltzmann machine is encouraged to produce a particular type of output such a class labels? Imagine that one wants to perform a k-way classification with a stack of RBMs. The use of a single-layer RBM for classification has already been discussed in Section 6.5.3, and the corresponding architecture is illustrated in Figure 6.7. This architecture can be modified by replacing the single hidden layer with a stack of hidden layers. The final layer of hidden features are then connected to the visible softmax layer that outputs the k probabilities corresponding to the different classes. As in the case of dimensionality reduction, pretraining is helpful Therefore, the first phase is completely unsupervised in which the class labels are not used. In other words, we train the weights of each hidden layer separately. This is achieved by training the weights of the lower layers first and then the higher layers, as in any stacked RBM. After the initial weights have been set in an unsupervised way, one can perform the initial training of weights between the final hidden layer and visible layer of softmax units. One can then create a directed computational graph with these initial weights, as in the case of the unsupervised setting. Backpropagation is performed on this computational graph in order to perform fine tuning of the learned weights.

6.7.3 Deep Boltzmann Machines and Deep Belief Networks

One can stack the different layers of the RBM in various ways to achieve different types of goals. In some forms of stacking, the interactions between different Boltzmann machines are bi-directional. This variation is referred to as a *deep Boltzmann machine*. In other forms of stacking, some of the layers are uni-directional, whereas others are bi-directional. An example is a *deep belief network* in which only the upper RBM is bi-directional, whereas the lower layers are uni-directional. Some of these methods can be shown to be equivalent to various types of probabilistic graphical models like *sigmoid belief nets* [350].

A deep Boltzmann machine is particularly noteworthy because of the bi-directional connections between each pair of units. The fact that the copying occurs both ways means

that we can merge the nodes in adjacent nodes of two RBMs into a single layer of nodes. Furthermore, observe that one could rearrange the RBM into a bipartite graph by putting all the odd layers in one set and the even layers in another set. In other words, the deep RBM is equivalent to a single RBM. The difference from a single RBM is that the visible units form only a small subset of the units in one layer, and all pairs of nodes are not connected. Because of the fact that all pairs of nodes are not connected, the nodes in the upper layers tend to receive smaller weights than the nodes in the lower layers. As a result, pretraining again becomes necessary in which the lower layers are trained first, and then followed up with the higher layers in a greedy way. Subsequently, all layers are trained together in order to fine-tune the method. Refer to the bibliographic notes for details of these advanced models.

6.8 Summary

The earliest variant of the Boltzmann machine was the Hopfield network. The Hopfield network is an energy-based model, which stores the training data instances in its local minima. The Hopfield network can be trained with the Hebbian learning rule. A stochastic variant of the Hopfield network is the Boltzmann machine, which uses a probabilistic model to achieve greater generalization. Furthermore, the hidden states of the Boltzmann machine hold a reduced representation of the data. The Boltzmann machine can be trained with a stochastic variant of the Hebbian learning rule. The main challenge in the case of the Boltzmann machine is that it requires Gibbs sampling, which can be slow in practice. The restricted Boltzmann machine allows connections only between hidden nodes and visible nodes, which eases the training process. More efficient training algorithms are available for the restricted Boltzmann machine. The restricted Boltzmann machine can be used as a dimensionality reduction method; it can also be used in recommender systems with incomplete data. The restricted Boltzmann machine has also been generalized to count data, ordinal data, and real-valued data. However, the vast majority of RBMs are still constructed under the assumption of binary units. In recent years, several deep variants of the restricted Boltzmann machine have been proposed, which can be used for conventional machine learning applications like classification.

6.9 Bibliographic Notes

The earliest variant of the Boltzmann family of models was the Hopfield network [207]. The Storkey learning rule is proposed in [471]. The earliest algorithms for learning Boltzmann machines with the use of Monte Carlo sampling were proposed in [1, 197]. Discussions of Markov Chain Monte Carlo methods are provided in [138, 351], and many of these methods are useful for Boltzmann machines as well. RBMs were originally invented by Smolensky, and referred to as the harmonium. A tutorial on energy-based models is provided in [280]. Boltzmann machines are hard to train because of the interdependent stochastic nature of the units. The intractability of the partition function also makes the learning of the Boltzmann machine hard. However, one can estimate the partition function with *annealed importance sampling* [352]. A variant of the Boltzmann machine is the *mean-field Boltzmann machine* [373], which uses deterministic real units rather than stochastic units. However, the approach is a heuristic and hard to justify. Nevertheless, the use of real-valued approximations is popular at inference time. In other words, a traditional neural network

with real-valued activations and derived weights from the trained Boltzmann machine is often used for prediction. Other variations of the RBM, such as the neural autoregressive distribution estimator [265], can be viewed as autoencoders.

The efficient mini-batch algorithm for Boltzmann machines is described in [491]. The contrastive divergence algorithm, which is useful for RBMs, is described in [61, 191]. A variation referred to as *persistent contrastive divergence* is proposed in [491]. The idea of gradually increasing the value of k in CD_k over the progress of training was proposed in [61]. The work in [61] showed that even a single iteration of the Gibbs sampling approach (which greatly reduces burn-in time) produces only a small bias in the final result, which can be reduced by gradually increasing the value of k in CD_k over the course of training. This insight was key to the efficient implementation of the RBM. An analysis of the bias in the contrastive divergence algorithm may be found in [29]. The work in [479] analyzes the convergence properties of the RBM. It is also shows that the contrastive divergence algorithm is a heuristic, which does not really optimize any objective function. A discussion and practical suggestions for training Boltzmann machines may be found in [119, 193]. The universal approximation property of RBMs is discussed in [341].

RBMs have been used for a variety of applications like dimensionality reduction, collaborative filtering, topic modeling and classification. The use of the RBM for collaborative filtering is discussed in [414]. This approach is instructive because it also shows how one can use an RBM for categorical data containing a small number of values. The application of discriminative restricted Boltzmann machines to classification is discussed in [263, 264]. The topic modeling of documents with Boltzmann machines with softmax units (as discussed in the chapter) is based on [469]. Advanced RBMs for topic modeling with a Poisson distribution are discussed in [134, 538]. The main problem with these methods is that they are unable to work well with documents of varying lengths. The use of replicated softmax is discussed in [199]. This approach is closely connected to ideas from *semantic hashing* [415].

Most of the RBMs are proposed for binary data. However, in recent years, RBMs have also been generalized to other data types. The modeling of count data with softmax units is discussed in the context of topic modeling in [469]. The challenges associated with this type of modeling are discussed in [86]. The use of the RBM for the exponential distribution family is discussed in [522], and discussion for real-valued data is provided in [348]. The introduction of binomial units to encode more information than binary units was proposed in [495]. This approach was shown to be a noisy version of the ReLU [348]. The replacement of binary units with linear units containing Gaussian noise was first proposed in [124]. The modeling of documents with deep Boltzmann machines is discussed in [469]. Boltzmann machines have also been used for multimodal learning with images and text [357, 468].

Training of deep variations of Boltzmann machines provided the first deep learning algorithms that worked well [196]. These algorithms were the first pretraining methods, which were later generalized to other types of neural networks. A detailed discussion of pretraining may be found in Section 4.7 of Chapter 4. Deep Boltzmann machines are discussed in [417], and efficient algorithms are discussed in [200, 418].

Several architectures that are related to the Boltzmann machine provide different types of modeling capabilities. The Helmholtz machine and a wake-sleep algorithm are proposed in [195]. RBMs and their multilayer variants can be shown to be equivalent to different types of probabilistic graphical models such as sigmoid belief nets [350]. A detailed discussion of probabilistic graphical models may be found in [251]. In higher-order Boltzmann machines, the energy function is defined by groups of k nodes for $k > 2$. For example, an order-3 Boltzmann machine will contain terms of the form $w_{ijk}s_i s_j s_k$. Such higher-order machines

are discussed in [437]. Although these methods are potentially more powerful that traditional Boltzmann machines, they have not found much popularity because of the large amount of data they require to train.

6.10 Exercises

1. This chapter discusses how Boltzmann machines can be used for collaborative filtering. Even though discrete sampling of the contrastive divergence algorithm is used for learning the model, the final phase of inference is done using real-valued sigmoid and softmax activations. Discuss how you can use this fact to your advantage in order to fine-tune the learned model with backpropagation.

2. Implement the contrastive divergence algorithm of a restricted Boltzmann machine. Also implement the inference algorithm for deriving the probability distribution of the hidden units for a given test example. Use Python or any other programming language of your choice.

3. Consider a Boltzmann machine without a bipartite restriction (of the RBM), but with the restriction that all units are visible. Discuss how this restriction simplifies the training process of the Boltzmann machine.

4. Propose an approach for using RBMs for outlier detection.

5. Derive the weight updates for the RBM-based topic modeling approach discussed in the chapter. Use the same notations.

6. Show how you can extend the RBM for collaborative filtering (discussed in Section 6.5.2 of the chapter) with additional layers to make it more powerful.

7. A discriminative Boltzmann machine is introduced for classification towards the end of Section 6.5.3. However, this approach is designed for binary classification. Show how you can extend the approach to multi-way classification.

8. Show how you can modify the topic modeling RBM discussed in the chapter in order to create a hidden representation of each node drawn from a large, sparse graph (like a social network).

9. Discuss how you can enhance the model of Exercise 8 to include data about an unordered list of keywords associated with each node. (For example, social network nodes are associated with wall-post and messaging content.)

10. Discuss how you can enhance the topic modeling RBM discussed in the chapter with multiple layers.

Chapter 7

Recurrent Neural Networks

> "Democracy is the recurrent suspicion that more than half the people are right more than half the time."—*The New Yorker*, July 3, 1944.

7.1 Introduction

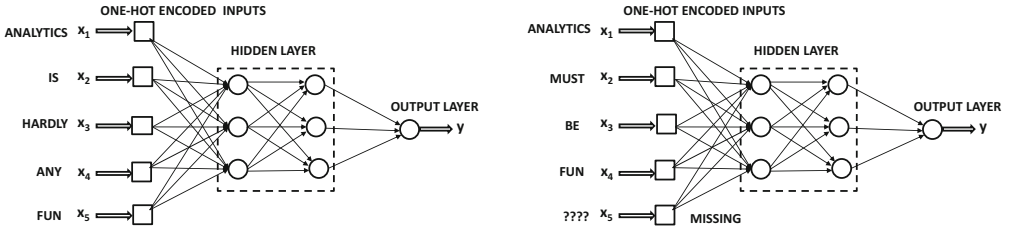
All the neural architectures discussed in earlier chapters are inherently designed for multi-dimensional data in which the attributes are largely independent of one another. However, certain data types such as time-series, text, and biological data contain sequential dependencies among the attributes. Examples of such dependencies are as follows:

1. In a time-series data set, the values on successive time-stamps are closely related to one another. If one uses the values of these time-stamps as independent features, then key information about the relationships among the values of these time-stamps is lost. For example, the value of a time-series at time t is closely related to its values in the previous window. However, this information is lost when the values at individual time-stamps are treated independently of one another.

2. Although text is often processed as a bag of words, one can obtain better semantic insights when the ordering of the words is used. In such cases, it is important to construct models that take the sequencing information into account. Text data is the most common use case of recurrent neural networks.

3. Biological data often contains sequences, in which the symbols might correspond to amino acids or one of the nucleobases that form the building blocks of DNA.

The individual values in a sequence can be either real-valued or symbolic. Real-valued sequences are also referred to as time-series. Recurrent neural networks can be used for either type of data. In practical applications, the use of symbolic values is more common. Therefore, this chapter will primarily focus on symbolic data in general, and on text data in particular. Throughout this chapter, the default assumption will be that the input to the recurrent network will be a text segment in which the corresponding symbols of the sequence are the word identifiers of the lexicon. However, we will also examine other settings, such as cases in which the individual elements are characters or in which they are real values.



(a) 5-word sentence
"*Analytics is hardly any fun.*"

(b) 4-word sentence
"*Analytics must be fun.*"

Figure 7.1: An attempt to use a conventional neural network for sentiment analysis faces the challenge of variable-length inputs. The network architecture also does not contain any helpful information about sequential dependencies among successive words.

Many sequence-centric applications like text are often processed as bags of words. Such an approach ignores the ordering of words in the document, and works well for documents of reasonable size. However, in applications where the semantic interpretation of the sentence is important, or in which the size of the text segment is relatively small (e.g., a single sentence), such an approach is simply inadequate. In order to understand this point, consider the following pair of sentences:

The cat chased the mouse.
The mouse chased the cat.

The two sentences are clearly very different (and the second one is unusual). However, the bag-of-words representation would deem them identical. Hence, this type of representation works well for simpler applications (such as classification), but a greater degree of linguistic intelligence is required for more sophisticated applications in difficult settings such as *sentiment analysis*, *machine translation*, or *information extraction*.

One possible solution is to avoid the bag-of-words approach and create one input for each position in the sequence. Consider a situation in which one tried to use a conventional neural network in order to perform sentiment analysis on sentences with one input for each position in the sentence. The sentiment can be a binary label depending on whether it is positive or negative. The first problem that one would face is that the length of different sentences is different. Therefore, if we used a neural network with 5 sets of one-hot encoded word inputs (cf. Figure 7.1(a)), it would be impossible to enter a sentence with more than five words. Furthermore, any sentence with less than five words would have missing inputs (cf. Figure 7.1(b)). In some cases, such as Web log sequences, the length of the input sequence might run into the hundreds of thousands. More importantly, small changes in word ordering can lead to semantically different connotations, and *it is important to somehow encode information about the word ordering more directly within the architecture of the*

network. The goal of such an approach would be to reduce the parameter requirements with increasing sequence length; recurrent neural networks provide an excellent example of (parameter-wise) *frugal architectural design* with the help of domain-specific insights. Therefore, the two main desiderata for the processing of sequences include (i) the ability to receive and process inputs in the same order as they are present in the sequence, and (ii) the treatment of inputs at each time-stamp in a similar manner in relation to previous history of inputs. A key challenge is that we somehow need to construct a neural network with a fixed number of parameters, but with the ability to process a variable number of inputs.

These desiderata are naturally satisfied with the use of *recurrent neural networks (RNNs)*. In a recurrent neural network, there is a one-to-one correspondence between the layers in the network and the specific positions in the sequence. The position in the sequence is also referred to as its *time-stamp*. Therefore, instead of a variable number of inputs in a single input layer, the network contains a variable number of layers, and each layer has a single input corresponding to that time-stamp. Therefore, the inputs are allowed to directly interact with down-stream hidden layers depending on their positions in the sequence. Each layer uses the same set of parameters to ensure similar modeling at each time stamp, and therefore the number of parameters is fixed as well. In other words, the same layer-wise architecture is repeated in time, and therefore the network is referred to as *recurrent*. Recurrent neural networks are also feed-forward networks with a specific structure based on the notion of *time layering*, so that they can take a *sequence* of inputs and produce a sequence of outputs. Each temporal layer can take in an input data point (either single attribute or multiple attributes), and optionally produce a multidimensional output. Such models are particularly useful for sequence-to-sequence learning applications like machine translation or for predicting the next element in a sequence. Some examples of applications include the following:

1. The input might be a sequence of words, and the output might be the same sequence shifted by 1, so that we are predicting the next word at any given point. This is a classical *language model* in which we are trying the predict the next word based on the sequential history of words. Language models have a wide variety of applications in text mining and information retrieval [6].

2. In a real-valued time-series, the problem of learning the next element is equivalent to *autoregressive analysis*. However, a recurrent neural network can learn far more complex models than those obtained with traditional time-series modeling.

3. The input might be a sentence in one language, and the output might be a sentence in another language. In this case, one can hook up two recurrent neural networks to learn the translation models between the two languages. One can even hook up a recurrent network with a different type of network (e.g., convolutional neural network) to learn captions of images.

4. The input might be a sequence (e.g., sentence), and the output might be a vector of class probabilities, which is triggered by the end of the sentence. This approach is useful for sentence-centric classification applications like sentiment analysis.

From these four examples, it can be observed that a wide variety of basic architectures have been employed or studied within the broader framework of recurrent neural networks.

There are significant challenges in learning the parameters of a recurrent neural network. One of the key problems in this context is that of the vanishing and the exploding gradient

problem. This problem is particularly prevalent in the context of deep networks like recurrent neural networks. As a result, a number of variants of the recurrent neural network, such as long short-term memory (LSTM) and gated recurrent unit (GRU), have been proposed. Recurrent neural networks and their variants have been used in the context of a variety of applications like sequence-to-sequence learning, image captioning, machine translation, and sentiment analysis. This chapter will also study the use of recurrent neural networks in the context of these different applications.

7.1.1 Expressiveness of Recurrent Networks

Recurrent neural networks are known to be *Turing complete* [444]. Turing completeness means that a recurrent neural network can simulate any algorithm, given enough data and computational resources [444]. This property is, however, not very useful in practice because the amount of data and computational resources required to achieve this goal in arbitrary settings can be unrealistic. Furthermore, there are practical issues in training a recurrent neural network, such as the vanishing and exploding gradient problems. These problems increase with the length of the sequence, and more stable variations such as long short-term memory can address this issue only in a limited way. The neural Turing machine is discussed in Chapter 10, which uses external memory to improve the stability of neural network learning. A neural Turing machine can be shown to be equivalent to a recurrent neural network, and it often uses a more traditional recurrent network, referred to as the *controller*, as an important action-deciding component. Refer to Section 10.3 of Chapter 10 for a detailed discussion.

Chapter Organization

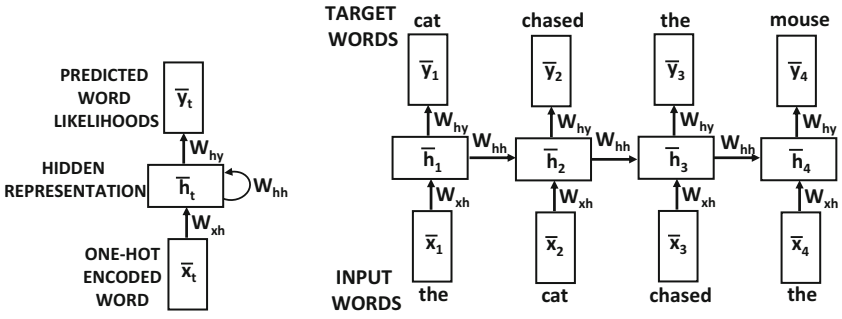
This chapter is organized as follows. The next section will introduce the basic architecture of the recurrent neural network along with the associated training algorithm. The challenges of training recurrent networks are discussed in Section 7.3. Because of these challenges, several variations of the recurrent neural network architecture have been proposed. This chapter will study several such variations. Echo-state networks are introduced in Section 7.4. Long short-term memory networks are discussed in Section 7.5. The gated recurrent unit is discussed in Section 7.6. Applications of recurrent neural networks are discussed in Section 7.7. A summary is given in Section 7.8.

7.2 The Architecture of Recurrent Neural Networks

In the following, the basic architecture of a recurrent network will be described. Although the recurrent neural network can be used in almost any sequential domain, its use in the text domain is both widespread and natural. We will assume the use of the text domain throughout this section in order to enable intuitively simple explanations of various concepts. Therefore, the focus of this chapter will be mostly on discrete RNNs, since that is the most popular use case. Note that exactly the same neural network can be used both for building a word-level RNN and a character-level RNN. The only difference between the two is the set of base symbols used to define the sequence. For consistency, we will stick to the word-level RNN while introducing the notations and definitions. However, variations of this setting are also discussed in this chapter.

The simplest recurrent neural network is shown in Figure 7.2(a). A key point here is the presence of the self-loop in Figure 7.2(a), which will cause the hidden state of the neural

network to change after the input of each word in the sequence. In practice, one only works with sequences of finite length, and it makes sense to unfold the loop into a "time-layered" network that looks more like a feed-forward network. This network is shown in Figure 7.2(b). Note that in this case, we have a different node for the hidden state at each time-stamp and the self-loop has been unfurled into a feed-forward network. This representation is mathematically equivalent to Figure 7.2(a), but is much easier to comprehend because of its similarity to a traditional network. The weight matrices in different temporal layers *are shared* to ensure that the same function is used at each time-stamp. The annotations W_{xh}, W_{hh}, and W_{hy} of the weight matrices in Figure 7.2(b) make the sharing evident.



(a) RNN (b) Time-layered representation of (a)

Figure 7.2: A recurrent neural network and its time-layered representation

It is noteworthy that Figure 7.2 shows a case in which each time-stamp has an input, output, and hidden unit. In practice, it is possible for either the input or the output units to be missing at any particular time-stamp. Examples of cases with missing inputs and outputs are shown in Figure 7.3. The choice of missing inputs and outputs would depend on the specific application at hand. For example, in a time-series forecasting application, we might need outputs at each time-stamp in order to predict the next value in the time-series. On the other hand, in a sequence-classification application, we might only need a single output label at the end of the sequence corresponding to its class. In general, it is possible for any subset of inputs or outputs to be missing in a particular application. The following discussion will assume that all inputs and outputs are present, although it is easy to generalize it to the case where some of them are missing by simply removing the corresponding terms or equations.

The particular architecture shown in Figure 7.2 is suited to language modeling. A language model is a well-known concept in natural language processing that predicts the next word, given the previous history of words. Given a sequence of words, their one-hot encoding is fed one at a time to the neural network in Figure 7.2(a). This temporal process is equivalent to feeding the individual words to the inputs at the relevant time-stamps in Figure 7.2(b). A time-stamp corresponds to the position in the sequence, which starts at 0 (or 1), and increases by 1 by moving forward in the sequence by one unit. In the setting of language modeling, the output is a vector of probabilities predicted for the next word in the sequence. For example, consider the sentence:

The cat chased the mouse.

When the word "*The*" is input, the output will be a vector of probabilities of the entire lexicon that includes the word "*cat*," and when the word "*cat*" is input, we will again get a

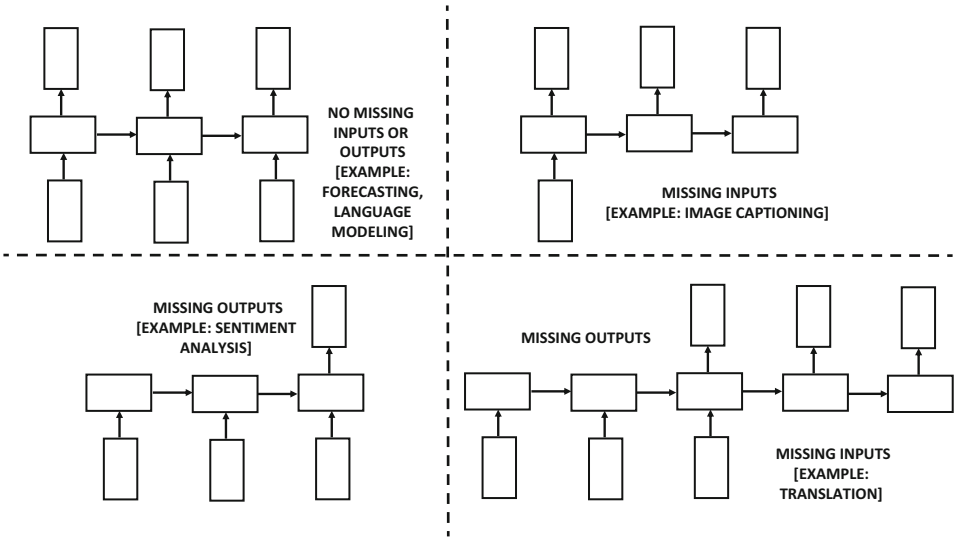


Figure 7.3: The different variations of recurrent networks with missing inputs and outputs

vector of probabilities predicting the next word. This is, of course, the classical definition of a language model in which the probability of a word is estimated based on the immediate history of previous words. In general, the input vector at time t (e.g., one-hot encoded vector of the tth word) is $\overline{x}_t$, the hidden state at time t is $\overline{h}_t$, and the output vector at time t (e.g., predicted probabilities of the $(t+1)$th word) is $\overline{y}_t$. Both $\overline{x}_t$ and $\overline{y}_t$ are d-dimensional for a lexicon of size d. The hidden vector $\overline{h}_t$ is p-dimensional, where p regulates the complexity of the embedding. For the purpose of discussion, we will assume that all these vectors are column vectors. In many applications like classification, the output is not produced at each time unit but is only triggered at the last time-stamp in the end of the sentence. Although output and input units may be present only at a subset of the time-stamps, we examine the simple case in which they are present in all time-stamps. Then, the hidden state at time t is given by a function of the input vector at time t and the hidden vector at time $(t-1)$:

$$\overline{h}_t = f(\overline{h}_{t-1}, \overline{x}_t) \tag{7.1}$$

This function is defined with the use of weight matrices and activation functions (as used by all neural networks for learning), and *the same weights are used at each time-stamp*. Therefore, even though the hidden state evolves over time, the weights and the underlying function $f(\cdot, \cdot)$ remain fixed over all time-stamps (i.e., sequential elements) after the neural network has been trained. A separate function $\overline{y}_t = g(\overline{h}_t)$ is used to learn the output probabilities from the hidden states.

Next, we describe the functions $f(\cdot, \cdot)$ and $g(\cdot)$ more concretely. We define a $p \times d$ input-hidden matrix W_{xh}, a $p \times p$ hidden-hidden matrix W_{hh}, and a $d \times p$ hidden-output matrix W_{hy}. Then, one can expand Equation 7.1 and also write the condition for the outputs as follows:

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1})$$
$$\overline{y}_t = W_{hy}\overline{h}_t$$

Here, the "tanh" notation is used in a relaxed way, in the sense that the function is applied to the p-dimensional column vector in an element-wise fashion to create a p-dimensional vector with each element in $[-1, 1]$. Throughout this section, this relaxed notation will be used for several activation functions such as tanh and sigmoid. In the very first time-stamp, $\overline{h}_{t-1}$ is assumed to be some default constant vector (such as 0), because there is no input from the hidden layer at the beginning of a sentence. One can also learn this vector, if desired. Although the hidden states change at each time-stamp, the weight matrices stay fixed over the various time-stamps. Note that the output vector $\overline{y}_t$ is a set of continuous values with the same dimensionality as the lexicon. A softmax layer is applied on top of $\overline{y}_t$ so that the results can be interpreted as probabilities. *The p-dimensional output $\overline{h}_t$ of the hidden layer at the end of a text segment of t words yields its embedding, and the p-dimensional columns of W_{xh} yield the embeddings of individual words.* The latter provides an alternative to *word2vec* embeddings (cf. Chapter 2).

Because of the recursive nature of Equation 7.1, the recurrent network has the *ability to compute a function of variable-length inputs*. In other words, one can expand the recurrence of Equation 7.1 to define the function for $\overline{h}_t$ in terms of t inputs. For example, starting at $\overline{h}_0$, which is typically fixed to some constant vector (such as the zero vector), we have $\overline{h}_1 = f(\overline{h}_0, \overline{x}_1)$ and $\overline{h}_2 = f(f(\overline{h}_0, \overline{x}_1), \overline{x}_2)$. Note that $\overline{h}_1$ is a function of only $\overline{x}_1$, whereas $\overline{h}_2$ is a function of both $\overline{x}_1$ and $\overline{x}_2$. In general, $\overline{h}_t$ is a function of $\overline{x}_1 \ldots \overline{x}_t$. Since the output $\overline{y}_t$ is a function of $\overline{h}_t$, these properties are inherited by $\overline{y}_t$ as well. In general, we can write the following:

$$\overline{y}_t = F_t(\overline{x}_1, \overline{x}_2, \ldots \overline{x}_t) \tag{7.2}$$

Note that the function $F_t(\cdot)$ varies with the value of t although its relationship to its immediately previous state is always the same (based on Equation 7.1). Such an approach is particularly useful for variable-length inputs. This setting occurs often in many domains like text in which the sentences are of variable length. For example, in a language modeling application, the function $F_t(\cdot)$ indicates the probability of the next word, taking into account all the previous words in the sentence.

7.2.1 Language Modeling Example of RNN

In order to illustrate the workings of the RNN, we will use a toy example of a single sequence defined on a vocabulary of four words. Consider the sentence:

The cat chased the mouse.

In this case, we have a lexicon of four words, which are {"*the*," "*cat*," "*chased*," "*mouse*"}. In Figure 7.4, we have shown the probabilistic prediction of the next word at each of time-stamps from 1 to 4. Ideally, we would like the probability of the next word to be predicted correctly from the probabilities of the previous words. Each one-hot encoded input vector $\overline{x}_t$ has length four, in which only one bit is 1 and the remaining bits are 0s. The main flexibility here is in the dimensionality p of the hidden representation, which we set to 2 in this case. As a result, the matrix W_{xh} will be a 2×4 matrix, so that it maps a one-hot encoded input vector into a hidden vector $\overline{h}_t$ vector of size 2. As a practical matter, each column of W_{xh} corresponds to one of the four words, and one of these columns is copied by the expression $W_{xh}\overline{x}_t$. Note that this expression is added to $W_{hh}\overline{h}_t$ and then transformed with the tanh function to produce the final expression. The final output $\overline{y}_t$ is defined by $W_{hy}\overline{h}_t$. Note that the matrices W_{hh} and W_{hy} are of sizes 2×2 and 4×2, respectively.

In this case, the outputs are continuous values (not probabilities) in which larger values indicate greater likelihood of presence. These continuous values are eventually converted

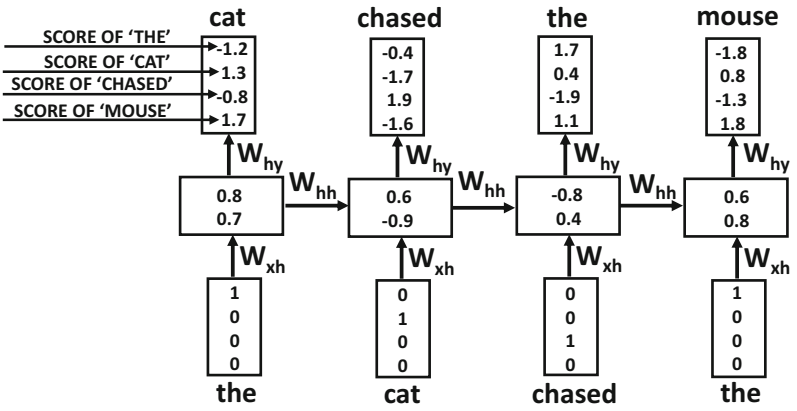


Figure 7.4: Example of language modeling with a recurrent neural network

to probabilities with the softmax function, and therefore one can treat them as substitutes to log probabilities. The word "*cat*" is predicted in the first time-stamp with a value of 1.3, although this value seems to be (incorrectly) outstripped by "*mouse*" for which the corresponding value is 1.7. However, the word "*chased*" seems to be predicted correctly at the next time-stamp. As in all learning algorithms, one cannot hope to predict every value exactly, and such errors are more likely to be made in the early iterations of the backpropagation algorithm. However, as the network is repeatedly trained over multiple iterations, it makes fewer errors over the training data.

7.2.1.1 Generating a Language Sample

Such an approach can also be used to generate an arbitrary sample of a language, once the training has been completed. How does one use such a language model at testing time, since each state requires an input word, and none is available during language generation? The likelihoods of the tokens at the first time-stamp can be generated using the <START> token as input. Since the <START> token is also available in the training data, the model will typically select a word that often starts text segments. Subsequently, the idea is to sample one of the tokens generated at each time-stamp (based on the predicted likelihood), and then use it as an input to the next time-stamp. To improve the accuracy of the sequentially predicted token, one might use beam search to expand on the most likely possibilities by always keeping track of the b best sequence prefixes of any particular length. The value of b is a user-driven parameter. By recursively applying this operation, one can generate an arbitrary sequence of text that reflects the particular training data at hand. If the <END> token is predicted, it indicates the end of that particular segment of text. Although such an approach often results in syntactically correct text, it might be nonsensical in meaning. For example, a character-level RNN[1] authored by Karpathy, Johnson, and Fei Fei [233, 580] was trained on William Shakespeare's plays. A character-level RNN requires the neural network to learn both syntax *and* spelling. After only five iterations of learning across the full data set, the following was a sample of the output:

[1] A long-short term memory network (LSTM) was used, which is a variation on the vanilla RNN discussed here.

KING RICHARD II:
Do cantant,-'for neight here be with hand her,-
Eptar the home that Valy is thee.

NORONCES:
Most ma-wrow, let himself my hispeasures;
An exmorbackion, gault, do we to do you comforr,
Laughter's leave: mire sucintracce shall have theref-Helt.

Note that there are a large number of misspellings in this case, and a lot of the words are gibberish. However, when the training was continued to 50 iterations, the following was generated as a part of the sample:

KING RICHARD II:
Though they good extremit if you damed;
Made it all their fripts and look of love;
Prince of forces to uncertained in conserve
To thou his power kindless. A brives my knees
In penitence and till away with redoom.

GLOUCESTER:
Between I must abide.

This generated piece of text is largely consistent with the syntax and spelling of the archaic English in William Shakespeare's plays, although there are still some obvious errors. Furthermore, the approach also indents and formats the text in a manner similar to the plays by placing new lines at reasonable locations. Continuing to train for more iterations makes the output almost error-free, and some impressive samples are also available at [235].

Of course, the semantic meaning of the text is limited, and one might wonder about the usefulness of generating such nonsensical pieces of text from the perspective of machine learning applications. The key point here is that by providing an additional *contextual* input, such as the neural representation of an image, the neural network can be made to give intelligent outputs such as a grammatically correct description (i.e., caption) of the image. In other words, language models are best used by generating *conditional* outputs.

The primary goal of the language-modeling RNN is not to create arbitrary sequences of the language, but to provide an architectural base that can be modified in various ways to incorporate the effect of the specific context. For example, applications like machine translation and image captioning learn a language model that is *conditioned* on another input such as a sentence in the source language or an image to be captioned. Therefore, the precise design of the application-dependent RNN will use the same principles as the language-modeling RNN, but will make small changes to this basic architecture in order to incorporate the specific context. In all these cases, the key is in choosing the input and output values of the recurrent units in a judicious way, so that one can backpropagate the output errors and learn the weights of the neural network in an application-dependent way.

7.2.2 Backpropagation Through Time

The negative logarithms of the softmax probability of the correct words at the various time-stamps are aggregated to create the loss function. The softmax function is described in Section 3.2.5.1 of Chapter 3, and we directly use those results here. If the output vector $\overline{y}_t$ can be written as $[\hat{y}_t^1 \ldots \hat{y}_t^d]$, it is first converted into a vector of d probabilities using the softmax function:

$$[\hat{p}_t^1 \ldots \hat{p}_t^d] = \text{Softmax}([\hat{y}_t^1 \ldots \hat{y}_t^d])$$

The softmax function above can be found in Equation 3.20 of Chapter 3. If j_t is the index of the ground-truth word at time t in the training data, then the loss function L for all T time-stamps is computed as follows:

$$L = -\sum_{t=1}^{T} \log(\hat{p}_t^{j_t}) \tag{7.3}$$

This loss function is a direct consequence of Equation 3.21 of Chapter 3. The derivative of the loss function with respect to the raw outputs may be computed as follows (cf. Equation 3.22 of Chapter 3):

$$\frac{\partial L}{\partial \hat{y}_t^k} = \hat{p}_t^k - I(k, j_t) \tag{7.4}$$

Here, $I(k, j_t)$ is an indicator function that is 1 when k and j_t are the same, and 0, otherwise. Starting with this partial derivative, one can use the straightforward backpropagation update of Chapter 3 (on the unfurled temporal network) to compute the gradients with respect to the weights in different layers. The main problem is that the weight sharing across different temporal layers will have an effect on the update process. An important assumption in correctly using the chain rule for backpropagation (cf. Chapter 3) is that the weights in different layers are distinct from one another, which allows a relatively straightforward update process. However, as discussed in Section 3.2.9 of Chapter 3, it is not difficult to modify the backpropagation algorithm to handle shared weights.

The main trick for handling shared weights is to first "pretend" that the parameters in the different temporal layers are independent of one another. For this purpose, we introduce the temporal variables $W_{xh}^{(t)}$, $W_{hh}^{(t)}$ and $W_{hy}^{(t)}$ for time-stamp t. Conventional backpropagation is first performed by working under the pretense that these variables are distinct from one another. Then, the contributions of the different temporal avatars of the weight parameters to the gradient are added to create a unified update for each weight parameter. This special type of backpropagation algorithm is referred to as *backpropagation through time (BPTT)*. We summarize the BPTT algorithm as follows:

(i) We run the input sequentially in the forward direction through time and compute the errors (and the negative-log loss of softmax layer) at each time-stamp.

(ii) We compute the gradients of the edge weights in the backwards direction on the unfurled network without any regard for the fact that weights in different time layers are shared. In other words, it is assumed that the weights $W_{xh}^{(t)}$, $W_{hh}^{(t)}$ and $W_{hy}^{(t)}$ in time-stamp t are distinct from other time-stamps. As a result, one can use conventional backpropagation to compute $\frac{\partial L}{\partial W_{xh}^{(t)}}$, $\frac{\partial L}{\partial W_{hh}^{(t)}}$, and $\frac{\partial L}{\partial W_{hy}^{(t)}}$. Note that we have used matrix calculus notations where the derivative with respect to a matrix is defined by a corresponding matrix of element-wise derivatives.

(iii) We add all the (shared) weights corresponding to different instantiations of an edge in time. In other words, we have the following:

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{xh}^{(t)}}$$
$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{hh}^{(t)}}$$
$$\frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{hy}^{(t)}}$$

The above derivations follow from a straightforward application of the multivariate chain rule. As in all backpropagation methods with shared weights (cf. Section 3.2.9 of Chapter 3), we are using the fact that the partial derivative of a temporal copy of each parameter (such as an element of $W_{xh}^{(t)}$) with respect to the original copy of the parameter (such as the corresponding element of W_{xh}) can be set to 1. Here, it is noteworthy that the computation of the partial derivatives with respect to the temporal copies of the weights is not different from traditional backpropagation at all. Therefore, one only needs to wrap the temporal aggregation around conventional backpropagation in order to compute the update equations. The original algorithm for backpropagation through time can be credited to Werbos's seminal work in 1990 [526], long before the use of recurrent neural networks became more popular.

Truncated Backpropagation Through Time

One of the computational problems in training recurrent networks is that the underlying sequences may be very long, as a result of which the number of layers in the network may also be very large. This can result in computational, convergence, and memory-usage problems. This problem is solved by using *truncated backpropagation through time*. This technique may be viewed as the analog of stochastic gradient descent for recurrent neural networks. In the approach, the state values are computed correctly during forward propagation, but the backpropagation updates are done only over segments of the sequence of modest length (such as 100). In other words, only the portion of the loss over the relevant segment is used to compute the gradients and update the weights. The segments are processed in the same order as they occur in the input sequence. The forward propagation does not need to be performed in a single shot, but it can also be done over the relevant segment of the sequence as long as the values in the final time-layer of the segment are used for computing the state values in the next segment of layers. The values in the final layer in the current segment are used to compute the values in the first layer of the next segment. Therefore, forward propagation is always able to accurately maintain state values, although the backpropagation uses only a small portion of the loss. Here, we have described truncated BPTT using non-overlapping segments for simplicity. In practice, one can update using overlapping segments of inputs.

Practical Issues

The entries of each weight matrix are initialized to small values in $[-1/\sqrt{r}, 1/\sqrt{r}]$, where r is the number of columns in that matrix. One can also initialize each of the d columns of the input weight matrix W_{xh} to the *word2vec* embedding of the corresponding word

(cf. Chapter 2). This approach is a form of pretraining. The specific advantage of using this type of pretraining depends on the amount of training data. It can be helpful to use this type of initialization when the amount of available training data is small. After all, pretraining is a form of regularization (see Chapter 4).

Another detail is that the training data often contains a special <START> and an <END> token at the beginning and end of each training segment. These types of tokens help the model to recognize specific text units such as sentences, paragraphs, or the beginning of a particular module of text. The distribution of the words at the beginning of a segment of text is often very different than how it is distributed over the whole training data. Therefore, after the occurrence of <START>, the model is more likely to pick words that begin a particular segment of text.

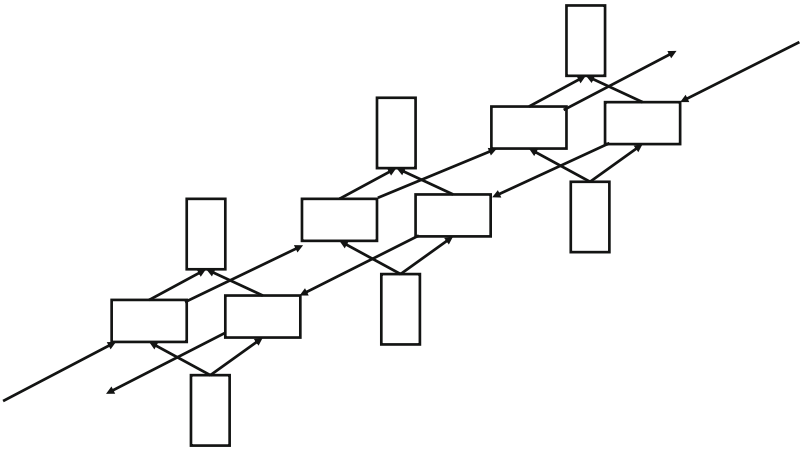


Figure 7.5: Showing three time-layers of a bidirectional recurrent network

There are other approaches that are used for deciding whether to end a segment at a particular point. A specific example is the use of a binary output that decides whether or not the sequence should continue at a particular point. Note that the binary output is in addition to other application-specific outputs. Typically, the sigmoid activation is used to model the prediction of this output, and the cross-entropy loss is used on this output. Such an approach is useful with real-valued sequences. This is because the use of <START> and <END> tokens is inherently designed for symbolic sequences. However, one disadvantage of this approach is that it changes the loss function from its application-specific formulation to one that provides a balance between end-of-sequence prediction and application-specific needs. Therefore, the weights of different components of the loss function would be yet another hyper-parameter that one would have to work with.

There are also several practical challenges in training an RNN, which make the design of various architectural enhancements of the RNN necessary. It is also noteworthy that multiple hidden layers (with long short-term memory enhancements) are used in all practical applications, which will be discussed in Section 7.2.4. However, the application-centric exposition will use the simpler single-layer model for clarity. The generalization of each of these applications to enhanced architectures is straightforward.

7.2.3 Bidirectional Recurrent Networks

One disadvantage of recurrent networks is that the state at a particular time unit only has knowledge about the past inputs up to a certain point in a sentence, but it has no knowledge about future states. In certain applications like language modeling, the results are vastly improved with knowledge about both past and future states. A specific example is handwriting recognition in which there is a clear advantage in using knowledge about both the past and future symbols, because it provides a better idea of the underlying context.

In the bidirectional recurrent network, we have separate hidden states $\overline{h}_t^{(f)}$ and $\overline{h}_t^{(b)}$ for the forward and backward directions. The forward hidden states interact only with each other and the same is true for the backward hidden states. The main difference is that the forward states interact in the forwards direction, while the backwards states interact in the backwards direction. Both $\overline{h}_t^{(f)}$ and $\overline{h}_t^{(b)}$, however, receive input from the same vector $\overline{x}_t$ (e.g., one-hot encoding of word) and they interact with the same output vector $\hat{y}_t$. An example of three time-layers of the bidirectional RNN is shown in Figure 7.5.

There are several applications in which one tries to predict the properties of the current tokens, such as the recognition of the characters in a handwriting sample, a the parts of speech in a sentence, or the classification of each token of the natural language. In general, any property of the *current* word can be predicted more effectively using this approach, because it uses the context on both sides. For example, the ordering of words in several languages is somewhat different depending on grammatical structure. Therefore, a bidirectional recurrent network often models the hidden representations of any specific point in the sentence in a more robust way with the use of backwards and forwards states, irrespective of the specific nuances of language structure. In fact, it has increasingly become more common to use bidirectional recurrent networks in various language-centric applications like speech recognition.

In the case of the bidirectional network, we have separate forward and backward parameter matrices. The forward matrices for the input-hidden, hidden-hidden, and hidden-output interactions are denoted by $W_{xh}^{(f)}$, $W_{hh}^{(f)}$, and $W_{hy}^{(f)}$, respectively. The backward matrices for the input-hidden, hidden-hidden, and hidden-output interactions are denoted by $W_{xh}^{(b)}$, $W_{hh}^{(b)}$, and $W_{hy}^{(b)}$, respectively.

The recurrence conditions can be written as follows:

$$\overline{h}_t^{(f)} = \tanh(W_{xh}^{(f)}\overline{x}_t + W_{hh}^{(f)}\overline{h}_{t-1}^{(f)})$$
$$\overline{h}_t^{(b)} = \tanh(W_{xh}^{(b)}\overline{x}_t + W_{hh}^{(b)}\overline{h}_{t+1}^{(b)})$$
$$\overline{y}_t = W_{hy}^{(f)}\overline{h}_t^{(f)} + W_{hy}^{(b)}\overline{h}_t^{(b)}$$

It is easy to see that the bidirectional equations are simple generalizations of the conditions used in a single direction. It is assumed that there are a total of T time-stamps in the neural network shown above, where T is the length of the sequence. One question is about the forward input at the boundary conditions corresponding to $t = 1$ and the backward input at $t = T$, which are not defined. In such cases, one can use a default constant value of 0.5 in each case, although one can also make the determination of these values as a part of the learning process.

An immediate observation about the hidden states in the forward and backwards direction is that they do not interact with one another at all. Therefore, one could first run the sequence in the forward direction to compute the hidden states in the forward direction, and then run the sequence in the backwards direction to compute the hidden states in the

backwards direction. At this point, the output states are computed from the hidden states in the two directions.

After the outputs have been computed, the backpropagation algorithm is applied to compute the partial derivatives with respect to various parameters. First, the partial derivatives are computed with respect to the output states because both forward and backwards states point to the output nodes. Then, the backpropagation pass is computed only for the forward hidden states starting from $t = T$ down to $t = 1$. The backpropagation pass is finally computed for the backwards hidden states from $t = 1$ to $t = T$. Finally, the partial derivatives with respect to the shared parameters are aggregated. Therefore, the BPTT algorithm can be modified easily to the case of bidirectional networks. One can summarize the steps as follows:

1. Compute forward and backwards hidden states in independent and separate passes.
2. Compute output states from backwards and forward hidden states.
3. Compute partial derivatives of loss with respect to output states and each copy of the output parameters.
4. Compute partial derivatives of loss with respect to forward states and backwards states independently using backpropagation. Use these computations to evaluate partial derivatives with respect to each copy of the forwards and backwards parameters.
5. Aggregate partial derivatives over shared parameters.

Bidirectional recurrent neural networks are appropriate for applications in which the predictions are not causal based on a historical window. A classical example of a causal setting is a stream of symbols in which an event is predicted on the basis of the history of previous symbols. Even though language-modeling applications are formally considered causal applications (i.e., based on immediate history of *previous* words), the reality is that a given word can be predicted with much greater accuracy through the use of the contextual words on each side of it. In general, bidirectional RNNs work well in applications where the predictions are based on bidirectional context. Examples of such applications include handwriting recognition and speech recognition, in which the properties of individual elements in the sequence depend on those on either side of it. For example, if a handwriting is expressed in terms of the strokes, the strokes on either side of a particular position are helpful in recognizing the particular character being synthesized. Furthermore, certain characters are more likely to be adjacent than others.

A bidirectional neural network achieves almost the same quality of results as using an ensemble of two separate recurrent networks, one in which the input is presented in original form and the other in which the input is reversed. The main difference is that the parameters of the forwards and backwards states are trained jointly in this case. However, this integration is quite weak because the two types of states do not interact directly with one another.

7.2.4 Multilayer Recurrent Networks

In all the aforementioned applications, a single-layer RNN architecture is used for ease in understanding. However, in practical applications, a multilayer architecture is used in order to build models of greater complexity. Furthermore, this multilayer architecture can be used in combination with advanced variations of the RNN, such as the LSTM architecture or the gated recurrent unit. These advanced architectures are introduced in later sections.

An example of a deep network containing three layers is shown in Figure 7.6. Note that nodes in higher-level layers receive input from those in lower-level layers. The relationships among the hidden states can be generalized directly from the single-layer network. First, we rewrite the recurrence equation of the hidden layers (for single-layer networks) in a form that can be adapted easily to multilayer networks:

$$\begin{aligned}\overline{h}_t &= \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1}) \\ &= \tanh\ W \left[\begin{array}{c} \overline{x}_t \\ \overline{h}_{t-1} \end{array} \right]\end{aligned}$$

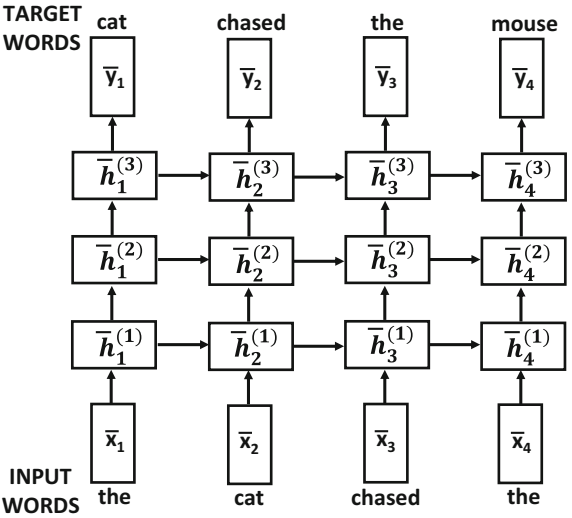


Figure 7.6: Multi-layer recurrent neural networks

Here, we have put together a larger matrix $W = [W_{xh}, W_{hh}]$ that includes the columns of W_{xh} and W_{hh}. Similarly, we have created a larger column vector that stacks up the state vector in the first hidden layer at time $t-1$ and the input vector at time t. In order to distinguish between the hidden nodes for the upper-level layers, let us add an additional superscript to the hidden state and denote the vector for the hidden states at time-stamp t and layer k by $\overline{h}_t^{(k)}$. Similarly, let the weight matrix for the kth hidden layer be denoted by $W^{(k)}$. It is noteworthy that the weights are shared across different time-stamps (as in the single-layer recurrent network), but they are not shared across different layers. Therefore, the weights are superscripted by the layer index k in $W^{(k)}$. The first hidden layer is special because it receives inputs both from the input layer at the current time-stamp and the adjacent hidden state at the previous time-stamp. Therefore, the matrices $W^{(k)}$ will have a size of $p \times (d+p)$ only for the first layer (i.e., $k = 1$), where d is the size of the input vector $\overline{x}_t$ and p is the size of the hidden vector $\overline{h}_t$. Note that d will typically not be the same as p. The recurrence condition for the first layer is already shown above by setting $W^{(1)} = W$. Therefore, let us focus on all the hidden layers k for $k \geq 2$. It turns out that the recurrence condition for the layers with $k \geq 2$ is also in a very similar form as the equation shown above:

$$\overline{h}_t^{(k)} = \tanh\ W^{(k)} \left[\begin{array}{c} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{array} \right]$$

In this case, the size of the matrix $W^{(k)}$ is $p \times (p+p) = p \times 2p$. The transformation from hidden to output layer remains the same as in single-layer networks. It is easy to see that this approach is a straightforward multilayer generalization of the case of single-layer networks. It is common to use two or three layers in practical applications. In order to use a larger number of layers, it is important to have access to more training data in order to avoid overfitting.

7.3 The Challenges of Training Recurrent Networks

Recurrent neural networks are very hard to train because of the fact that the time-layered network is a very deep network, especially if the input sequence is long. In other words, the depth of the temporal layering is input-dependent. As in all deep networks, the loss function has highly varying sensitivities of the loss function (i.e., loss gradients) to different temporal layers. Furthermore, even though the loss function has highly varying gradients to the variables in different layers, the same parameter matrices are shared by different temporal layers. This combination of varying sensitivity and shared parameters in different layers can lead to some unusually unstable effects.

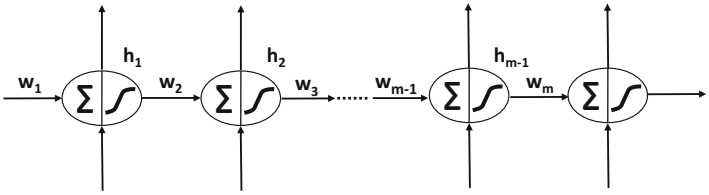


Figure 7.7: The vanishing and exploding gradient problems

The primary challenge associated with a recurrent neural network is that of the *vanishing* and *exploding gradient problems*. This point is explained in detail in Section 3.4 of Chapter 3. In this section, we will revisit this issue in the context of recurrent neural networks. It is easiest to understand the challenges associated with recurrent networks by examining the case of a recurrent network with a single unit in each layer.

Consider a set of T consecutive layers, in which the tanh activation function, $\Phi(\cdot)$, is applied between each pair of layers. The shared weight between a pair of hidden nodes is denoted by w. Let $h_1 \ldots h_T$ be the hidden values in the various layers. Let $\Phi'(h_t)$ be the derivative of the activation function in hidden layer t. Let the copy of the shared weight w in the tth layer be denoted by w_t so that it is possible to examine the effect of the backpropagation update. Let $\frac{\partial L}{\partial h_t}$ be the derivative of the loss function with respect to the hidden activation h_t. The neural architecture is illustrated in Figure 7.7. Then, one derives the following update equations using backpropagation:

$$\frac{\partial L}{\partial h_t} = \Phi'(h_{t+1}) \cdot w_{t+1} \cdot \frac{\partial L}{\partial h_{t+1}} \tag{7.5}$$

Since the shared weights in different temporal layers are the same, the gradient is multiplied with the same quantity $w_t = w$ for each layer. Such a multiplication will have a consistent bias towards vanishing when $w < 1$, and it will have a consistent bias towards exploding when $w > 1$. However, the choice of the activation function will also play a role because the derivative $\Phi'(h_{t+1})$ is included in the product. For example, the presence of the tanh

activation function, for which the derivative $\Phi'(\cdot)$ is almost always less than 1, tends to increase the chances of the vanishing gradient problem.

Although the above discussion only studies the simple case of a hidden layer with one unit, one can generalize the argument to a hidden layer with multiple units [220]. In such a case, it can be shown that the update to the gradient boils down to a repeated multiplication with the same matrix A. One can show the following result:

Lemma 7.3.1 *Let A be a square matrix, the* **magnitude** *of whose largest eigenvalue is λ. Then, the entries of A^t tend to 0 with increasing values of t, when we have $\lambda < 1$. On the other hand, the entries of A^t diverge to large values, when we have $\lambda > 1$.*

The proof of the above result is easy to show by diagonalizing $A = P\Delta P^{-1}$. Then, it can be shown that $A^t = P\Delta^t P^{-1}$, where Δ is a diagonal matrix. The magnitude of the largest diagonal entry of Δ^t either vanishes with increasing t or it grows to an increasingly large value (in absolute magnitude) depending on whether than eigenvalue is less than 1 or larger than 1. In the former case, the matrix A^t tends to 0, and therefore the gradient vanishes. In the latter case, the gradient explodes. Of course, this does not yet include the effect of the activation function, and one can change the threshold on the largest eigenvalue to set up the conditions for the vanishing or exploding gradients. For example, the largest possible value of the sigmoid activation derivative is 0.25, and therefore the vanishing gradient problem will definitely occur when the largest eigenvalue is less that $1/0.25 = 4$. One can, of course, combine the effect of the matrix multiplication and activation function into a single *Jacobian* matrix (cf. Table 3.1 of Chapter 3), whose eigenvalues can be tested.

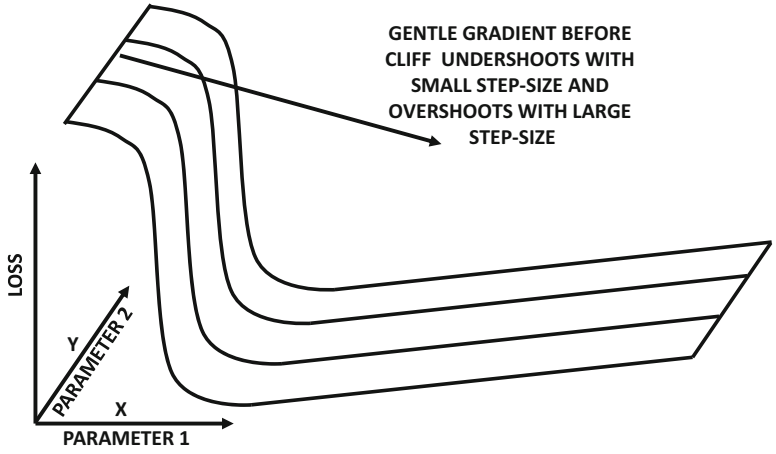


Figure 7.8: Revisiting Figure 3.13 of Chapter 3: An example of a cliff in the loss surface

In the particular case of recurrent neural networks, the *combination of the vanishing/exploding gradient and the parameter tying across different layers causes the recurrent neural network to behave in an unstable way with gradient-descent step size.* In other words, if we choose a step size that is too small, then the effect of some of the layers will cause little progress. On the other hand, if we choose a step size that is too large, then the effect of some of the layers will cause the step to overshoot the optimal point in an unstable way. An important issue here is that the gradient only tells us the best direction of movement for infinitesimally small steps; for finite steps, the behavior of the update could be substantially different from what is predicted by the gradient. The optimal points in recurrent networks

are often hidden near cliffs or other regions of unpredictable change in the topography of the loss function, which causes the best directions of *instantaneous* movement to be extremely poor predictors of the best directions of *finite* movement. Since any practical learning algorithm is required to make finite steps of reasonable sizes to make good progress towards the optimal solution, this makes training rather hard. An example of a cliff is illustrated in Figure 7.8. The challenges associated with cliffs are discussed in Section 3.5.4 of Chapter 3. A detailed discussion of the exploding gradient problem and its geometric interpretation may be found in [369].

There are several solutions to the vanishing and exploding gradient problems, not all of which are equally effective. For example, the simplest solution is to use strong regularization on the parameters, which tends to reduce some of the problematic instability caused by the vanishing and exploding gradient problems. However, very strong levels of regularization can lead to models that do not achieve the full potential of a particular architecture of the neural network. A second solution that is discussed in Section 3.5.5 of Chapter 3 is *gradient clipping*. Gradient clipping is well suited to solving the exploding gradient problem. There are two types of clipping that are commonly used. The first is value-based clipping, and the second is norm-based clipping. In value-based clipping, the largest temporal components of the gradient are clipped before adding them. This was the original form of clipping that was proposed by Mikolov in his Ph.D. thesis [324]. The second type of clipping is norm-based clipping. The idea is that when the entire gradient vector has a norm that increases beyond a particular threshold, it is re-scaled back to the threshold. Both types of clipping perform in a similar way, and an analysis is provided in [368].

One observation about suddenly changing curvatures (like cliffs) is that first-order gradients are generally inadequate to fully model local error surfaces. Therefore, a natural solution is to use higher-order gradients. The main challenge with higher-order gradients is that they are computationally expensive. For example, the use of second-order methods (cf. Section 3.5.6 of Chapter 3) requires the inversion of a Hessian matrix. For a network with 10^6 parameters, this would require the inversion of a $10^6 \times 10^6$ matrix. As a practical matter, this is impossible to do with the computational power available today. However, some clever tricks for implementing second-order methods with Hessian-free methods have been proposed recently [313, 314]. The basic idea is to never compute the Hessian matrix exactly, but always work with rough approximations. A brief overview of many of these methods is provided in Section 3.5.6 of Chapter 3. These methods have also met with some success in training recurrent neural networks.

The type of instability faced by the optimization process is sensitive to the specific point on the loss surface at which the current solution resides. Therefore, choosing good initialization points is crucial. The work in [140] discusses several types of initialization that can avoid instability in the gradient updates. Using momentum methods (cf. Chapter 3) can also help in addressing some of the instability. A discussion of the power of initialization and momentum in addressing some of these issues is provided in [478]. Often simplified variants of recurrent neural networks, like *echo-state networks*, are used for creating a robust initialization of recurrent neural networks.

Another useful trick that is often used to address the vanishing and exploding gradient problems is that of batch normalization, although the basic approach requires some modifications for recurrent networks [81]. Batch normalization methods are discussed in Section 3.6 of Chapter 3. However, a variant known as *layer normalization* is more effective in recurrent networks. Layer normalization methods have been so successful that they have become a standard option while using a recurrent neural network or its variants.

Finally, a number of variants of recurrent neural networks are used to address the vanishing and exploding gradient problems. The first simplification is the use of echo-state networks in which the hidden-to-hidden matrices are randomly chosen, but only the output layers are trained. In the early years, echo-state networks were used as viable alternatives to recurrent neural networks, when it was considered too hard to train recurrent neural networks. However, these methods are too simplified to used in very complex settings. Nevertheless, these methods can still be used for robust initialization in recurrent neural networks [478]. A more effective approach for dealing with the vanishing and exploding gradient problems is to arm the recurrent network with internal memory, which lends more stability to the states of the network. The use of long short-term memory (LSTM) has become an effective way of handing the vanishing and exploding gradient problems. This approach introduces some additional states, which can be interpreted as a kind of long-term memory. The long-term memory provides states that are more stable over time, and also provide a greater level of stability to the gradient-descent process. This approach is discussed in Section 7.5.

7.3.1 Layer Normalization

The batch normalization technique discussed in Section 3.6 of Chapter 3 is designed to address the vanishing and exploding gradient problems in deep neural networks. In spite of its usefulness in most types of neural networks, the approach faces some challenges in recurrent neural networks. First, the batch statistics vary with the time-layer of the neural network, and therefore different statistics need to be maintained for different time-stamps. Furthermore, the number of layers in a recurrent network is *input-dependent*, depending on the length of the input sequence. Therefore, if a test sequence is longer than any of the training sequences encountered in the data, mini-batch statistics may not be available for some of the time-stamps. In general, the computation of the mini-batch statistics is not equally reliable for different time-layers (irrespective of mini-batch size). Finally, batch normalization cannot be applied to online learning tasks. One of the problematic issues is that batch normalization is a relatively unconventional neural network operation (compared to traditional neural networks) because the activations of the units depend on other training instances in the batch, and not just the current instance. Although batch-normalization can be adapted to recurrent networks [81], a more effective approach is *layer normalization*.

In layer normalization, the normalization is performed only over a single training instance, although the normalization factor is obtained by using all the current activations *in that layer of only the current instance*. This approach is closer to a conventional neural network operation, and we no longer have the problem of maintaining mini-batch statistics. All the information needed to compute the activations for an instance can be obtained from that instance only!

In order to understand how layer-wise normalization works, we repeat the hidden-to-hidden recursion of page 276:

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1})$$

This recursion is prone to unstable behavior because of the multiplicative effect across time-layers. We will show how to modify this recurrence with layer-wise normalization. As in the case of conventional batch normalization of Chapter 3, the normalization is applied to *pre-activation* values before applying the tanh activation function. Therefore, the pre-activation value at the tth time-stamp is computed as follows:

$$\overline{a}_t = W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1}$$

Note that $\overline{a}_t$ is a vector with as many components as the number of units in the hidden layer (which we have consistently denoted as p in this chapter). We compute the mean μ_t and standard σ_t of the pre-activation values in $\overline{a}_t$:

$$\mu_t = \frac{\sum_{i=1}^{p} a_{ti}}{p}, \qquad \sigma_t = \sqrt{\frac{\sum_{i=1}^{p} a_{ti}^2}{p} - \mu_t^2}$$

Here, a_{ti} denotes the ith component of the vector $\overline{a}_t$.

As in batch normalization, we have additional learning parameters, associated with each unit. Specifically, for the p units in the tth layer, we have a p-dimensional vector of *gain parameters* $\overline{\gamma}_t$, and a p-dimensional vector of *bias parameters* denoted by $\overline{\beta}_t$. These parameters are analogous to the parameters γ_i and β_i in Section 3.6 on batch normalization. The purpose of these parameters is to re-scale the normalized values and add bias in a learnable way. The hidden activations $\overline{h}_t$ of the next layer are therefore computed as follows:

$$\overline{h}_t = \tanh\left(\frac{\overline{\gamma}_t}{\sigma_t} \odot (\overline{a}_t - \overline{\mu}_t) + \overline{\beta}_t\right) \tag{7.6}$$

Here, the notation $\odot$ indicates elementwise multiplication, and the notation $\overline{\mu}_t$ refers to a vector containing p copies of the scalar μ_t. The effect of layer normalization is to ensure that the magnitudes of the activations do not continuously increase or decrease with time-stamp (causing vanishing and exploding gradients), although the learnable parameters allow some flexibility. It has been shown in [14] that layer normalization provides better performance than batch normalization in recurrent neural networks. Some related normalizations can also be used for streaming and online learning [294].

7.4 Echo-State Networks

Echo-state networks represent a simplification of recurrent neural networks. They work well when the dimensionality of the input is small; this is because echo-state networks scale well with the number of temporal units but not with the dimensionality of the input. Therefore, these networks would be a solid option for regression-based modeling of a single or small number of real-valued time series over a relatively long time horizon. However, they would be a poor choice for modeling text in which the input dimensionality (based on one-hot encoding) would be the size of the lexicon in which the documents are represented. Nevertheless, even in this case, echo-state networks are practically useful in the initialization of weights within the network. Echo-state networks are also referred to as *liquid-state machines* [304], except that the latter uses spiking neurons with binary outputs, whereas echo-state networks use conventional activations like the sigmoid and the tanh functions.

Echo-state networks use *random weights* in the hidden-to-hidden layer and even the input-to-hidden layer, although the dimensionality of the hidden states is almost always much larger than the dimensionality of input states. For a single input series, it is not uncommon to use hidden states of dimensionality about 200. Therefore, only the output layer is trained, which is typically done with a linear layer for real-valued outputs. Note that the training of the output layer simply aggregates the errors at different output nodes, although the weights at different output nodes are still shared. Nevertheless, the objective function would still evaluate to a case of linear regression, which can be trained very simply without the need for backpropagation. Therefore, the training of the echo-state network is very fast.

As in traditional recurrent networks, the hidden-to-hidden layers have nonlinear activations such as the logistic sigmoid function, although tanh activations are also possible. A very important caveat in the initialization of the hidden-to-hidden units is that the largest eigenvector of the weight matrix W_{hh} should be set to 1. This can be easily achieved by first sampling the weights of the matrix W_{hh} randomly from a standard normal distribution, and then dividing each entry by the largest absolute eigenvalue $|\lambda_{max}|$ of this matrix.

$$W_{hh} \Leftarrow W_{hh}/|\lambda_{max}| \tag{7.7}$$

After this normalization, the largest eigenvalue of this matrix will be 1, which corresponds to its *spectral radius*. However, using a spectral radius of 1 can be too conservative because the nonlinear activations will have a dampening effect on the values of the states. For example, when using the sigmoid activation, the *largest* possible partial derivative of the sigmoid is always 0.25, and therefore using a spectral radius much larger than 4 (say, 10) is okay. When using the tanh activation function it would make sense to have a spectral radius of about 2 or 3. These choices would often still lead to a certain level of dampening over time, which is actually a useful regularization because very long-term relationships are generally much weaker than short-term relationships in time-series. One can also tune the spectral radius based on performance by trying different values of the scaling factor γ on held-out data to set $W_{hh} = \gamma W_0$. Here, W_0 is a randomly initialized matrix.

It is recommended to use sparse connectivity in the hidden-to-hidden connections, which is not uncommon in settings involving transformations with random projections. In order to achieve this goal, a number of connections in W_{hh} can be sampled to be non-zero and others are set to 0. This number of connections is typically linear in the number of hidden units. Another key trick is to divide the hidden units into groups indexed $1 \ldots K$ and only allow connectivity between hidden states belonging to with the same index. Such an approach can be shown to be equivalent to training an ensemble of echo-state networks (see. Exercise 2).

Another issue is about setting the input-to-hidden matrices W_{xh}. One needs to be careful about the scaling of this matrix as well, or else the effect of the inputs in each time-stamp can seriously damage the information carried in the hidden states from the previous time-stamp. Therefore, the matrix W_{xh} is first chosen randomly to W_1, and then it is scaled with different values of the hyper-parameter β in order to determine the final matrix $W_{xh} = \beta W_1$ that gives the best accuracy on held-out data.

The core of the echo-state network is based on a very old idea that expanding the number of features of a data set with a nonlinear transformation can often increase the expressive power of the input representation. For example, the RBF network (cf. Chapter 5) and the kernel support-vector machine both gain their power from expansion of the underlying feature space according to Cover's theorem on separability of patterns [84]. The only difference is that the echo-state network performs the feature expansion with random projection; such an approach is not without precedent because various types of random transformations are also used in machine learning as fast alternatives to kernel methods [385, 516]. It is noteworthy that feature expansion is primarily effective through nonlinear transformations, and these are provided through the activations in the hidden layers. In a sense, the echo-state method works using a similar principle to the RBF network in the temporal domain, just as the recurrent neural network is the replacement of feed-forward networks in the temporal domain. Just as the RBF network uses very little training for extracting the hidden features, the echo-state network uses little training for extracting the hidden features and instead relies on the randomized expansion of the feature space.

When used on time-series data, the approach provides excellent results on predicting values far out in the future. The key trick is to choose target output values at a time-stamp

t that correspond to the time-series input values at $t+k$, where k is the lookahead required for forecasting. In other words, an echo-state network is an excellent nonlinear autoregressive technique for modeling time-series data. One can even use this approach for forecasting multivariate time-series, although it is inadvisable to use the approach when the number of time series is very large. This is because the dimensionality of hidden states required for modeling would be simply too large. A detailed discussion on the application of the echo-state network for time-series modeling is provided in Section 7.7.5. A comparison with respect to traditional time-series forecasting models is also provided in the same section.

Although the approach cannot be realistically used for very high-dimensional inputs (like text), it is still very useful for initialization [478]. The basic idea is to initialize the recurrent network by using its echo-state variant to train the output layer. Furthermore, a proper scaling of the initialized values W_{hh} and W_{xh} can be set by trying different values of the scaling factors β and γ (as discussed above). Subsequently, traditional backpropagation is used to train the recurrent network. This approach can be viewed as a lightweight pretraining for recurrent networks.

A final issue is about the sparsity of the weight connections. Should the matrix W_{hh} be sparse? This is generally a matter of some controversy and disagreement; while sparse connectivity of echo-state networks has been recommended since the early years [219], the reasons for doing so are not very clear. The original work [219] states that sparse connectivity leads to a decoupling of the individual subnetworks, which encourages the development of individual dynamics. This seems to be an argument for increased diversity of the features learned by the echo-state network. If decoupling is indeed the goal, it would make a lot more sense to do so explicitly, and divide the hidden states into disconnected groups. Such an approach has an ensemble-centric interpretation. It is also often recommended to increase sparsity in methods involving random projections for improved efficiency of the computations. Having dense connections can cause the activations of different states to be embedded in the multiplicative noise of a large number of Gaussian random variables, and therefore more difficult to extract.

7.5 Long Short-Term Memory (LSTM)

As discussed in Section 7.3, recurrent neural networks have problems associated with vanishing and exploding gradients [205, 368, 369]. This is a common problem in neural network updates where successive multiplication by the matrix $W^{(k)}$ is inherently unstable; it either results in the gradient disappearing during backpropagation, or in it blowing up to large values in an unstable way. This type of instability is the direct result of successive multiplication with the (recurrent) weight matrix at various time-stamps. One way of viewing this problem is that a neural network that uses only multiplicative updates is good only at learning over short sequences, and is therefore inherently endowed with good short-term memory but poor long-term memory [205]. To address this problem, a solution is to change the recurrence equation for the hidden vector with the use of the LSTM with the use of long-term memory. The operations of the LSTM are designed to have fine-grained control over the data written into this long-term memory.

As in the previous sections, the notation $\overline{h}_t^{(k)}$ represents the hidden states of the kth layer of a multi-layer LSTM. For notational convenience, we also assume that the input layer $\overline{x}_t$ can be denoted by $\overline{h}_t^{(0)}$ (although this layer is obviously not hidden). As in the case of the recurrent network, the input vector $\overline{x}_t$ is d-dimensional, whereas the hidden states are p-dimensional. The LSTM is an enhancement of the recurrent neural network architecture

of Figure 7.6 in which we change the recurrence conditions of how the hidden states $\overline{h}_t^{(k)}$ are propagated. In order to achieve this goal, we have an additional hidden vector of p dimensions, which is denoted by $\overline{c}_t^{(k)}$ and referred to as the *cell state*. One can view the cell state as a kind of long-term memory that retains at least a part of the information in earlier states by using a combination of partial "forgetting" and "increment" operations on the previous cell states. It has been shown in [233] that the nature of the memory in $\overline{c}_t^{(k)}$ is occasionally interpretable when it is applied to text data such as literary pieces. For example, one of the p values in $\overline{c}_t^{(k)}$ might change in sign after an opening quotation and then revert back only when that quotation is closed. The upshot of this phenomenon is that the resulting neural network is able to model long-range dependencies in the language or even a specific pattern (like a quotation) extended over a large number of tokens. This is achieved by using a gentle approach to update these cell states over time, so that there is greater persistence in information storage. Persistence in state values avoids the kind of instability that occurs in the case of the vanishing and exploding gradient problems. One way of understanding this intuitively is that if the states in different temporal layers share a greater level of similarity (through long-term memory), it is harder for the gradients with respect to the incoming weights to be drastically different.

As with the multilayer recurrent network, the update matrix is denoted by $W^{(k)}$ and is used to premultiply the column vector $[\overline{h}_t^{(k-1)}, \overline{h}_{t-1}^{(k)}]^T$. However, this matrix is of size[2] $4p \times 2p$, and therefore pre-multiplying a vector of size $2p$ with $W^{(k)}$ results in a vector of size $4p$. In this case, the updates use four intermediate, p-dimensional vector variables $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$ that correspond to the $4p$-dimensional vector. The intermediate variables $\overline{i}$, $\overline{f}$, and $\overline{o}$ are respectively referred to as *input*, *forget*, and *output* variables, because of the roles they play in updating the cell states and hidden states. The determination of the hidden state vector $\overline{h}_t^{(k)}$ and the cell state vector $\overline{c}_t^{(k)}$ uses a multi-step process of first computing these intermediate variables and then computing the hidden variables from these intermediate variables. Note the difference between intermediate variable vector $\overline{c}$ and primary cell state $\overline{c}_t^{(k)}$, which have completely different roles. The updates are as follows:

$$\begin{array}{l}\text{Input Gate:}\\ \text{Forget Gate:}\\ \text{Output Gate:}\\ \text{New C.-State:}\end{array} \begin{bmatrix} \overline{i} \\ \overline{f} \\ \overline{o} \\ \overline{c} \end{bmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^{(k)} \begin{bmatrix} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{bmatrix} \textbf{[Setting up intermediates]}$$

$$\overline{c}_t^{(k)} = \overline{f} \odot \overline{c}_{t-1}^{(k)} + \overline{i} \odot \overline{c} \textbf{ [Selectively forget and add to long-term memory]}$$

$$\overline{h}_t^{(k)} = \overline{o} \odot \tanh(\overline{c}_t^{(k)}) \textbf{ [Selectively leak long-term memory to hidden state]}$$

[2] In the first layer, the matrix $W^{(1)}$ is of size $4p \times (p + d)$ because it is multiplied with a vector of size $(p + d)$.

Here, the element-wise product of vectors is denoted by "$\odot$," and the notation "sigm" denotes a sigmoid operation. For the very first layer (i.e., $k = 1$), the notation $\overline{h}_t^{(k-1)}$ in the above equation should be replaced with $\overline{x}_t$ and the matrix $W^{(1)}$ is of size $4p \times (p + d)$. In practical implementations, biases are also used[3] in the above updates, although they are omitted here for simplicity. The aforementioned update seems rather cryptic, and therefore it requires further explanation.

The first step in the above sequence of equations is to set up the intermediate variable vectors $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$, of which the first three should *conceptually* be considered binary values, although they are continuous values in $(0, 1)$. Multiplying a pair of binary values is like using an AND gate on a pair of boolean values. We will henceforth refer to this operation as gating. The vectors $\overline{i}$, $\overline{f}$, and $\overline{o}$ are referred to as input, forget, and output gates. In particular, these vectors are conceptually used as boolean gates for deciding (i) whether to add to a cell-state, (ii) whether to forget a cell state, and (iii) whether to allow leakage into a hidden state from a cell state. The use of the binary abstraction for the input, forget, and output variables helps in understanding the types of decisions being made by the updates. In practice, a continuous value in $(0, 1)$ is contained in these variables, which can enforce the effect of the binary gate in a probabilistic way if the output is seen as a probability. In the neural network setting, it is essential to work with continuous functions in order to ensure the differentiability required for gradient updates. The vector $\overline{c}$ contains the newly proposed contents of the cell state, although the input and forget gates regulate how much it is allowed to change the previous cell state (to retain long-term memory).

The four intermediate variables $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$, are set up using the weight matrices $W^{(k)}$ for the kth layer in the first equation above. Let us now examine the second equation that updates the cell state with the use of some of these intermediate variables:

$$\overline{c}_t^{(k)} = \underbrace{\overline{f} \odot \overline{c}_{t-1}^{(k)}}_{\text{Reset?}} + \underbrace{\overline{i} \odot \overline{c}}_{\text{Increment?}}$$

This equation has two parts. The first part uses the p forget bits in $\overline{f}$ to decide which of the p cell states from the previous time-stamp to reset[4] to 0, and it uses the p input bits in $\overline{i}$ to decide whether to add the corresponding components from $\overline{c}$ to each of the cell states. Note that such updates of the cell states are in additive form, which is helpful in avoiding the vanishing gradient problem caused by multiplicative updates. One can view the cell-state vector as a continuously updated long-term memory, where the forget and input bits respectively decide (i) whether to reset the cell states from the previous time-stamp and forget the past, and (ii) whether to increment the cell states from the previous time-stamp to incorporate new information into long-term memory from the current word. The vector $\overline{c}$ contains the p amounts with which to increment the cell states, and these are values in $[-1, +1]$ because they are all outputs of the tanh function.

[3] The bias associated with the forget gates is particularly important. The bias of the forget gate is generally initialized to values greater than 1 [228] because it seems to avoid the vanishing gradient problem at initialization.

[4] Here, we are treating the forget bits as a vector of binary bits, although it contains continuous values in $(0, 1)$, which can be viewed as probabilities. As discussed earlier, the binary abstraction helps us understand the conceptual nature of the operations.

Finally, the hidden states $\overline{h}_t^{(k)}$ are updated using leakages from the cell state. The hidden state is updated as follows:

$$\overline{h}_t^{(k)} = \underbrace{\overline{o} \odot \tanh(\overline{c}_t^{(k)})}_{\text{Leak } \overline{c}_t^{(k)} \text{ to } \overline{h}_t^{(k)}}$$

Here, we are copying a functional form of each of the p cell states into each of the p hidden states, depending on whether the output gate (defined by $\overline{o}$) is 0 or 1. Of course, in the continuous setting of neural networks, partial gating occurs and only a fraction of the signal is copied from each cell state to the corresponding hidden state. It is noteworthy that the final equation does not always use the tanh activation function. The following alternative update may be used:

$$\overline{h}_t^{(k)} = \overline{o} \odot \overline{c}_t^{(k)}$$

As in the case of all neural networks, the backpropagation algorithm is used for training purposes.

In order to understand why LSTMs provide better gradient flows than vanilla RNNs, let us examine the update for a simple LSTM with a single layer and $p = 1$. In such a case, the cell update can be simplified to the following:

$$c_t = c_{t-1} * f + i * c \tag{7.8}$$

Therefore, the partial derivative c_t with respect to c_{t-1} is f, which means that the backward gradient flows for c_t are multiplied with the value of the forget gate f. Because of elementwise operations, this result generalizes to arbitrary values of the state dimensionality p. The biases of the forget gates are often set to high values initially, so that the gradient flows decay relatively slowly. The forget gate f can also be different at different time-stamps, which reduces the propensity of the vanishing gradient problem. The hidden states can be expressed in terms of the cell states as $h_t = o * \tanh(c_t)$, so that one can compute the partial derivative with respect to h_t with the use of a single tanh derivative. In other words, the long-term cell states function as gradient super-highways, which leak into hidden states.

7.6 Gated Recurrent Units (GRUs)

The Gated Recurrent Unit (GRU) can be viewed as a simplification of the LSTM, which does not use explicit cell states. Another difference is that the LSTM directly controls the amount of information changed in the hidden state using separate forget and output gates. On the other hand, a GRU uses a single reset gate to achieve the same goal. However, the basic idea in the GRU is quite similar to that of an LSTM, in terms of how it partially resets the hidden states. As in the previous sections, the notation $\overline{h}_t^{(k)}$ represents the hidden states of the kth layer for $k \geq 1$. For notational convenience, we also assume that the input layer $\overline{x}_t$ can be denoted by $\overline{h}_t^{(0)}$ (although this layer is obviously not hidden). As in the case of LSTM, we assume that the input vector $\overline{x}_t$ is d-dimensional, whereas the hidden states are p-dimensional. The sizes of the transformation matrices in the first layer are accordingly adjusted to account for this fact.

In the case of the GRU, we use two matrices $W^{(k)}$ and $V^{(k)}$ of sizes[5] $2p \times 2p$ and $p \times 2p$, respectively. Pre-multiplying a vector of size $2p$ with $W^{(k)}$ results in a vector of size $2p$, which will be passed through the sigmoid activation to create two intermediate, p-dimensional vector variables $\overline{z}_t$ and $\overline{r}_t$, respectively. The intermediate variables $\overline{z}_t$ and $\overline{r}_t$ are respectively referred to as update and reset gates. The determination of the hidden state vector $\overline{h}_t^{(k)}$ uses a two-step process of first computing these gates, then using them to decide how much to change the hidden vector with the weight matrix $V^{(k)}$:

$$\begin{array}{l}\text{Update Gate:}\\ \text{Reset Gate:}\end{array} \left[\begin{array}{c}\overline{z}\\ \overline{r}\end{array}\right] = \left(\begin{array}{c}\text{sigm}\\ \text{sigm}\end{array}\right) W^{(k)} \left[\begin{array}{c}\overline{h}_t^{(k-1)}\\ \overline{h}_{t-1}^{(k)}\end{array}\right] \quad \textbf{[Set up gates]}$$

$$\overline{h}_t^{(k)} = \overline{z} \odot \overline{h}_{t-1}^{(k)} + (1 - \overline{z}) \odot \tanh\, V^{(k)} \left[\begin{array}{c}\overline{h}_t^{(k-1)}\\ \overline{r} \odot \overline{h}_{t-1}^{(k)}\end{array}\right] \quad \textbf{[Update hidden state]}$$

Here, the element-wise product of vectors is denoted by "$\odot$," and the notation "sigm" denotes a sigmoid operation. For the very first layer (i.e., $k = 1$), the notation $\overline{h}_t^{(k-1)}$ in the above equation should be replaced with $\overline{x}_t$. Furthermore, the matrices $W^{(1)}$ and $V^{(1)}$ are of sizes $2p \times (p + d)$ and $p \times (p + d)$, respectively. We have also omitted the mention of biases here, but they are usually included in practical implementations. In the following, we provide a further explanation of these updates and contrast them with those of the LSTM.

Just as the LSTM uses input, output, and forget gates to decide how much of the information from the previous time-stamp to carry over to the next step, the GRU uses the update and the reset gates. The GRU does not have a separate internal memory and also requires fewer gates to perform the update from one hidden state to another. Therefore, a natural question arises about the precise role of the update and reset gates. The reset gate $\overline{r}$ decides how much of the hidden state to carry over from the previous time-stamp for a matrix-based update (like a recurrent neural network). The update gate $\overline{z}$ decides the *relative* strength of the contributions of this matrix-based update and a more direct contribution from the hidden vector $\overline{h}_{t-1}^{(k)}$ at the previous time-stamp. By allowing a direct (partial) copy of the hidden states from the previous layer, the gradient flow becomes more stable during backpropagation. The update gate of the GRU simultaneously performs the role of the input and forget gates in the LSTM in the form of $\overline{z}$ and $1 - \overline{z}$, respectively. However, the mapping between the GRU and the LSTM is not precise, because it performs these updates directly on the hidden state (and there is no cell state). Like the input, output, and forget gates in the LSTM, the update and reset gates are intermediate "scratch-pad" variables.

In order to understand why GRUs provide better performance than vanilla RNNs, let us examine a GRU with a single layer and single state dimensionality $p = 1$. In such a case, the update equation of the GRU can be written as follows:

$$h_t = z \cdot h_{t-1} + (1 - z) \cdot \tanh[v_1 \cdot x_t + v_2 \cdot r \cdot h_{t-1}] \tag{7.9}$$

Note that layer superscripts are missing in this single-layer case. Here, v_1 and v_2 are the two elements of the 2×1 matrix V. Then, it is easy to see the following:

$$\frac{\partial h_t}{\partial h_{t-1}} = z + (\text{Additive Terms}) \tag{7.10}$$

[5]In the first layer ($k = 1$), these matrices are of sizes $2p \times (p + d)$ and $p \times (p + d)$.

Backward gradient flow is multiplied with this factor. Here, the term $z \in (0, 1)$ helps in passing *unimpeded* gradient flow and makes computations more stable. Furthermore, since the additive terms heavily depend on $(1-z)$, the overall multiplicative factor that tends to be closer to 1 even when z is small. Another point is that the value of z and the multiplicative factor $\frac{\partial h_t}{\partial h_{t-1}}$ is *different* for each time stamp, which tends to reduce the propensity for vanishing or exploding gradients.

Although the GRU is a closely related simplification of the LSTM, it should not be seen as a special case of the LSTM. A comparison of the LSTM and the GRU is provided in [71, 228]. The two models are shown to be roughly similar in performance, and the relative performance seems to depend on the task at hand. The GRU is simpler and enjoys the advantage of greater ease of implementation and efficiency. It might generalize slightly better with less data because of a smaller parameter footprint [71], although the LSTM would be preferable with an increased amount of data. The work in [228] also discusses several practical implementation issues associated with the LSTM. The LSTM has been more extensively tested than the GRU, simply because it is an older architecture and enjoys widespread popularity. As a result, it is generally seen as a safer option, particularly when working with longer sequences and larger data sets. The work in [160] also showed that none of the variants of the LSTM can reliably outperform it in a consistent way. This is because of the explicit internal memory and the greater gate-centric control in updating the LSTM.

7.7 Applications of Recurrent Neural Networks

Recurrent neural networks have numerous applications in machine learning applications, which are associated with information retrieval, speech recognition, and handwriting recognition. Text data forms the predominant setting for applications of RNNs, although there are several applications to computational biology as well. Most of the applications of RNNs fall into one of two categories:

1. *Conditional language modeling:* When the output of a recurrent network is a language model, one can enhance it with context in order to provide a relevant output to the context. In most of these cases, the context is the neural output of another neural network. To provide one example, in image captioning the context is the neural representation of an image provided by a convolutional network, and the language model provides a caption for the image. In machine translation, the context is the representation of a sentence in a source language (produced by another RNN), and the language model in the target language provides a translation.

2. *Leveraging token-specific outputs:* The outputs at the different tokens can be used to learn other properties than a language model. For example, the labels output at different time-stamps might correspond to the properties of the tokens (such as their parts of speech). In handwriting recognition, the labels might correspond to the characters. In some cases, all the time-stamps might not have an output, but the end-of-sentence marker might output a label for the entire sentence. This approach is referred to as sentence-level classification, and is often used in sentiment analysis. In some of these applications, bidirectional recurrent networks are used because the context on both sides of a word is helpful.

The following material will provide an overview of the numerous applications of recurrent neural networks. In most of these cases, we will use a single-layer recurrent network for ease in explanation and pictorial illustration. However, in most cases, a multi-layer LSTM is used. In other cases, a bidirectional LSTM is used, because it provides better performance. Replacing a single-layer RNN with a multi-layer/bidirectional LSTM in any of the following applications is straightforward. Out broader goal is to illustrate how this *family* of architectures can be used in these settings.

7.7.1 Application to Automatic Image Captioning

In image captioning, the training data consists of image-caption pairs. For example, the image[6] in the left-hand side of Figure 7.9 is obtained from the National Aeronautics and Space Administration Web site. This image is captioned "*cosmic winter wonderland*." One might have hundreds of thousands of such image-caption pairs. These pairs are used to train the weights in the neural network. Once the training has been completed, the captions are predicted for unknown test instances. Therefore, one can view this approach as an instance of image-to-sequence learning.

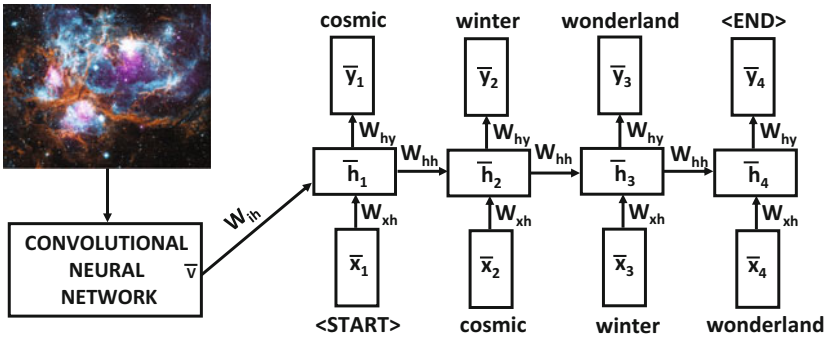


Figure 7.9: Example of image captioning with a recurrent neural network. An additional convolutional neural network is required for representational learning of the images. The image is represented by the vector $\overline{v}$, which is the output of the convolutional neural network. The inset image is by courtesy of the National Aeronautics and Space Administration (NASA).

One issue in the automatic captioning of images is that a separate neural network is required to learn the representation of the images. A common architecture to learn the representation of images is the *convolutional neural network*. A detailed discussion of convolutional neural networks is provide in Chapter 8. Consider a setting in which the convolutional neural network produces the q-dimensional vector $\overline{v}$ as the output representation. This vector is then used as an input to the neural network, but only[7] at the first time-stamp. To account for this additional input, we need another $p \times q$ matrix W_{ih}, which maps the image representation to the hidden layer. Therefore, the update equations for the various layers now need to be modified as follows:

$$\overline{h}_1 = \tanh(W_{xh}\overline{x}_1 + W_{ih}\overline{v})$$

[6] https://www.nasa.gov/mission_pages/chandra/cosmic-winter-wonderland.html

[7] In principle, one can also allow it to be input at all time-stamps, but it only seems to worsen performance.

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1}) \quad \forall t \geq 2$$
$$\overline{y}_t = W_{hy}\overline{h}_t$$

An important point here is that the convolutional neural network and the recurrent neural network are not trained in isolation. Although one might train them in isolation in order to create an initialization, the final weights are always trained jointly by running each image through the network and matching up the predicted caption with the true caption. In other words, for each image-caption pair, the weights in both networks are updated when errors are made in predicting any particular token of the caption. In practice, the errors are soft because the tokens at each point are predicted probabilistically. Such an approach ensures that the learned representation $\overline{v}$ of the images is sensitive to the specific application of predicting captions.

After all the weights have been trained, a test image is input to the entire system and passed through both the convolutional and recurrent neural network. For the recurrent network, the input at the first time-stamp is the <START> token and the representation of the image. At later time-stamps, the input is the most likely token predicted at the previous time-stamp. One can also use beam search to keep track of the b most likely sequence prefixes to expand on at each point. This approach is not very different from the language generation approach discussed in Section 7.2.1.1, except that it is conditioned on the image representation that is input to the model in the first time-stamp of the recurrent network. This results in the prediction of a relevant caption for the image.

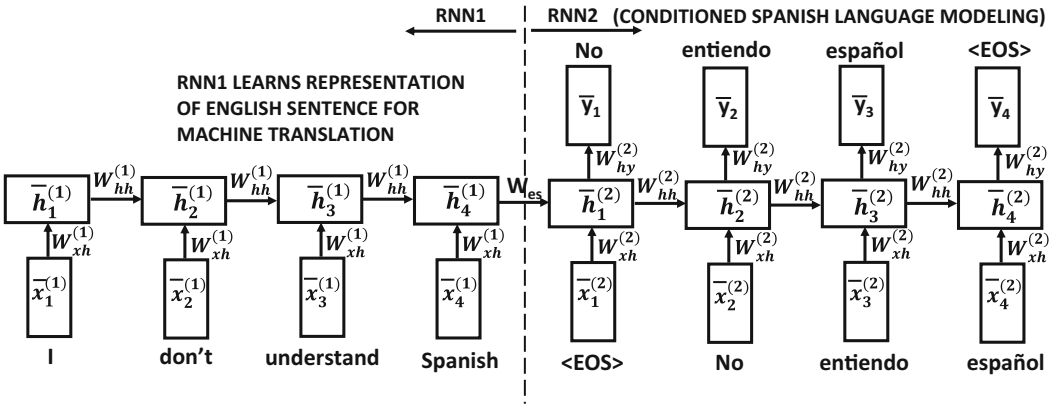


Figure 7.10: Machine translation with recurrent neural networks. Note that there are two separate recurrent networks with their own sets of shared weights. The output of $\overline{h}_4^{(1)}$ is a fixed length encoding of the 4-word English sentence.

7.7.2 Sequence-to-Sequence Learning and Machine Translation

Just as one can put together a convolutional neural network and a recurrent neural network to perform image captioning, one can put together two recurrent networks to translate one language into another. Such methods are also referred to as *sequence-to-sequence* learning because a sequence in one language is mapped to a sequence in another language. In principle, sequence-to-sequence learning can have applications beyond machine translation. For example, even question-answering (QA) systems can be viewed as sequence-to-sequence learning applications.

In the following, we provide a simple solution to machine translation with recurrent neural networks, although such applications are rarely addressed directly with the simple forms of recurrent neural networks. Rather, a variation of the recurrent neural network, referred to as the long short-term memory (LSTM) model is used. Such a model is much better in learning long-term dependencies, and can therefore work well with longer sentences. Since the general approach of using an RNN applies to an LSTM as well, we will provide the discussion of machine translation with the (simple) RNN. A discussion of the LSTM is provided in Section 7.5, and the generalization of the machine translation application to the LSTM is straightforward.

In the machine translation application, two different RNNs are hooked end-to-end, just as a convolutional neural network and a recurrent neural network are hooked together for image captioning. The first recurrent network uses the words from the source language as input. No outputs are produced at these time-stamps and the successive time-stamps accumulate knowledge about the source sentence in the hidden state. Subsequently, the end-of-sentence symbol is encountered, and the second recurrent network starts by outputting the first word of the target language. The next set of states in the second recurrent network output the words of the sentence in the target language one by one. These states also use the words of the target language as input, which is available for the case of the training instances but not for test instances (where predicted values are used instead). This architecture is shown in Figure 7.10.

The architecture of Figure 7.10 is similar to that of an autoencoder, and can even be used with pairs of identical sentences in the same language to create fixed-length representations of sentences. The two recurrent networks are denoted by RNN1 and RNN2, and their weights are not the same. For example, the weight matrix between two hidden nodes at successive time-stamps in RNN1 is denoted by $W_{hh}^{(1)}$, whereas the corresponding weight matrix in RNN2 is denoted by $W_{hh}^{(2)}$. The weight matrix W_{es} of the link joining the two neural networks is special, and can be independent of either of the two networks. This is necessary if the sizes of the hidden vectors in the two RNNs are different because the dimensions of the matrix W_{es} will be different from those of both $W_{hh}^{(1)}$ and $W_{hh}^{(2)}$. As a simplification, one can use[8] the same size of the hidden vector in both networks, and set $W_{es} = W_{hh}^{(1)}$. The weights in RNN1 are devoted to learning an encoding of the input in the source language, and the weights in RNN2 are devoted to using this encoding in order to create an output sentence in the target language. One can view this architecture in a similar way to the image captioning application, except that we are using two recurrent networks instead of a convolutional-recurrent pair. The output of the final hidden node of RNN1 is a fixed-length encoding of the source sentence. Therefore, irrespective of the length of the sentence, the encoding of the source sentence depends on the dimensionality of the hidden representation.

The grammar and length of the sentence in the source and target languages may not be the same. In order to provide a grammatically correct output in the target language, RNN2 needs to learn its language model. It is noteworthy that the units in RNN2 associated with the target language have both inputs and outputs arranged in the same way as a language-modeling RNN. At the same time, the output of RNN2 is conditioned on the input it receives from RNN1, which effectively causes language translation. In order to achieve this goal, training pairs in the source and target languages are used. The approach passes the source-target pairs through the architecture of Figure 7.10 and learns the model parameters

[8]The original work in [478] seems to use this option. In the Google Neural Machine Translation system [579], this weight is removed. This system is now used in Google Translate.

with the use of the backpropagation algorithm. Since only the nodes in RNN2 have outputs, only the errors made in predicting the target language words are backpropagated to train the weights in both neural networks. The two networks are jointly trained, and therefore the weights in both networks are optimized to the errors in the translated outputs of RNN2. As a practical matter, this means that the internal representation of the source language learned by RNN1 is highly optimized to the machine translation application, and is very different from one that would be learned if one had used RNN1 to perform language modeling of the source sentence. After the parameters have been learned, a sentence in the source language is translated by first running it through RNN1 to provide the necessary input to RNN2. Aside from this contextual input, another input to the first unit of RNN2 is the <EOS> tag, which causes RNN2 to output the likelihoods of the first token in the target language. The most likely token using beam search (cf. Section 7.2.1.1) is selected and used as the input to the recurrent network unit in the next time-stamp. This process is recursively applied until the output of a unit in RNN2 is also <EOS>. As in Section 7.2.1.1, we are generating a sentence from the target language using a language-modeling approach, except that the specific output is conditioned on the internal representation of the source sentence.

The use of neural networks for machine translation is relatively recent. Recurrent neural network models have a sophistication that greatly exceeds that of traditional machine translation models. The latter class of methods uses phrase-centric machine learning, which is often not sophisticated enough to learn the subtle differences between the grammars of the two languages. In practice, deep models with multiple layers are used to improve the performance.

One weakness of such translation models is that they tend to work poorly when the sentences are long. Numerous solutions have been proposed to solve the problem. A recent solution is that the sentence in the source language is input in the *opposite order* [478]. This approach brings the first few words of the sentences in the two languages closer in terms of their time-stamps within the recurrent neural network architecture. As a result, the first few words in the target language are more likely to be predicted correctly. The correctness in predicting the first few words is also helpful in predicting the subsequent words, which are also dependent on a neural language model in the target language.

7.7.2.1 Question-Answering Systems

A natural application of sequence-to-sequence learning is that of question answering (QA). Question-answering systems are designed with different types of training data. In particular, two types of question-answering systems are common:

1. In the first type, the answers are directly inferred based on the phrases and clue words in the question.

2. In the second type, the question is first transformed into a database query, and is used to query a structured knowledge base of facts.

Sequence-to-sequence learning can be helpful in both settings. Consider the first setting, in which we have training data containing question-answer pairs like the following:

What is the capital of China? <EOQ> The capital is Beijing. <EOA>

These types of training pairs are not very different from those available in the case of machine translation, and the same techniques can be used in these cases. However, note that one key difference between machine translation and question-answering systems is that

there is a greater level of reasoning in the latter, which typically requires an understanding of the relationships between various entities (e.g., people, places, and organizations). This problem is related to the quintessential problem of *information extraction*. Since questions are often crafted around various types of named entities and relationships among them, information extraction methods are used in various ways. The utility of entities and information extraction is well known in answering "what/who/where/when" types of questions (e.g., entity-oriented search), because *named entities* are used to represent persons, locations, organizations, dates, and events, and *relationship extraction* provides information about the interactions among them. One can incorporate the meta-attributes about tokens, such as entity types, as additional inputs to the learning process. Specific examples of such input units are shown in Figure 7.12 of Section 7.7.4, although the figure is designed for the different application of token-level classification.

An important difference between question-answering and machine translation systems is that the latter is seeded with a large corpus of documents (e.g., a large knowledge base like Wikipedia). The query resolution process can be viewed as a kind of entity-oriented search. From the perspective of deep learning, an important challenge of QA systems is that a much larger capacity to store the knowledge is required than is typically available in recurrent neural networks. A deep learning architecture that works well in these settings is that of *memory networks* [528]. Question-answering systems pose many different settings in which the training data may be presented, and the ways in which various types of questions may be answered and evaluated. In this context, the work in [527] discusses a number of template tasks that can be useful for evaluating question-answering systems.

A somewhat different approach is to convert natural language questions into queries that are properly posed in terms of entity-oriented search. Unlike machine translation systems, question answering is often considered a multi-stage process in which understanding what is being asked (in terms of a properly represented query) is sometimes more difficult than answering the query itself. In such cases, the training pairs will correspond to the informal and formal representations of questions. For example, one might have a pair as follows:

$$\underbrace{\text{What is the capital of China? <EOQ1>}}_{\text{Natural language question}} \quad \underbrace{\textbf{CapitalOf}(\ \textit{China}, ?)\ \text{<EOQ2>}}_{\text{Formal Representation}}$$

The expression on the right-hand side is a structured question, which queries for entities of different types such as persons, places, and organizations. The first step would be to convert the question into an internal representation like the one above, which is more prone to query answering. This conversion can be done using training pairs of questions and their internal representations in conjunction with an recurrent network. Once the question is understood as an entity-oriented search query, it can be posed to the indexed corpus, from which relevant relationships might already have been extracted up front. Therefore, the knowledge base is also preprocessed in such cases, and the question resolution boils down to matching the query with the extracted relations. It is noteworthy that this approach is limited by the complexity of the syntax in which questions are expressed, and the answers might also be simple one-word responses. Therefore, this type of approach is often used for more restricted domains. In some cases, one learns how to paraphrase questions by rewording a more complex question as a simpler question before creating the query representation [115, 118]:

$$\underbrace{\text{How can you tell if you have the flu? <EOQ1>}}_{\text{Complex question}} \quad \underbrace{\text{What are the signs of the flu? <EOQ2>}}_{\text{Paraphrased}}$$

The paraphrased question can be learned with sequence-to-sequence learning, although the work in [118] does not seem to use this approach. Subsequently, it is easier to convert the paraphrased question into a structured query. Another option is to provide the question in structured form to begin with. An example of a recurrent neural network that supports factoid question answering from QA training pairs is provided in [216]. However, unlike pure sequence-to-sequence learning, it uses the dependency parse trees of questions as the input representation. Therefore, a part of the formal understanding of the question is already encoded into the input.

7.7.3 Application to Sentence-Level Classification

In this problem, each sentence is treated as a training (or test) instance for classification purposes. Sentence-level classification is generally a more difficult problem than document-level classification because sentences are short, and there is often not enough evidence in the vector space representation to perform the classification accurately. However, the sequence-centric view is more powerful and can often be used to perform more accurate classification. The RNN architecture for sentence-level classification is shown in Figure 7.11. Note that the only difference from Figure 7.11(b) is that we no longer care about the outputs at each node but defer the class output to the end of the sentence. In other words, a single class label is predicted at the very last time-stamp of the sentence, and it is used to backpropagate the class prediction errors.

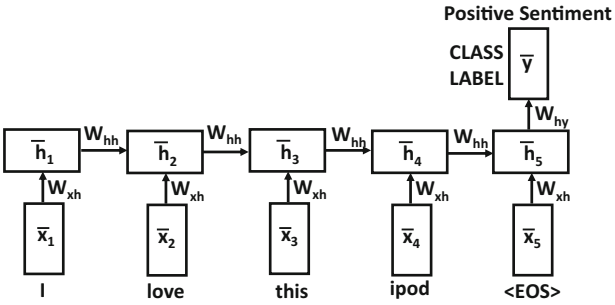


Figure 7.11: Example of sentence-level classification in a sentiment analysis application with the two classes "*positive sentiment*" and "*negative sentiment*."

Sentence-level classification is often leveraged in *sentiment analysis*. This problem attempts to discover how positive or negative users are about specific topics by analyzing the content of a sentence [6]. For example, one can use sentence-level classification to determine whether or not a sentence expresses a positive sentiment by treating the sentiment polarity as the class label. In the example shown in Figure 7.11, the sentence clearly indicates a positive sentiment. Note, however, that one cannot simply use a vector space representation containing the word "*love*" to infer the positive sentiment. For example, if words such as "*don't*" or "*hardly*" occur before "*love*", the sentiment would change from positive to negative. Such words are referred to as *contextual valence shifters* [377], and their effect can be modeled only in a sequence-centric setting. Recurrent neural networks can handle such settings because they use the accumulated evidence over the specific sequence of words in order to predict the class label. One can also combine this approach with linguistic features. In the next section, we show how to use linguistic features for token-level classification; similar ideas also apply to the case of sentence-level classification.

7.7.4 Token-Level Classification with Linguistic Features

The numerous applications of token-level classification include information extraction and text segmentation. In information extraction, specific words or combinations of words are identified that correspond to persons, places, or organizations. The linguistic features of the word (capitalization, part-of-speech, orthography) are more important in these applications than in typical language modeling or machine translation applications. Nevertheless, the methods discussed in this section for incorporating linguistic features can be used for any of the applications discussed in earlier sections. For the purpose of discussion, consider a *named-entity recognition application* in which every entity is to be classified as one of the categories corresponding to person (P), location (L), and other (O). In such cases, each token in the training data has one of these labels. An example of a possible training sentence is as follows:

$$\underbrace{\text{William}}_{P}\ \underbrace{\text{Jefferson}}_{P}\ \underbrace{\text{Clinton}}_{P}\ \underbrace{\text{lives}}_{O}\ \underbrace{\text{in}}_{O}\ \underbrace{\text{New}}_{L}\ \underbrace{\text{York}}_{L}.$$

In practice, the tagging scheme is often more complex because it encodes information about the beginning and end of a set of contiguous tokens with the same label. For test instances, the tagging information about the tokens is not available.

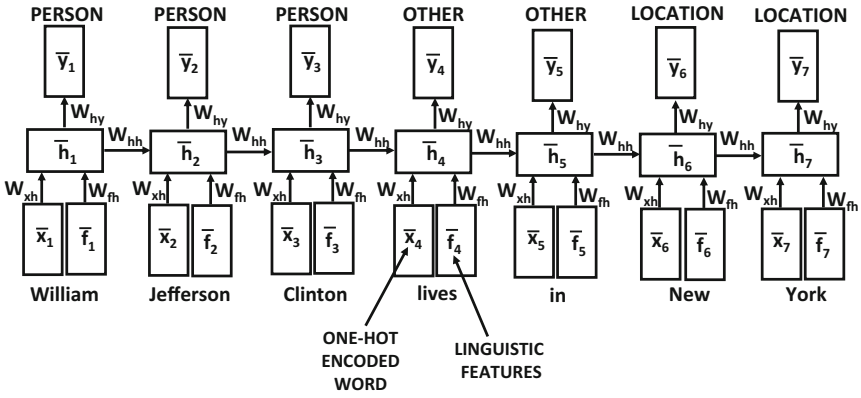


Figure 7.12: Token-wise classification with linguistic features

The recurrent neural network can be defined in a similar way as in the case of language modeling applications, except that the outputs are defined by the tags rather than the next set of words. The input at each time-stamp t is the one-hot encoding $\overline{x}_t$ of the token, and the output $\overline{y}_t$ is the tag. Furthermore, we have an additional set of q-dimensional linguistic features $\overline{f}_t$ associated with the tokens at time-stamp t. These linguistic features might encode information about the capitalization, orthography, capitalization, and so on. The hidden layer, therefore, receives two separate inputs from the tokens and from the linguistic features. The corresponding architecture is illustrated in Figure 7.12. We have an additional $p \times q$ matrix W_{fh} that maps the features $\overline{f}_t$ to the hidden layer. Then, the recurrence condition at each time-stamp t is as follows:

$$\overline{h}_t = \tanh(W_{xh}\overline{x}_t + W_{fh}\overline{f}_t + W_{hh}\overline{h}_{t-1})$$
$$\overline{y}_t = W_{hy}\overline{h}_t$$

The main innovation here is in the use of an additional weight matrix for the linguistic features. The change in the type of output tag does not affect the overall model significantly.

In some variations, it might also be helpful to *concatenate* the linguistic and token-wise features into as separate *embedding layer*, rather than adding them. The work in [565] provides an example in the case of recommender systems, although the principle can also be applied here. The overall learning process is also not significantly different. In token-level classification applications, it is sometimes helpful to use bidirectional recurrent networks in which recurrence occurs in both temporal directions [434].

7.7.5 Time-Series Forecasting and Prediction

Recurrent neural networks present a natural choice for time-series forecasting and prediction. The main difference from text is that the input units are real-valued vectors rather than (discrete) one-hot encoded vectors. For real-valued prediction, the output layer always uses linear activations, rather than the softmax function. In the event that the output is a discrete value (e.g., identifier of a specific event), it is also possible to use discrete outputs with softmax activation. Although any of the variants of the recurrent neural network (e.g., LSTM or GRU) can be used, one of the common problems in time-series analysis is that such sequences can be extremely long. Even though the LSTM and the GRU provide a certain level of protection with increased time-series length, there are limitations to the performance. This is because LSTMs and GRUs do degrade for series beyond certain lengths. Many time-series can have a very large number of time-stamps with various types of short- and long-term dependencies. The prediction and forecasting problems present unique challenges in these cases.

However, a number of useful solutions exist, at least in cases where the number of time-series to be forecasted is not too large. The most effective method is the use of the echo-state network (cf. Section 7.4), in which it is possible to effectively forecast and predict both real-valued and discrete observations with a *small* number of time-series. The caveat that the number of inputs is small is an important one, because echo-state networks rely on randomized expansion of the feature space via the hidden units (see Section 7.4). If the number of original time series is too large, then it may not turn out to be practical to expand the dimensionality of the hidden space sufficiently to capture this type of feature engineering. It is noteworthy that the vast majority of forecasting models in the time-series literature are, in fact, univariate models. A classical example is the *autoregressive model* (AR), which uses the immediate window of history in order to perform forecasting.

The use of an echo-state network in order to perform time-series regression and forecasting is straightforward. At each time-stamp, the input is a vector of d values corresponding to the d different time series that are being modeled. It is assumed that the d time series are synchronized, and this is often accomplished by preprocessing and interpolation. The output at each time-stamp is the predicted value. In forecasting, the predicted value is simply the value(s) of the different time-series at k units ahead. One can view this approach as the time-series analog of language models with discrete sequences. It is also possible to choose an output corresponding to a time-series not present in the data (e.g., predicting one stock price from another) or to choose an output corresponding to a discrete event (e.g., equipment failure). The main differences among all these cases lie in the specific choice of the loss function for the output at hand. In the specific case of time-series forecasting, a neat relationship can be shown between autoregressive models and echo-state networks.

Relationship with Autoregressive Models

An *autoregressive model* models the values of a time-series as a linear function of its immediate history of length p. The p coefficients of this model are learned with linear regression. Echo-state networks can be shown to be closely related to autoregressive models, in which the connections of the hidden-to-hidden matrix are sampled in a particular way. The additional power of the echo-state network over an autoregressive model arises from the nonlinearity used in the hidden-to-hidden layer. In order to understand this point, we will consider the special case of an echo-state network in which its input corresponds to a single time series and the hidden-to-hidden layers have linear activations. Now imagine that we could somehow choose the hidden-to-hidden connections in such a way that the values of the hidden state in each time-stamp is exactly equal to the values of the time-series in the last p ticks. What kind of sampled weight matrix would achieve this goal?

First, the hidden state needs to have p units, and therefore the size of W_{hh} is $p \times p$. It is easy to show that a weight matrix W_{hh} that shifts the hidden state by one unit and copies the input value to the vacated state caused by the shifting will result in a hidden state, which is exactly the same as the last window of p points. In other words, the matrix W_{hh} will have exactly $(p-1)$ non-zero entries of the form $(i, i+1)$ for each $i \in \{1 \ldots p-1\}$. As a result, pre-multiplying any p-dimensional column vector $\overline{h}_t$ with W_{hh} will shift the entries of $\overline{h}_t$ by one unit. For a 1-dimensional time-series, the element x_t is a 1-dimensional input into the tth hidden state of the echo state network, and W_{xh} is therefore of size $p \times 1$. Setting only the entry $(p, 0)$ of W_{xh} to 1 and all other entries to 0 will result in copying x_t into the first element of $\overline{h}_t$. The matrix W_{hy} is a $1 \times p$ matrix of *learned weights*, so that $W_{hy}\overline{h}_t$ yields the prediction $\hat{y}_t$ of the observed value y_t. In autoregressive modeling, the value of y_t is simply set to x_{t+k} for some lookahead k, and the value of k is often set to 1. It is noteworthy that the matrices W_{hh} and W_{xh} are fixed, and only W_{hy} needs to learned. This process leads to the development of a model that is identical to the time-series autoregressive model [3].

The main difference of the time-series autoregressive model from the echo-state network is that the latter fixes W_{hh} and W_{xh} randomly, and uses much larger dimensionalities of the hidden states. Furthermore, nonlinear activations are used in the hidden units. As long as the spectral radius of W_{hh} is (slightly) less than 1, a random choice of the matrices W_{hh} and W_{xh} with linear activations can be viewed as a decay-based variant of the autoregressive model. This is because the matrix W_{hh} only performs a random (but slightly decaying) transformation of the previous hidden state. Using a decaying random projection of the previous hidden state intuitively achieves similar goals as a sliding window-shifted copy of the previous state. The precise spectral radius of W_{hh} governs the rate of decay. With a sufficient number of hidden states, the matrix W_{hy} provides enough degrees of freedom to model any decay-based function of recent history. Furthermore, the proper scaling of the W_{xh} ensures that the most recent entry is not given too much or too little weight. Note that echo-state networks do test different scalings of the matrix W_{xh} to ensure that the effect of this input does not wipe out the contributions from the hidden states. The nonlinear activations in the echo-state network give greater power to this approach over a time-series autoregressive model. In a sense, echo-state networks can model complex nonlinear dynamics of the time-series, unlike an off-the-shelf autoregressive model.

7.7.6 Temporal Recommender Systems

Several solutions [465, 534, 565] have been proposed in recent years for temporal modeling of recommender systems. Some of these methods use temporal aspects of users, whereas others use temporal aspects of users and items. One observation is that the properties of items tend to be more strongly fixed in time than the properties of users. Therefore, solutions that use the temporal modeling only at the user level are often sufficient. However, some methods [534] perform the temporal modeling both at the user level and at the item level.

In the following, we discuss a simplification of the model discussed in [465]. In temporal recommender systems, the time-stamps associated with user ratings are leveraged for the recommendation process. Consider a case in which the observed rating of user i for item j at time-stamp t is denoted by r_{ijt}. For simplicity, we assume that the time-stamp t is simply the index of the rating in the sequential order it was received (although many models use the wall-clock time). Therefore, the sequence being modeled by the RNN is a sequence of rating values associated with the content-centric representations of the users and items to which the rating belongs. Therefore, we want to model the value of the rating as a function of content-centric inputs at each time-stamp.

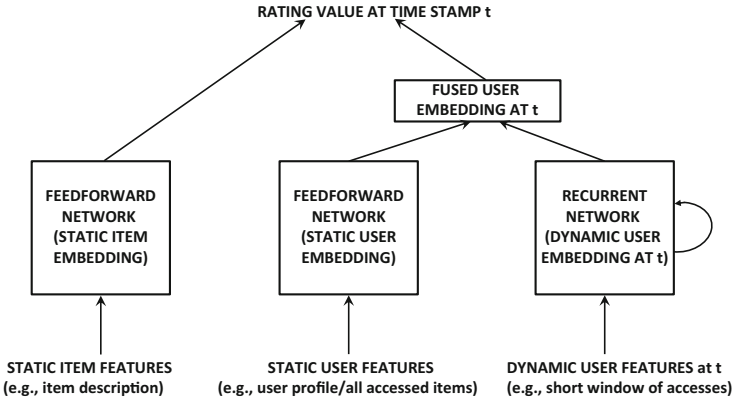


Figure 7.13: Recommendations with recurrent neural networks. At each time-stamp, the static/dynamic user features and static item features are input, and a rating value is output for that user-item combination.

We describe these content-centric representations below. The prediction of the rating r_{ijt} is assumed to be depend on (i) static features associated with the item, (ii) static features associated with the user, and (iii) the dynamic features associated with the user. The static features associated with the item might be item titles or descriptions, and one can create a bag-of-words representation of the item. The static features associated with the user might be a user-specific profile or a fixed history of accesses of this user, which does not change over the data set. The static features associated with the users are also typically represented as a bag of words, and one can even consider item-rating pairs as pseudo-keywords in order to combine user-specified keywords with ratings activity. In the case where ratings activity is used, a fixed history of accesses of the user is always leveraged for designing static features. The dynamic user features are more interesting because they are based on the dynamically changing user access history. In this case, a short history of item-rating pairs can be used as pseudo-keywords, and a bag-of-words representation can be created at time-stamp t.

In several cases, explicit ratings are not available, but implicit feedback data is available corresponding to a user clicking on an item. In the event that implicit feedback is used, negative sampling becomes necessary in which user-item pairs for which activity has not occurred are included in the sequence at random. This approach can be viewed as a hybrid between a content-based and collaborative recommendation approach. While it does use the user-item-rating triplets like a traditional recommender model, the content-centric representations of the users and items are input at each time-stamp. However, the inputs at different time-stamps correspond to different user-item pairs, and therefore the collaborative power of the patterns of ratings among different users and items is used as well.

The overall architecture of this recommender system is illustrated in Figure 7.13. It is evident that this architecture contains three different subnetworks to create feature embeddings out of static item features, static user features, and dynamic user features. The first two of these three are feed-forward networks, whereas the last of them is a recurrent neural network. First, the embeddings from the two user-centric networks are fused using either concatenation or element-wise multiplication. In the latter case, it is necessary to create embeddings of the same dimensionality for static and dynamic user features. Then, this fused user embedding at time-stamp t and the static item embedding is used to predict the rating at time-stamp t. For implicit feedback data, one can predict probabilities of positive activity for a particular user-item pair. The chosen loss function depends on the nature of the rating being predicted. The training algorithm needs to work with a consecutive sequence of training triplets (of some fixed mini-batch size) and backpropagate to the static and dynamic portions of the network simultaneously.

The aforementioned presentation has simplified several aspects of the training procedure presented in [465]. For example, it is assumed that a single rating is received at each time-stamp t, and that a fixed time-horizon is sufficient for temporal modeling. In reality, different settings might required different levels of granularity at which temporal aspects are handled. Therefore, the work in [465] proposes methods to address varying levels of granularity in the modeling process. It is also possible to perform the recommendation under a pure collaborative filtering regime without using content-centric features in any way. For example, it is possible[9] to adapt the recommender system discussed in Section 2.5.7 of Chapter 2 by using a recurrent neural network (cf. Exercise 3).

Another recent work [565] treats the problem as that of working with product-action-time triplets at an e-commerce site. The idea is that a site logs sequential actions performed by each user to various products, such as visiting a product page from a homepage, category page, or sales page, and that of actually buying the product. Each action has a *dwell time*, which indicates the amount of time that the user spends in performing that action. The dwell time is discretized into a set of intervals, which would be uniform or geometric, depending on the application at hand. It makes sense to discretize the time into geometrically increasing intervals.

One sequence is collected for each user, corresponding to the actions performed by the user. One can represent the rth element of the sequence as $(\overline{p}_r, \overline{a}_r, \overline{t}_r)$, where $\overline{p}_r$ is the one-hot encoded product, $\overline{a}_r$ is the one-hot encoded action, and $\overline{t}_r$ is the one-hot encoded discretized value of the time interval. Each of $\overline{p}_r$, $\overline{a}_r$, and $\overline{t}_r$ is a one-hot encoded vector. An embedding layer with weight matrices W_p, W_a, and W_t is used to create the representation $\overline{e}_r = (W_p\overline{p}_r, W_a\overline{a}_r, W_t\overline{t}_r)$. These matrices were pretrained with *word2vec* training applied to sequences extracted from the e-commerce site. Subsequently, the input to the recurrent

[9]Even though the adaptation from Section 2.5.7 is the most natural and obvious one, we have not seen it elsewhere in the literature. Therefore, it might be an interesting exercise for the reader to implement the adaptation of Exercise 3.

neural network is $\overline{e}_1 \ldots \overline{e}_T$, which was used to predict the outputs $\overline{o}_1 \ldots \overline{o}_T$. The output at the time-stamp t corresponds to the next action of the user at that time-stamp. Note that the embedding layer is also attached to the recurrent network, and it is fine-tuned during backpropagation (beyond its *word2vec* initialization). The original work [565] also adds an attention layer, although good results can be obtained even without this layer.

7.7.7 Secondary Protein Structure Prediction

In protein structure prediction, the elements of the sequence are the symbols representing one of the 20 amino acids. The 20 possible amino acids are akin to the vocabulary used in the text setting. Therefore, a one-hot encoding of the input is effective in these cases. Each position is associated with a class label corresponding to the secondary protein structure. This secondary structure can be either the alpha-helix, beta-sheet, or coil. Therefore, this problem can be reduced to token-level classification. A three-way softmax is used in the output layer. The work in [20] used a bidirectional recurrent neural network for prediction. This is because protein structure prediction is a problem that benefits from the context on both sides of a particular position. In general, the choice between using a uni-directional network and a bidirectional network is highly regulated by whether or not the prediction is causal to a historical segment or whether it depends on the context on both sides.

7.7.8 End-to-End Speech Recognition

In end-to-end speech recognition, one attempts to transcribe the raw audio files into character sequences while going through as few intermediate steps as possible. A small amount of preprocessing is still needed in order to make the data presentable as an input sequence. For example, the work in [157] presents the data as *spectrograms* derived from raw audio files using the *specgram* function of the *matplotlib* python toolkit. The width used was 254 Fourier windows with an overlap of 127 frames and 128 inputs per frame. The output is a character in the transcription sequence, which could include a character, a punctuation mark, a space, or even a null character. The label could be different depending on the application at hand. For example, the labels could be characters, phonemes, or musical notes. A bidirectional recurrent neural network is most appropriate to this setting, because the context on both sides of a character helps in improving accuracy.

One challenge associated with this type of setting is that we need the alignment between the frame representation of the audios and the transcription sequence. This type of alignment is not available a priori, and is in fact one of the outputs of the system. This leads to the problem of circular dependency between segmentation and recognition, which is also referred to as *Sayre's paradox*. This problem is solved with the use of *connectionist temporal classification*. In this approach, a dynamic programming algorithm [153] is combined with the (softmax) probabilistic outputs of the recurrent network in order to determine the alignment that maximizes the overall probability of generation. The reader is referred to [153, 157] for details.

7.7.9 Handwriting Recognition

A closely related application to speech recognition is that of handwriting recognition [154, 156]. In handwriting recognition, the input consists of a sequence of (x, y) coordinates, which represents the position of the tip of the pen at each time-stamp. The output corresponds to a sequence of characters written by the pen. These coordinates are then used to extract

further features such as a feature indicating whether the pen is touching the writing surface, the angles between nearby line segments, the velocity of the writing, and normalized values of the coordinates. The work in [154] extracts a total of 25 features. It is evident that multiple coordinates will create a character. However, it is hard to know exactly how many coordinates will create each character because it may vary significantly over the handwriting and style of different writers. Much like speech recognition, the issue of proper segmentation creates numerous challenges. This is the same Sayre's paradox that is encountered in speech recognition.

In unconstrained handwriting recognition, the handwriting contains a set of *strokes*, and by putting them together one can obtain characters. One possibility is to identify the strokes up front, and then use them to build characters. However, such an approach leads to inaccurate results, because the identification of stroke boundaries is an error-prone task. Since the errors tend to be additive over different phases, breaking up the task into separate stages is generally not a good idea. At a basic level, the task of handwriting recognition is no different from speech recognition. The only difference is in terms of the specific way in which the inputs and outputs are represented. As in the case of speech recognition, connectionist temporal classification is used in which a dynamic programming approach is combined with the softmax outputs of a recurrent neural network. Therefore, the alignment and the label-wise classification is performed simultaneously with dynamic programming in order to maximize the probability that a particular output sequence is generated for a particular input sequence. Readers are referred to [154, 156].

7.8 Summary

Recurrent neural networks are a class of neural networks that are used for sequence modeling. They can be expressed as time-layered networks in which the weights are shared between different layers. Recurrent neural networks can be hard to train, because they are prone to the vanishing and the exploding gradient problems. Some of these problems can be addressed with the use of enhanced training methods as discussed in Chapter 3. However, there are other ways of training more robust recurrent networks. A particular example that has found favor is the use of long short-term memory network. This network uses a gentler update process of the hidden states in order to avoid the vanishing and exploding gradient problems. Recurrent neural networks and their variants have found use in many applications such as image captioning, token-level classification, sentence classification, sentiment analysis, speech recognition, machine translation, and computational biology.

7.9 Bibliographic Notes

One of the earliest forms of the recurrent network was the Elman network [111]. This network was a precursor to modern recurrent networks. Werbos proposed the original version of backpropagation through time [526]. Another early algorithm for backpropagation in recurrent neural networks is provided in [375]. The vast majority of work on recurrent networks has been on symbolic data, although there is also some work on real-valued time series [80, 101, 559]. The regularization of recurrent neural networks is discussed in [552].

The effect of the spectral radius of the hidden-hidden matrix on the vanishing/exploding gradient problem is discussed in [220]. A detailed discussion of the exploding gradient problem and other problems associated with recurrent neural networks may be found

in [368, 369]. Recurrent neural networks (and their advanced variations) began to become more attractive after about 2010, when hardware advancements, increased data, and algorithmic tweaks made these methodologies far more attractive. The vanishing and exploding gradient problems in different types of deep networks, including recurrent networks, are discussed in [140, 205, 368]. The gradient clipping rule was discussed by Mikolov in his Ph.D. thesis [324]. The initialization of recurrent networks containing ReLUs is discussed in [271].

Early variants of the recurrent neural network included the echo-state network [219], which is also referred to as the *liquid-state machine* [304]. This paradigm is also referred to as *reservoir computing*. An overview of echo-state networks in the context of reservoir computing principles is provided in [301]. The use of batch normalization is discussed in [214]. Teacher forcing methods are discussed in [105]. Initialization strategies that reduce the effect of the vanishing and exploding gradient problems are discussed in [140].

The LSTM was first proposed in [204], and its use for language modeling is discussed in [476]. The challenges associated with training recurrent neural networks are discussed in [205, 368, 369]. It has been shown [326] that it is also possible to address some of the problems associated with the vanishing and exploding gradient problems by imposing constraints on the hidden-to-hidden matrix. Specifically, a block of the matrix is constrained to be close to the identity matrix, so that the corresponding hidden variables are updated slowly in much the same way as the memory of the LSTM is updated slowly. Several variations of recurrent neural networks and LSTMs for language modeling are discussed in [69, 71, 151, 152, 314, 328]. Bidirectional recurrent neural networks are proposed in [434]. The particular discussion of LSTMs in this chapter is based on [151], and an alternative gated recurrent unit (GRU) is presented in [69, 71]. A guide to understanding recurrent neural networks is available in [233]. Further discussions on the sequence-centric and natural language applications of recurrent neural networks are available in [143, 298]. LSTM networks are also used for sequence labeling [150], which is useful in sentiment analysis [578]. The use of a combination of convolutional neural networks and recurrent neural networks for image captioning is discussed in [225, 509]. Sequence-to-sequence learning methods for machine translation are discussed in [69, 231, 480]. Bidirectional recurrent networks and LSTMs for protein structure prediction, handwriting recognition, translation, and speech recognition are discussed in [20, 154, 155, 157, 378, 477]. In recent years, neural networks have also been used in temporal collaborative filtering, which was first introduced in [258]. Numerous methods for temporal collaborative filtering are discussed in [465, 534, 560]. A generative model for dialogues with recurrent networks is discussed in [439, 440]. The use of recurrent neural networks for action recognition is discussed in [504].

Recurrent neural networks have also been generalized to recursive neural networks for modeling arbitrary structural relationships in the data [379]. These methods generalize the use of the recurrent neural networks to trees (rather than sequences) by considering a tree-like computational graph. Their use for discovering task-dependent representations is discussed in [144]. These methods can be applied to cases in which data structures are considered as inputs to the neural network [121]. Recurrent neural networks are a special case of recursive neural network in which the structure corresponds to a linear chain of dependencies. The use of recursive neural networks for various types of natural-language and scene-processing applications is discussed in [459, 460, 461].

7.9.1 Software Resources

Recurrent neural networks and their variants are supported by numerous software frameworks like *Caffe* [571], *Torch* [572], *Theano* [573], and *TensorFlow* [574]. Several other frame-

works like **DeepLearning4j** provide implementations of LSTMs [617]. Implementations of sentiment analysis with LSTM networks are available at [578]. This approach is based on the sequence labeling technique presented in [152]. A notable piece of code [580] is a character-level RNN, and it is particularly instructive for learning purposes. The conceptual description of this code is provided in [233, 618].

7.10 Exercises

1. Download the character-level RNN in [580], and train it on the "*tiny Shakespeare*" data set available at the same location. Create outputs of the language model after training for (i) 5 epochs, (ii) 50 epochs, and (iii) 500 epochs. What significant differences do you see between the three outputs?

2. Consider an echo-state network in which the hidden states are partitioned into K groups with p/K units each. The hidden states of a particular group are only allowed to have connections within their own group in the next time-stamp. Discuss how this approach is related to an ensemble method in which K independent echo-state networks are constructed and the predictions of the K networks are averaged.

3. Show how you can modify the feed-forward architecture discussed in Section 2.5.7 of Chapter 2 in order to create a recurrent neural network that can handle temporal recommender systems. Implement this adaptation and compare its performance to the feed-forward architecture on the Netflix prize data set.

4. Consider a recurrent network in which the hidden states have a dimensionality of 2. Every entry of the 2×2 matrix W_{hh} of transformations between hidden states is 3.5. Furthermore, sigmoid activation is used between hidden states of different temporal layers. Would such a network be more prone to the vanishing or the exploding gradient problem?

5. Suppose that you have a large database of biological strings containing sequences of nucleobases drawn from $\{A, C, T, G\}$. Some of these strings contain unusual mutations representing changes in the nucleobases. Propose an unsupervised method (i.e., neural architecture) using RNNs in order to detect these mutations.

6. How would your architecture for the previous question change if you were given a training database in which the mutation positions in each sequence were tagged, and the test database was untagged?

7. Recommend possible methods for pre-training the input and output layers in the machine translation approach with sequence-to-sequence learning.

8. Consider a social network with a large volume of messages sent between sender-receiver pairs, and we are interested only in the messages containing an identifying keyword, referred to as a *hashtag*. Create a real-time model using an RNN, which has the capability to recommend hashtags of interest to each user, together with potential followers of that user who might be interested in messages related to that hashtag. Assume that you have enough computational resources to incrementally train an RNN.

9. If the training data set is re-scaled by a particular factor, do the learned weights of either batch normalization or layer normalization change? What would be your answer

if only a small subset of points in the training data set are re-scaled? Would the learned weights in either normalization method be affected if the data set is re-centered?

10. Consider a setting in which you have a large database of pairs of sentences in different languages. Although you have sufficient representation of each language, some *pairs* might not be well represented in the database. Show how you can use this training data to (i) create the same universal code for a particular sentence across all languages, and (ii) have the ability to translate even between pairs of languages not well represented in the database.

Chapter 8

Convolutional Neural Networks

"The soul never thinks without a picture."—Aristotle

8.1 Introduction

Convolutional neural networks are designed to work with grid-structured inputs, which have strong spatial dependencies in local regions of the grid. The most obvious example of grid-structured data is a 2-dimensional image. This type of data also exhibits spatial dependencies, because adjacent spatial locations in an image often have similar color values of the individual pixels. An additional dimension captures the different colors, which creates a 3-dimensional input *volume*. Therefore, the features in a convolutional neural network have dependencies among one another based on spatial distances. Other forms of sequential data like text, time-series, and sequences can also be considered special cases of grid-structured data with various types of relationships among adjacent items. The vast majority of applications of convolutional neural networks focus on image data, although one can also use these networks for all types of temporal, spatial, and spatiotemporal data.

An important property of image data is that it exhibits a certain level of *translation invariance*, which is not the case in many other types of grid-structured data. For example, a banana has the same interpretation, whether it is at the top or the bottom of an image. Convolutional neural networks tend to create similar feature values from local regions with similar patterns. One advantage of image data is that the effects of specific inputs on the feature representations can often be described in an intuitive way. Therefore, this chapter will primarily work with the image data setting. A brief discussion will also be devoted to the applications of convolutional neural networks to other settings.

An important defining characteristic of convolutional neural networks is an operation, which is referred to as *convolution*. A convolution operation is a dot-product operation between a grid-structured set of weights and similar grid-structured inputs drawn from different spatial localities in the input volume. This type of operation is useful for data with a high level of spatial or other locality, such as image data. Therefore, convolutional neural networks are defined as networks that use the convolutional operation in at least one layer, although most convolutional neural networks use this operation in multiple layers.

8.1.1 Historical Perspective and Biological Inspiration

Convolutional neural networks were one of the first success stories of deep learning, well before recent advancements in training techniques led to improved performance in other types of architectures. In fact, the eye-catching successes of some convolutional neural network architectures in image-classification contests after 2011 led to broader attention to the field of deep learning. Long-standing benchmarks like *ImageNet* [581] with a top-5 classification error-rate of more than 25% were brought down to less than 4% in the years between 2011 and 2015. Convolutional neural networks are well suited to the process of hierarchical feature engineering with depth; this is reflected in the fact that the deepest neural networks in all domains are drawn from the field of convolutional networks. Furthermore, these networks also represent excellent examples of how biologically inspired neural networks can sometimes provide ground-breaking results. The best convolutional neural networks today reach or exceed human-level performance, a feat considered impossible by most experts in computer vision only a couple of decades back.

The early motivation for convolutional neural networks was derived from experiments by Hubel and Wiesel on a cat's visual cortex [212]. The visual cortex has small regions of cells that are sensitive to specific regions in the visual field. In other words, if specific areas of the visual field are excited, then those cells in the visual cortex will be activated as well. Furthermore, the excited cells also depend on the shape and orientation of the objects in the visual field. For example, vertical edges cause some neuronal cells to be excited, whereas horizontal edges cause other neuronal cells to be excited. The cells are connected using a layered architecture, and this discovery led to the conjecture that mammals use these different layers to construct portions of images at different levels of abstraction. From a machine learning point of view, this principle is similar to that of hierarchical feature extraction. As we will see later, convolutional neural networks achieve something similar by encoding primitive shapes in earlier layers, and more complex shapes in later layers.

Based on these biological inspirations, the earliest neural model was the *neocognitron* [127]. However, there were several differences between this model and the modern convolutional neural network. The most prominent of these differences was that the notion of weight sharing was not used. Based on this architecture, one of the first fully convolutional architectures, referred to as *LeNet-5* [279], was developed. This network was used by banks to identify hand-written numbers on checks. Since then, the convolutional neural network has not evolved much; the main difference is in terms of using more layers and stable activation functions like the ReLU. Furthermore, numerous training tricks and powerful hardware options are available to achieve better success in training when working with deep networks and large data sets.

A factor that has played an important role in increasing the prominence of convolutional neural networks has been the annual *ImageNet* competition [582] (also referred to as *"ImageNet Large Scale Visual Recognition Challenge [ILSVRC]"*). The ILSVRC competition uses the *ImageNet* data set [581], which is discussed in Section 1.8.2 of Chapter 1. Convolutional

neural networks have been consistent winners of this contest since 2012. In fact, the dominance of convolutional neural networks for image classification is so well recognized today that almost all entries in recent editions of this contest have been convolutional neural networks. One of the earliest methods that achieved success in the 2012 *ImageNet* competition by a large margin was *AlexNet* [255]. Furthermore, the improvements in accuracy have been so extraordinarily large in the last few years that it has changed the landscape of research in the area. In spite of the fact that the vast majority of eye-catching performance gains have occurred from 2012 to 2015, the architectural differences between recent winners and some of the earliest convolutional neural networks are rather small at least at a conceptual level. Nevertheless, small details seem to matter a lot when working with almost all types of neural networks.

8.1.2 Broader Observations About Convolutional Neural Networks

The secret to the success of any neural architecture lies in tailoring the structure of the network with a semantic understanding of the domain at hand. Convolutional neural networks are heavily based on this principle, because they use sparse connections with a high-level of parameter-sharing in a domain-sensitive way. In other words, not all states in a particular layer are connected to those in the previous layer in an indiscriminate way. Rather, the value of a feature in a particular layer is connected only to a local spatial region in the previous layer with a consistent set of shared parameters across the full spatial footprint of the image. This type of architecture can be viewed as a domain-aware regularization, which was derived from the biological insights in Hubel and Wiesel's early work. In general, the success of the convolutional neural network has important lessons for other data domains. A carefully designed architecture, in which the relationships and dependencies among the data items are used in order to reduce the parameter footprint, provides the key to results of high accuracy.

A significant level of domain-aware regularization is also available in recurrent neural networks, which share the parameters from different temporal periods. This sharing is based on the assumption that temporal dependencies remain invariant with time. Recurrent neural networks are based on intuitive understanding of temporal relationships, whereas convolutional neural networks are based on an intuitive understanding of spatial relationships. The latter intuition was directly extracted from the organization of biological neurons in a cat's visual cortex. This outstanding success provides a motivation to explore how neuroscience may be leveraged to design neural networks in clever ways. Even though artificial neural networks are only caricatures of the true complexity of the biological brain, one should not underestimate the intuition that one can obtain by studying the basic principles of neuroscience [176].

Chapter Organization

This chapter is organized as follows. The next section will introduce the basics of a convolutional neural network, the various operations, and the way in which they are organized. The training process for convolutional networks is discussed in Section 8.3. Case studies with some typical convolutional neural networks that have won recent competitions are discussed in Section 8.4. The convolutional autoencoder is discussed in Section 8.5. A variety of applications of convolutional networks are discussed in Section 8.6. A summary is given in Section 8.7.

8.2 The Basic Structure of a Convolutional Network

In convolutional neural networks, the states in each layer are arranged according to a spatial grid structure. These spatial relationships are inherited from one layer to the next because each feature value is based on a small local spatial region in the previous layer. It is important to maintain these spatial relationships among the grid cells, because the convolution operation and the transformation to the next layer is critically dependent on these relationships. Each layer in the convolutional network is a 3-dimensional grid structure, which has a *height*, *width*, and *depth*. The depth of a layer in a convolutional neural network should not be confused with the depth of the network itself. The word "depth" (when used in the context of a single layer) refers to the number of *channels* in each layer, such as the number of primary color channels (e.g., blue, green, and red) in the input image or the number of feature maps in the hidden layers. The use of the word "depth" to refer to both the number of feature maps in each layer as well as the number of layers is an unfortunate overloading of terminology used in convolutional networks, but we will be careful while using this term, so that it is clear from its context.

The convolutional neural network functions much like a traditional feed-forward neural network, except that the operations in its layers are spatially organized with sparse (and carefully designed) connections between layers. The three types of layers that are commonly present in a convolutional neural network are *convolution*, *pooling*, and *ReLU*. The ReLU activation is no different from a traditional neural network. In addition, a final set of layers is often fully connected and maps in an application-specific way to a set of output nodes. In the following, we will describe each of the different types of operations and layers, and the typical way in which these layers are interleaved in a convolutional neural network.

Why do we need depth in each layer of a convolutional neural network? To understand this point, let us examine how the input to the convolutional neural network is organized. The input data to the convolutional neural network is organized into a 2-dimensional grid structure, and the values of the individual grid points are referred to as *pixels*. Each pixel, therefore, corresponds to a spatial location within the image. However, in order to encode the precise color of the pixel, we need a multidimensional array of values at each grid location. In the RGB color scheme, we have an intensity of the three primary colors, corresponding to red, green, and blue, respectively. Therefore, if the spatial dimensions of an image are 32×32 pixels and the depth is 3 (corresponding to the RGB color channels), then the overall number of pixels in the image is $32 \times 32 \times 3$. This particular image size is quite common, and also occurs in a popularly used data set for benchmarking, known as CIFAR-10 [583]. An example of this organization is shown in Figure 8.1(a). It is natural to represent the input layer in this 3-dimensional structure because two dimensions are devoted to spatial relationships and a third dimension is devoted to the independent properties along these channels. For example, the intensities of the primary colors are the independent properties in the first layer. In the hidden layers, these independent properties correspond to various types of shapes extracted from local regions of the image. For the purpose of discussion, assume that the input in the qth layer is of size $L_q \times B_q \times d_q$. Here, L_q refers to the *height* (or length), B_q refers to the width (or breadth), and d_q is the depth. In almost all image-centric applications, the values of L_q and B_q are the same. However, we will work with separate notations for height and width in order to retain generality in presentation.

For the first (input) layer, these values are decided by the nature of the input data and its preprocessing. In the above example, the values are $L_1 = 32$, $B_1 = 32$, and $d_1 = 3$. Later layers have exactly the same 3-dimensional organization, except that each of the d_q 2-dimensional grid of values for a particular input can no longer be considered a grid of

raw pixels. Furthermore, the value of d_q is much larger than three for the hidden layers because the number of independent properties of a given local region that are relevant to classification can be quite significant. For $q > 1$, these grids of values are referred to as *feature maps* or *activation maps*. These values are analogous to the values in the hidden layers in a feed-forward network.

In the convolutional neural network, the parameters are organized into sets of 3-dimensional structural units, known as *filters* or *kernels*. The filter is usually square in terms of its spatial dimensions, which are typically much smaller than those of the layer the filter is applied to. On the other hand, *the depth of a filter is always same is the same as that of the layer to which it is applied*. Assume that the dimensions of the filter in the qth layer are $F_q \times F_q \times d_q$. An example of a filter with $F_1 = 5$ and $d_1 = 3$ is shown in Figure 8.1(a). It is common for the value of F_q to be small and odd. Examples of commonly used values of F_q are 3 and 5, although there are some interesting cases in which it is possible to use $F_q = 1$.

The *convolution operation* places the filter at each possible position in the image (or hidden layer) so that the filter fully overlaps with the image, and performs a dot product between the $F_q \times F_q \times d_q$ parameters in the filter and the matching grid in the input volume (with same size $F_q \times F_q \times d_q$). The dot product is performed by treating the entries in the relevant 3-dimensional region of the input volume and the filter as vectors of size $F_q \times F_q \times d_q$, so that the elements in both vectors are ordered based on their corresponding positions in the grid-structured volume. How many possible positions are there for placing the filter? This question is important, because each such position therefore defines a spatial "pixel" (or, more accurately, a *feature*) in the next layer. In other words, the number of alignments between the filter and image defines the spatial height and width of the next hidden layer. The relative spatial positions of the features in the next layer are defined based on the relative positions of the upper left corners of the corresponding spatial grids in the previous layer. When performing convolutions in the qth layer, one can align the filter at $L_{q+1} = (L_q - F_q + 1)$ positions along the height and $B_{q+1} = (B_q - F_q + 1)$ along the width of the image (without having a portion of the filter "sticking out" from the borders of the image). This results in a total of $L_{q+1} \times B_{q+1}$ possible dot products, which defines the size of the next hidden layer. In the previous example, the values of L_2 and B_2 are therefore defined as follows:

$$L_2 = 32 - 5 + 1 = 28$$
$$B_2 = 32 - 5 + 1 = 28$$

The next hidden layer of size 28×28 is shown in Figure 8.1(a). However, this hidden layer also has a depth of size $d_2 = 5$. Where does this depth come from? This is achieved by using 5 different filters with their own independent sets of parameters. Each of these 5 sets of spatially arranged features obtained from the output of a single filter is referred to as a *feature map*. Clearly, an increased number of feature maps is a result of a larger number of filters (i.e., parameter footprint), which is $F_q^2 \cdot d_q \cdot d_{q+1}$ for the qth layer. *The number of filters used in each layer controls the capacity of the model because it directly controls the number of parameters*. Furthermore, increasing the number of filters in a particular layer increases the number of feature maps (i.e., depth) of the next layer. It is possible for different layers to have very different numbers of feature maps, depending on the number of filters we use for the convolution operation in the previous layer. For example, the input layer typically only has three color channels, but it is possible for the each of the later hidden layers to have depths (i.e., number of feature maps) of more than 500. The idea here is that each

filter tries to identify a particular type of spatial pattern in a small rectangular region of the image, and therefore a large number of filters is required to capture a broad variety of the possible shapes that are combined to create the final image (unlike the case of the input layer, in which three RGB channels are sufficient). Typically, the later layers tend to have a smaller spatial footprint, but greater depth in terms of the number of feature maps. For example, the filter shown in Figure 8.1(b) represents a horizontal edge detector on a grayscale image with one channel. As shown in Figure 8.1(b), the resulting feature will have high activation at each position where a horizontal edge is seen. A perfectly vertical edge will give zero activation, whereas a slanted edge might give intermediate activation. Therefore, sliding the filter everywhere in the image will already detect several key outlines of the image in a single feature map of the output volume. Multiple filters are used to create an output volume with more than one feature map. For example, a different filter might create a spatial feature map of vertical edge activations.

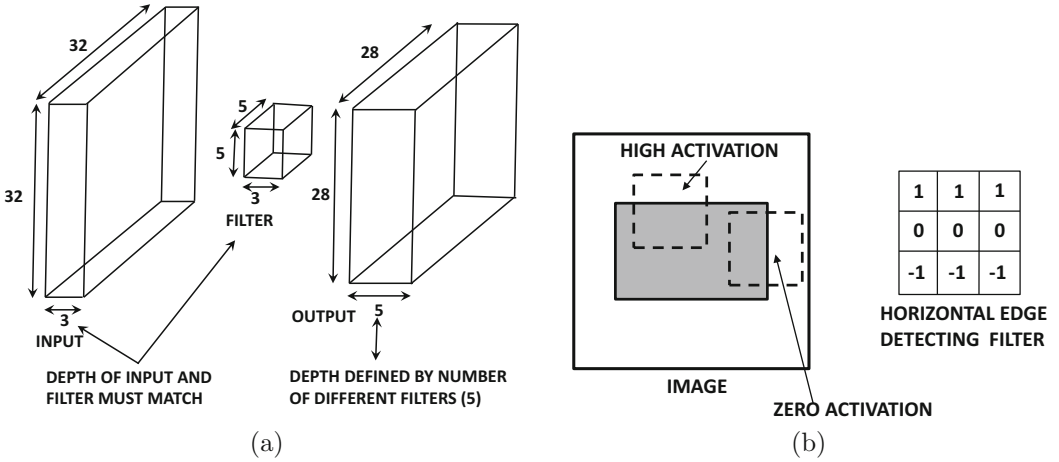


Figure 8.1: (a) The convolution between an input layer of size $32 \times 32 \times 3$ and a filter of size $5 \times 5 \times 3$ produces an output layer with spatial dimensions 28×28. The depth of the resulting output depends on the number of distinct filters and not on the dimensions of the input layer or filter. (b) Sliding a filter around the image tries to look for a particular feature in various windows of the image.

We are now ready to formally define the convolution operation. The pth filter in the qth layer has parameters denoted by the 3-dimensional tensor $W^{(p,q)} = [w_{ijk}^{(p,q)}]$. The indices i, j, k indicate the positions along the height, width, and depth of the filter. The feature maps in the qth layer are represented by the 3-dimensional tensor $H^{(q)} = [h_{ijk}^{(q)}]$. When the value of q is 1, the special case corresponding to the notation $H^{(1)}$ simply represents the input layer (which is not hidden). Then, the convolutional operations from the qth layer to the $(q+1)$th layer are defined as follows:

$$h_{ijp}^{(q+1)} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w_{rsk}^{(p,q)} h_{i+r-1,j+s-1,k}^{(q)} \qquad \forall i \in \{1 \ldots, L_q - F_q + 1\}$$

$$\forall j \in \{1 \ldots B_q - F_q + 1\}$$

$$\forall p \in \{1 \ldots d_{q+1}\}$$

The expression above seems notationally complex, although the underlying convolutional operation is really a simple dot product over the entire volume of the filter, which is repeated over all valid spatial positions (i, j) and filters (indexed by p). It is intuitively helpful to understand a convolution operation by placing the filter at each of the 28×28 possible spatial positions in the first layer of Figure 8.1(a) and performing a dot product between the vector of $5 \times 5 \times 3 = 75$ values in the filter and the corresponding 75 values in $H^{(1)}$. Even though the size of the input layer in Figure 8.1(a) is 32×32, there are only $(32-5+1) \times (32-5+1)$ possible spatial alignments between an input volume of size 32×32 and a filter of size 5×5.

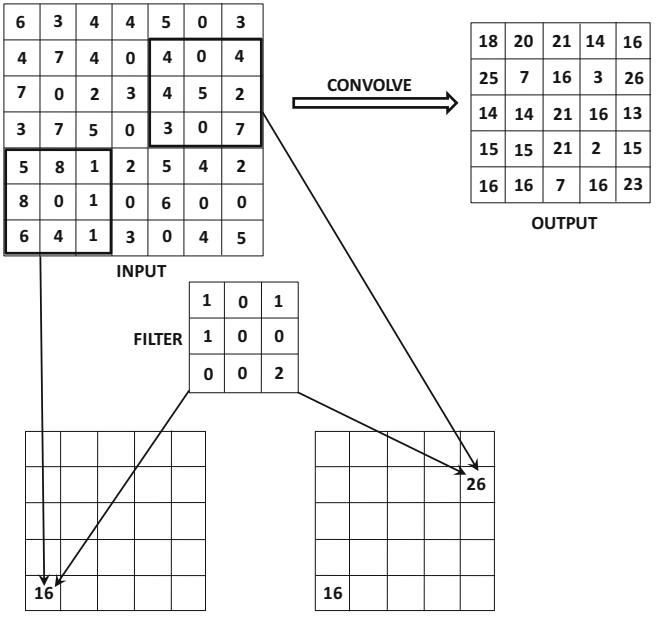


Figure 8.2: An example of a convolution between a $7 \times 7 \times 1$ input and a $3 \times 3 \times 1$ filter with stride of 1. A depth of 1 has been chosen for the filter/input for simplicity. For depths larger than 1, the contributions of each input feature map will be added to create a single value in the feature map. A single filter will always create a single feature map irrespective of its depth.

The convolution operation brings to mind Hubel and Wiesel's experiments that use the activations in small regions of the visual field to activate particular neurons. In the case of convolutional neural networks, this visual field is defined by the filter, which is applied to all locations of the image in order to detect the presence of a shape at each spatial location. Furthermore, the filters in earlier layers tend to detect more primitive shapes, whereas the filters in later layers create more complex compositions of these primitive shapes. This is not particularly surprising because most deep neural networks are good at hierarchical feature engineering.

One property of convolution is that it shows *equivariance to translation*. In other words, if we shifted the pixel values in the input in any direction by one unit and then applied convolution, the corresponding feature values will shift with the input values. This is because of the shared parameters of the filter across the entire convolution. The reason for sharing

parameters across the entire convolution is that the presence of a particular shape in any part of the image should be processed in the same way irrespective of its specific spatial location.

In the following, we provide an example of the convolution operation. In Figure 8.2, we have shown an example of an input layer and a filter with depth 1 for simplicity (which does occur in the case of grayscale images with a single color channel). Note that the depth of a layer must exactly match that of its filter/kernel, and the contributions of the dot products over all the feature maps in the corresponding grid region of a particular layer will need to be added (in the general case) to create a single output feature value in the next layer. Figure 8.2 depicts two specific examples of the convolution operations with a layer of size $7 \times 7 \times 1$ and a $3 \times 3 \times 1$ filter in the bottom row. Furthermore, the entire feature map of the next layer is shown on the upper right-hand side of Figure 8.2. Examples of two convolution operations are shown in which the outputs are 16 and 26, respectively. These values are arrived at by using the following multiplication and aggregation operations:

$$5 \times 1 + 8 \times 1 + 1 \times 1 + 1 \times 2 = 16$$
$$4 \times 1 + 4 \times 1 + 4 \times 1 + 7 \times 2 = 26$$

The multiplications with zeros have been omitted in the above aggregation. In the event that the depths of the layer and its corresponding filter are greater than 1, the above operations are performed for each spatial map and then aggregated across the entire depth of the filter.

A convolution in the qth layer increases the *receptive field* of a feature from the qth layer to the $(q+1)$th layer. In other words, each feature in the next layer captures a larger spatial region in the input layer. For example, when using a 3×3 filter convolution successively in three layers, the activations in the first, second, and third hidden layers capture pixel regions of size 3×3, 5×5, and 7×7, respectively, in the *original input image*. As we will see later, other types of operations increase the receptive fields further, as they reduce the size of the spatial footprint of the layers. This is a natural consequence of the fact that features in later layers capture complex characteristics of the image over larger spatial regions, and then combine the simpler features in earlier layers.

When performing the operations from the qth layer to the $(q+1)$th layer, the depth d_{q+1} of the computed layer depends on the *number* of filters in the qth layer, and it is independent of the *depth* of the qth layer or any of its other dimensions. In other words, the depth d_{q+1} in the $(q+1)$th layer is always equal to the number of filters in the qth layer. For example, the depth of the second layer in Figure 8.1(a) is 5, because a total of five filters are used in the first layer for the transformation. However, in order to perform the convolutions in the second layer (to create the third layer), one must now use filters of depth 5 in order to match the new depth of this layer, even though filters of depth 3 were used in the convolutions of the first layer (to create the second layer).

8.2.1 Padding

One observation is that the convolution operation reduces the size of the $(q+1)$th layer in comparison with the size of the qth layer. This type of reduction in size is not desirable in general, because it tends to lose some information along the borders of the image (or of the feature map, in the case of hidden layers). This problem can be resolved by using *padding*. In padding, one adds $(F_q - 1)/2$ "pixels" all around the borders of the feature map in order to maintain the spatial footprint. Note that these pixels are really feature values in the case of padding hidden layers. The value of each of these padded feature values is set

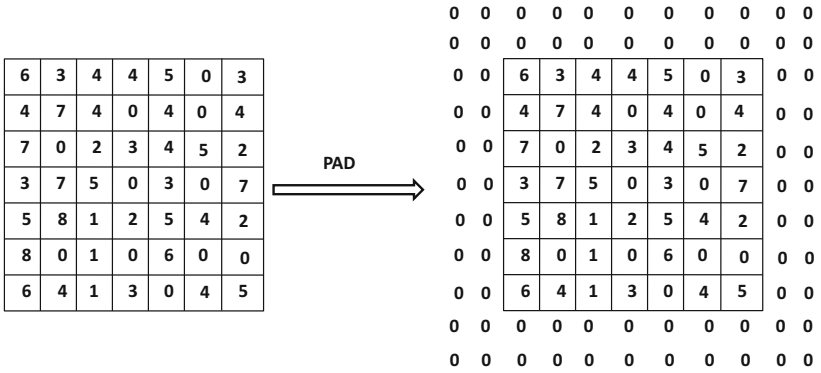


Figure 8.3: An example of padding. Each of the d_q activation maps in the entire depth of the qth layer are padded in this way.

to 0, irrespective of whether the input or the hidden layers are being padded. As a result, the spatial height and width of the input volume will both increase by $(F_q - 1)$, which is exactly what they reduce by (in the output volume) after the convolution is performed. The padded portions do not contribute to the final dot product because their values are set to 0. In a sense, what padding does is to allow the convolution operation with a portion of the filter "sticking out" from the borders of the layer and then performing the dot product only over the portion of the layer where the values are defined. This type of padding is referred to as *half-padding* because (almost) half the filter is sticking out from all sides of the spatial input in the case where the filter is placed in its extreme spatial position along the edges. Half-padding is designed to maintain the spatial footprint exactly.

When padding is not used, the resulting "padding" is also referred to as a *valid padding*. Valid padding generally does not work well from an experimental point of view. Using half-padding ensures that some of the critical information at the borders of the layer is represented in a standalone way. In the case of valid padding, the contributions of the pixels on the borders of the layer will be under-represented compared to the central pixels in the next hidden layer, which is undesirable. Furthermore, this under-representation will be compounded over multiple layers. Therefore, padding is typically performed in all layers, and not just in the first layer where the spatial locations correspond to input values. Consider a situation in which the layer has size $32 \times 32 \times 3$ and the filter is of size $5 \times 5 \times 3$. Therefore, $(5-1)/2 = 2$ zeros are padded on all sides of the image. As a result, the 32×32 spatial footprint first increases to 36×36 because of padding, and then it reduces back to 32×32 after performing the convolution. An example of the padding of a single feature map is shown in Figure 8.3, where two zeros are padded on all sides of the image (or feature map). This is a similar situation as discussed above (in terms of addition of two zeros), except that the spatial dimensions of the image are much smaller than 32×32 in order to enable illustration in a reasonable amount of space.

Another useful form of padding is *full-padding*. In full-padding, we allow (almost) the *full* filter to stick out from various sides of the input. In other words, a portion of the filter of size $F_q - 1$ is allowed to stick out from any side of the input with an overlap of only one spatial feature. For example, the kernel and the input image might overlap at a single pixel at an extreme corner. Therefore, the input is padded with $(F_q - 1)$ zeros on each side. In other words, each spatial dimension of the input increases by $2(F_q - 1)$. Therefore, if the

input dimensions in the original image are L_q and B_q, the padded spatial dimensions in the input volume become $L_q + 2(F_q - 1)$ and $B_q + 2(F_q - 1)$. After performing the convolution, the feature-map dimensions in layer $(q+1)$ become $L_q + F_q - 1$ and $B_q + F_q - 1$, respectively. While convolution normally reduces the spatial footprint, full padding *increases* the spatial footprint. Interestingly, full-padding increases each dimension of the spatial footprint by the same value $(F_q - 1)$ that no-padding decreases it. *This relationship is not a coincidence because a "reverse" convolution operation can be implemented by applying another convolution on the fully padded output (of the original convolution) with an appropriately defined kernel of the same size.* This type of "reverse" convolution occurs frequently in the backpropagation and autoencoder algorithms for convolutional neural networks. Fully padded inputs are useful because they increase the spatial footprint, which is required in several types of convolutional autoencoders.

8.2.2 Strides

There are other ways in which convolution can reduce the spatial footprint of the image (or hidden layer). The above approach performs the convolution at every position in the spatial location of the feature map. However, it is not necessary to perform the convolution at every spatial position in the layer. One can reduce the level of granularity of the convolution by using the notion of *strides*. The description above corresponds to the case when a stride of 1 is used. When a stride of S_q is used in the qth layer, the convolution is performed at the locations 1, $S_q + 1$, $2\,S_q + 1$, and so on along both spatial dimensions of the layer. The spatial size of the output on performing this convolution[1] has height of $(L_q - F_q)/S_q + 1$ and a width of $(B_q - F_q)/S_q + 1$. As a result, the use of strides will result in a reduction of each spatial dimension of the layer by a factor of approximately S_q and the area by S_q^2, although the actual factor may vary because of edge effects. It is most common to use a stride of 1, although a stride of 2 is occasionally used as well. It is rare to use strides more than 2 in normal circumstances. Even though a stride of 4 was used in the input layer of the winning architecture [255] of the ILSVRC competition of 2012, the winning entry in the subsequent year reduced the stride to 2 [556] to improve accuracy. Larger strides can be helpful in memory-constrained settings or to reduce overfitting if the spatial resolution is unnecessarily high. Strides have the effect of rapidly increasing the receptive field of each feature in the hidden layer, while reducing the spatial footprint of the entire layer. An increased receptive field is useful in order to capture a complex feature in a larger spatial region of the image. As we will see later, the hierarchical feature engineering process of a convolutional neural network captures more complex shapes in later layers. Historically, the receptive fields have been increased with another operation, known as the *max-pooling* operation. In recent years, larger strides have been used in lieu [184, 466] of max-pooling operations, which will be discussed later.

8.2.3 Typical Settings

It is common to use stride sizes of 1 in most settings. Even when strides are used, small strides of size 2 are used. Furthermore, it is common to have $L_q = B_q$. In other words, it is desirable to work with square images. In cases where the input images are not square, preprocessing is used to enforce this property. For example, one can extract square patches

[1] Here, it is assumed that $(L_q - F_q)$ is exactly divisible by S_q in order to obtain a clean fit of the convolution filter with the original image. Otherwise, some ad hoc modifications are needed to handle edge effects. In general, this is not a desirable solution.

of the image to create the training data. The number of filters in each layer is often a power of 2, because this often results in more efficient processing. Such an approach also leads to hidden layer depths that are powers of 2. Typical values of the spatial extent of the filter size (denoted by F_q) are 3 or 5. In general, small filter sizes often provide the best results, although some practical challenges exist in using filter sizes that are too small. Small filter sizes typically lead to deeper networks (for the same parameter footprint) and therefore tend to be more powerful. In fact, one of the top entries in an ILSVRC contest, referred to as *VGG* [454], was the first to experiment with a spatial filter dimension of only $F_q = 3$ for all layers, and the approach was found to work very well in comparison with larger filter sizes.

Use of Bias

As in all neural networks, it is also possible to add biases to the forward operations. Each unique filter in a layer is associated with its own bias. Therefore, the pth filter in the qth layer has bias $b^{(p,q)}$. When any convolution is performed with the pth filter in the qth layer, the value of $b^{(p,q)}$ is added to the dot product. The use of the bias simply increases the number of parameters in each filter by 1, and therefore it is not a significant overhead. Like all other parameters, the bias is learned during backpropagation. One can treat the bias as a weight of a connection whose input is always set to +1. This special input is used in all convolutions, irrespective of the spatial location of the convolution. Therefore, one can assume that a special pixel appears in the input whose value is always set to 1. Therefore, the number of input features in the qth layer is $1 + L_q \times B_q \times d_q$. This is a standard feature-engineering trick that is used for handling bias in all forms of machine learning.

8.2.4 The ReLU Layer

The convolution operation is interleaved with the pooling and ReLU operations. The ReLU activation is not very different from how it is applied in a traditional neural network. For each of the $L_q \times B_q \times d_q$ values in a layer, the ReLU activation function is applied to it to create $L_q \times B_q \times d_q$ thresholded values. These values are then passed on to the next layer. Therefore, applying the ReLU does not change the dimensions of a layer because it is a simple one-to-one mapping of activation values. In traditional neural networks, the activation function is combined with a linear transformation with a matrix of weights to create the next layer of activations. Similarly, a ReLU typically follows a convolution operation (which is the rough equivalent of the linear transformation in traditional neural networks), and the ReLU layer is often not explicitly shown in pictorial illustrations of the convolution neural network architectures.

It is noteworthy that the use of the ReLU activation function is a recent evolution in neural network design. In the earlier years, saturating activation functions like sigmoid and tanh were used. However, it was shown in [255] that the use of the ReLU has tremendous advantages over these activation functions both in terms of speed and accuracy. Increased speed is also connected to accuracy because it allows one to use deeper models and train them for a longer time. In recent years, the use of the ReLU activation function has replaced the other activation functions in convolutional neural network design to an extent that this chapter will simply use the ReLU as the default activation function (unless otherwise mentioned).