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,

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-319-94463-0_4

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

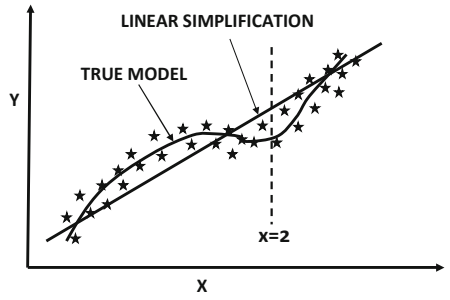


Figure 4.1: An example of a nonlinear distribution in which one would expect a model with $d = 3$ to work better than a linear model with $d = 1$.

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

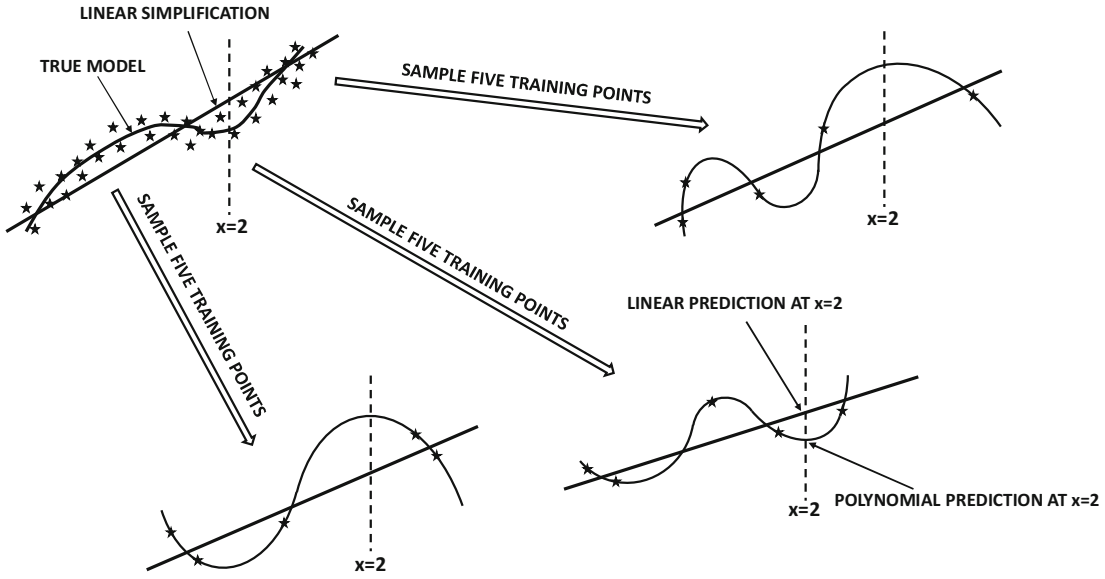


Figure 4.2: **Overfitting with increased model complexity:** The linear model does not change much with the training data, whereas the polynomial model changes drastically. As a result, the inconsistent predictions of the polynomial model at $x = 2$ are often more inaccurate than those of the linear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

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

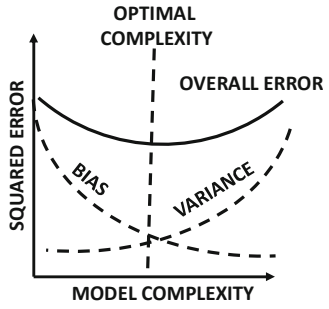


Figure 4.3: The trade-off between bias and variance usually causes a point of optimal model complexity.

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.

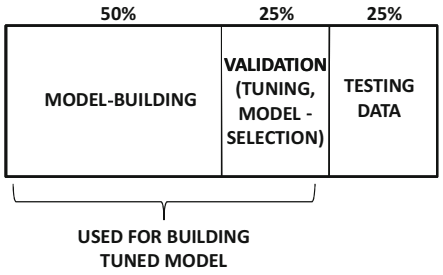


Figure 4.4: Partitioning a labeled data set for evaluation design

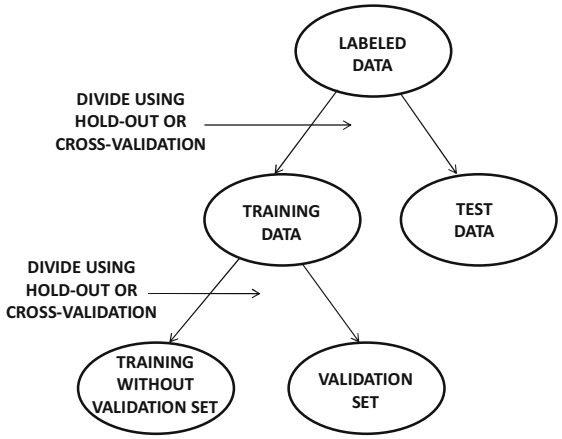


Figure 4.5: Hierarchical division into training, validation, and testing portions

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 backpropagation. 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.

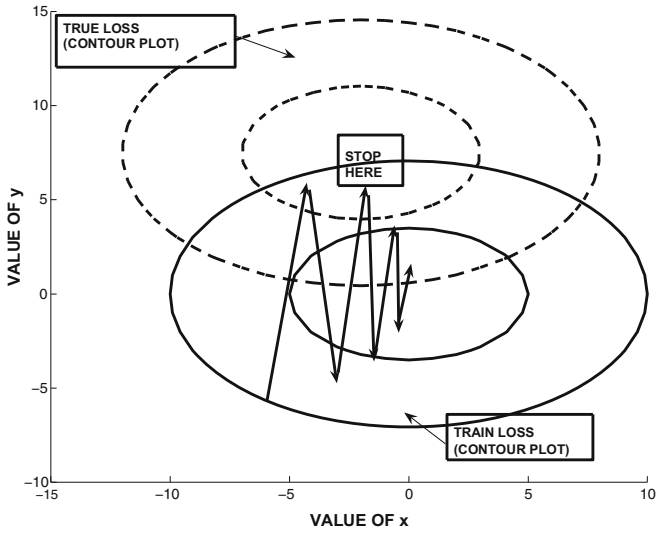


Figure 4.6: Shift in loss function caused by variance effects and the effect of early stopping. Because of the differences in the true loss function and that on the training data, the error will begin to rise if gradient descent is continued beyond a certain point. Here, we have shown a similar shape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

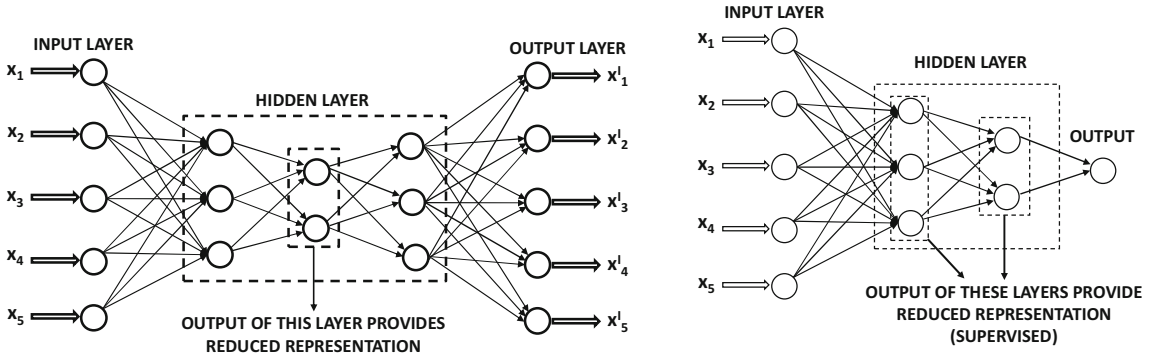
Unfortunately, the learning procedure can perform the gradient-descent only on the loss function defined on the training data set, because the true loss function is unknown. However, if the training data is representative of the true loss function, the optimum solutions in the two cases will be reasonably 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.

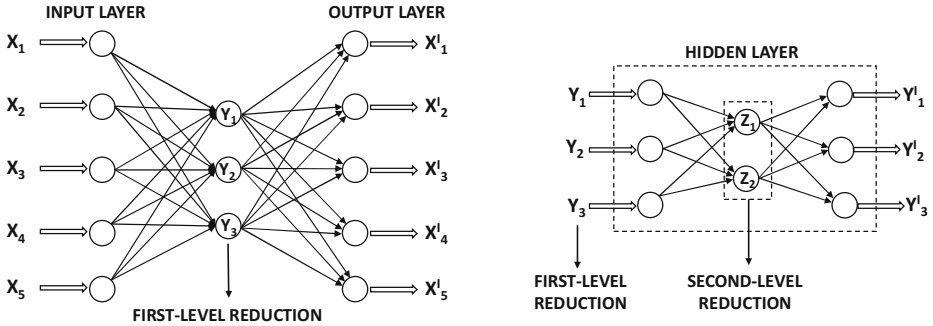
Although the depth of the neural network causes challenges, the problems associated with depth are also heavily dependent on how the network is initialized. A good initialization point can often solve many of the problems associated with reaching good solutions. A ground-breaking break-through in this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

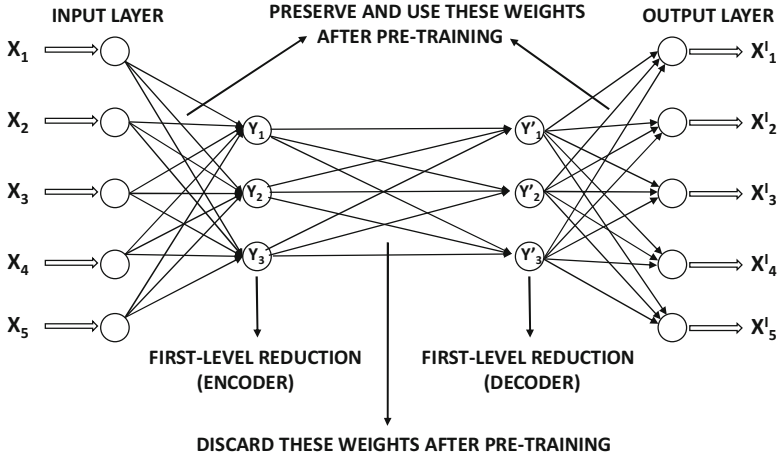


Figure 4.9: This architecture allows the first-level representations in the encoder and decoder to be significantly different. It is helpful to compare this architecture with that in Figure 4.8(a).

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.

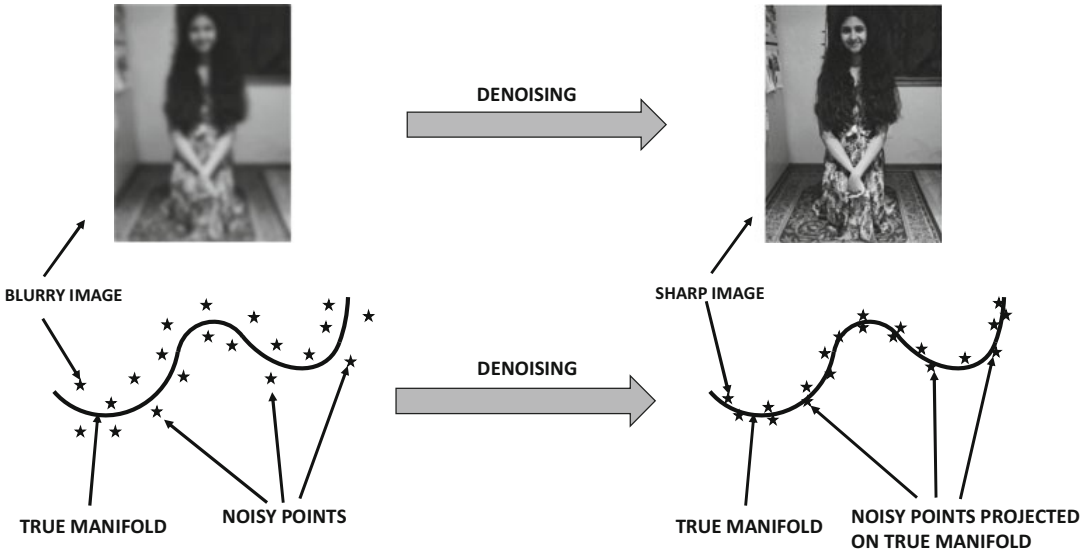


Figure 4.10: The de-noising autoencoder

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.

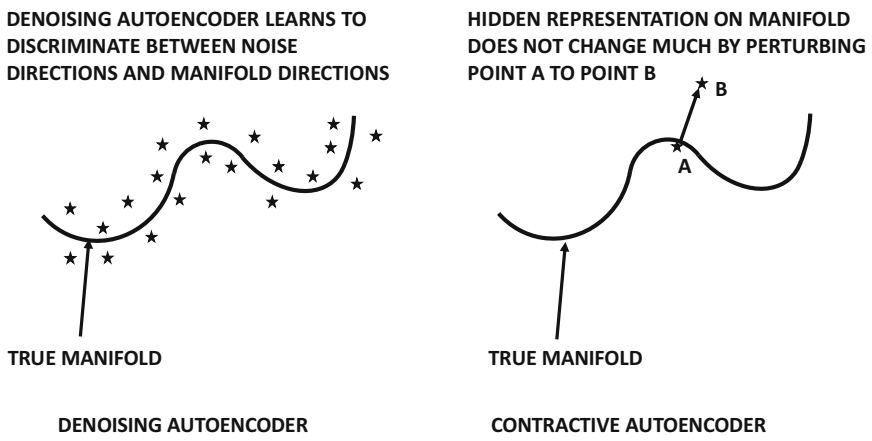


Figure 4.11: The difference between the de-noising and the contractive autoencoder

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:

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

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:

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

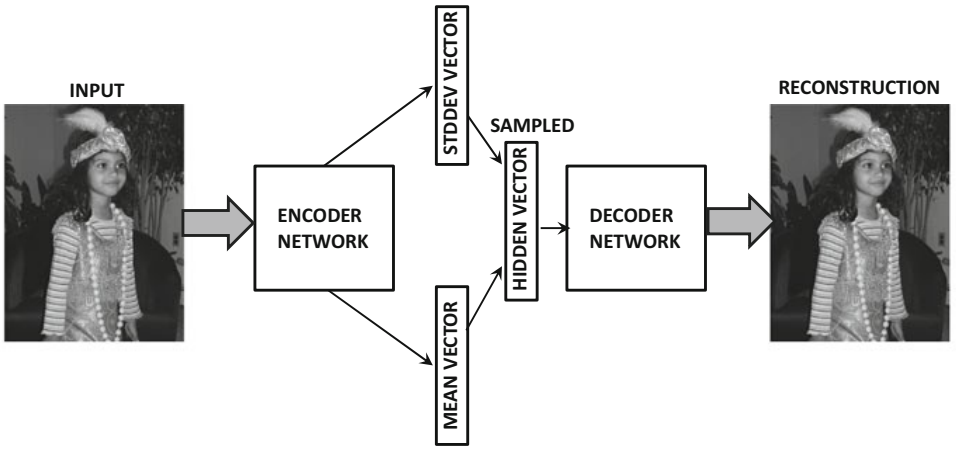
Here, $\lambda > 0$ is the regularization parameter. Small values of λ will favor exact reconstruction, and the approach will behave like a traditional autoencoder. The regularization term forces the hidden representations to be stochastic, so that multiple hidden representations generate almost the 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

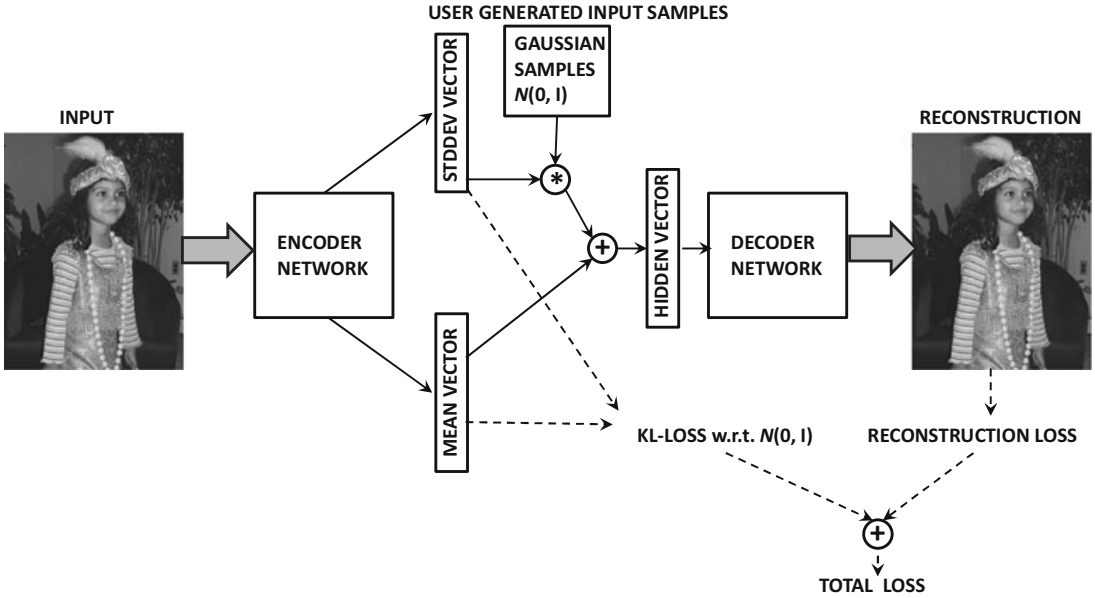
The training of a variational autoencoder is relatively straightforward because the stochasticity has been pulled out as an additional input. One can backpropagate as in any

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

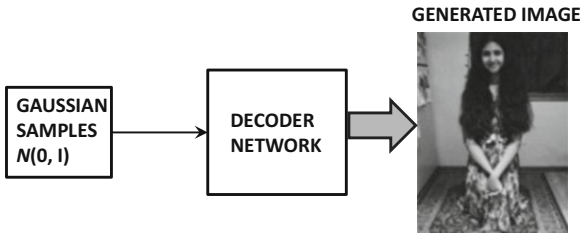


Figure 4.13: Generating samples from the variational autoencoder. The images are illustrative only.

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

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

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

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

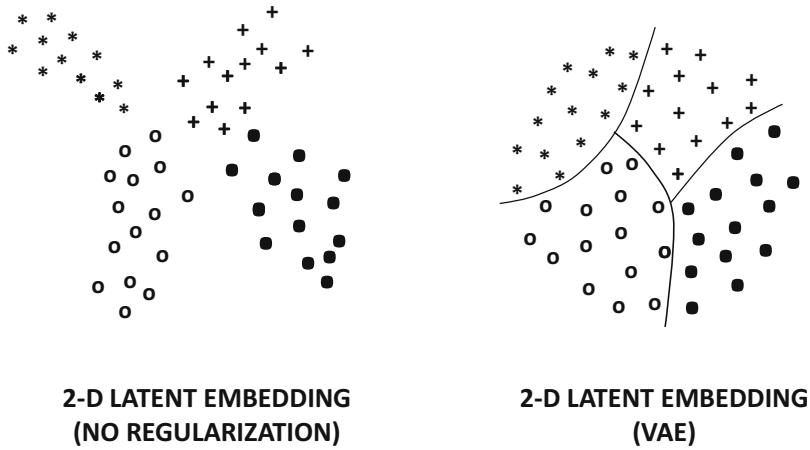


Figure 4.14: Illustrations of the embeddings created by a variational autoencoder in relation to the unregularized version. The unregularized version has large discontinuities in the latent space, which might not correspond to meaningful points. The Gaussian embedding of the points in the variational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

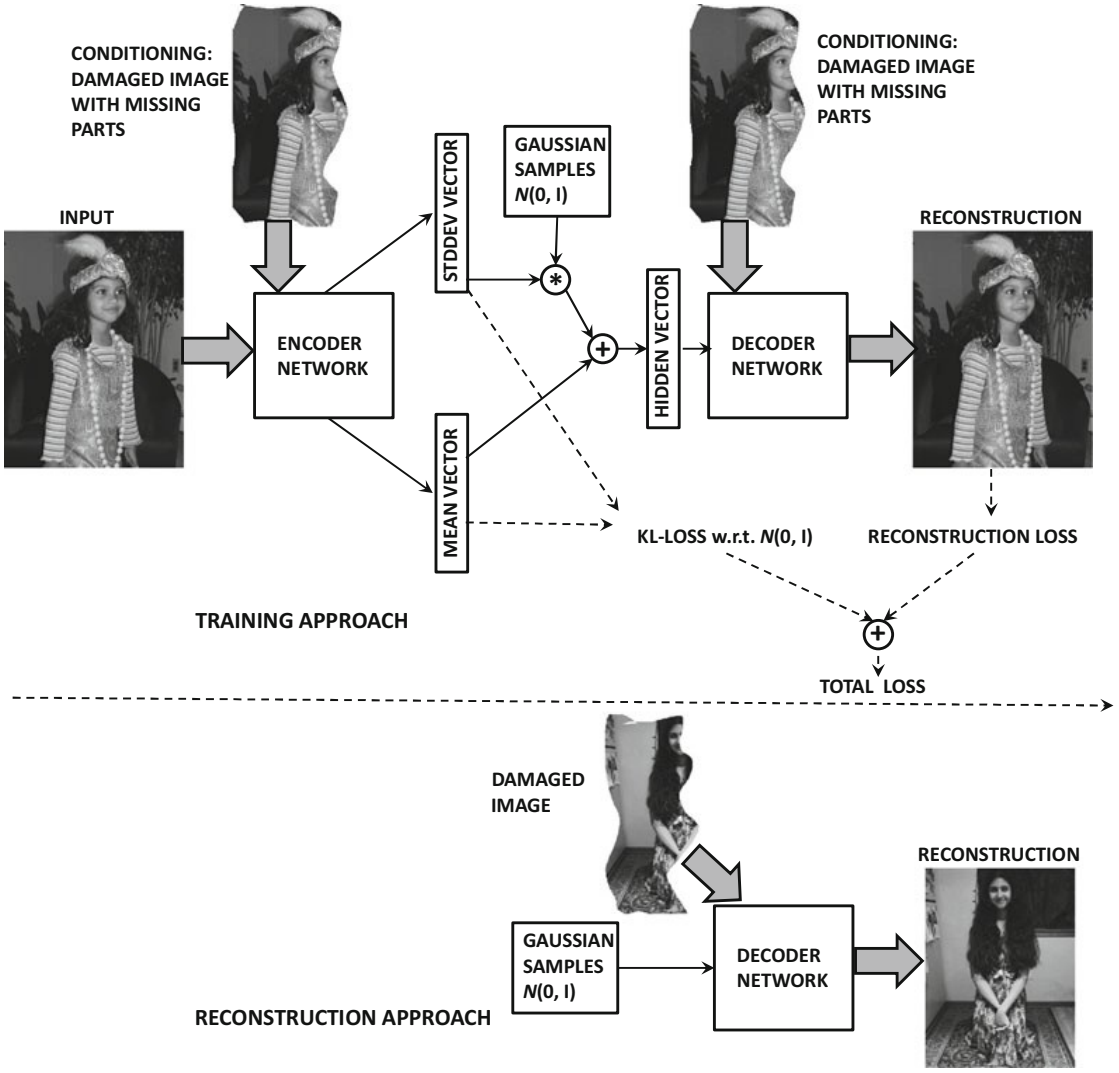


Figure 4.15: Reconstructing damaged images with the conditional variational autoencoder. The images are illustrative only.

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?