2.4 Backpropagated Saliency for Interpretability and Feature Selection

One of the common refrains about neural networks has been their lack of interpretability [97]. However, it turns out that one can use backpropagation in order to determine the features that contribute the most to the classification of a particular test instance. This provides the analyst with an understanding of the relevance of each feature to classification. This approach also has the useful property that it can be used for feature selection [406].

Consider a test instance $\overline{X} = (x_1, \ldots x_d)$, for which the multilabel output scores of the neural network are $o_1 \ldots o_k$. Furthermore, let the output of the winning class among the k outputs be o_m, where $m \in \{1 \ldots k\}$. Our goal is to identify the features that are most relevant to the classification of this test instance. In general, for each attribute x_i, we would like to determine the sensitivity of the output o_m to x_i. Features with large *absolute* magnitudes of this sensitivity are obviously relevant to the classification of this test instance. In order to achieve this goal, we would like to compute the absolute magnitude of $\frac{\partial o_m}{\partial x_i}$. The features with the largest absolute value of the partial derivative have the greatest influence on the classification to the winning class. The sign of this derivative also tells us whether increasing x_i slightly from its current value increases or decreases the score of the winning class. For classes other than the winning class, the derivative also provides some understanding of the sensitivity, but this is less important, particularly when the number of classes is large. The value of $\frac{\partial o_m}{\partial x_i}$ can be computed by a straightforward application of the backpropagation algorithm, in which one does not stop backpropagating at the first hidden layer but applies the process all the way to the input layer.

One can also use this approach for feature selection by aggregating the absolute value of the gradient over all classes and all correctly classified training instances. The features with the largest aggregate sensitivity over the whole training data are the most relevant. Strictly speaking, one does not need to aggregate this value over all classes, but one can simply use only the winning class for correctly classified training instances. However, the original work in [406] aggregates this value over all classes and all instances.

Similar methods for interpreting the effects of different portions of the input are also used in computer vision with convolutional neural networks [466]. A discussion of some of these methods is provided in Section 8.5.1 of Chapter 8. In the case of computer vision, the visual effects of this type of saliency analysis are sometimes spectacular. For example, for an image of a dog, the analysis will tell us which features (i.e., pixels) results in the image being considered a dog. As a result, we can create a black-and-white saliency image in which the portion corresponding to a dog is emphasized in light color against a dark background (cf. Figure 8.12 of Chapter 8).

2.5 Matrix Factorization with Autoencoders

Autoencoders represent a fundamental architecture that is used for various types of unsupervised learning, including matrix factorization, principal component analysis, and dimensionality reduction. Natural architectural variations of the autoencoder can also be used for matrix factorization of incomplete data to create recommender systems. Furthermore, some recent feature engineering methods in the natural language domain like *word2vec* can also be viewed as variations of autoencoders, which perform nonlinear matrix factorizations of word-context matrices. The nonlinearity is achieved with the activation function in the output layer, which is usually not available with traditional matrix factorization. Therefore,

one of our goals will be to demonstrate how small changes to the underlying building blocks of the neural network can be used to implement sophisticated variations of a given family of methods. This is particularly convenient for the analyst, who only has to experiment with small variations of the architecture to test different types of models. Such variations would require more effort to construct in traditional machine learning, because one does not have the benefit of learning abstractions like backpropagation. First, we begin with a simple simulation of a traditional matrix factorization method with a shallow neural architecture. Then, we discuss how this basic setup provides the path to generalizations to nonlinear dimensionality reduction methods by adding layers and/or nonlinear activation functions. Therefore, the goal of this section is to show two things:

1. Classical dimensionality reduction methods like singular value decomposition and principal component analysis are special cases of neural architectures.

2. By adding different types of complexities to the basic architecture, one can generate complex nonlinear embeddings of the data. While nonlinear embeddings are also available in machine learning, neural architectures provide unprecedented flexibility in controlling the properties of the embedding by making various types of architectural changes (and allowing backpropagation to take care of the changes in the underlying learning algorithms).

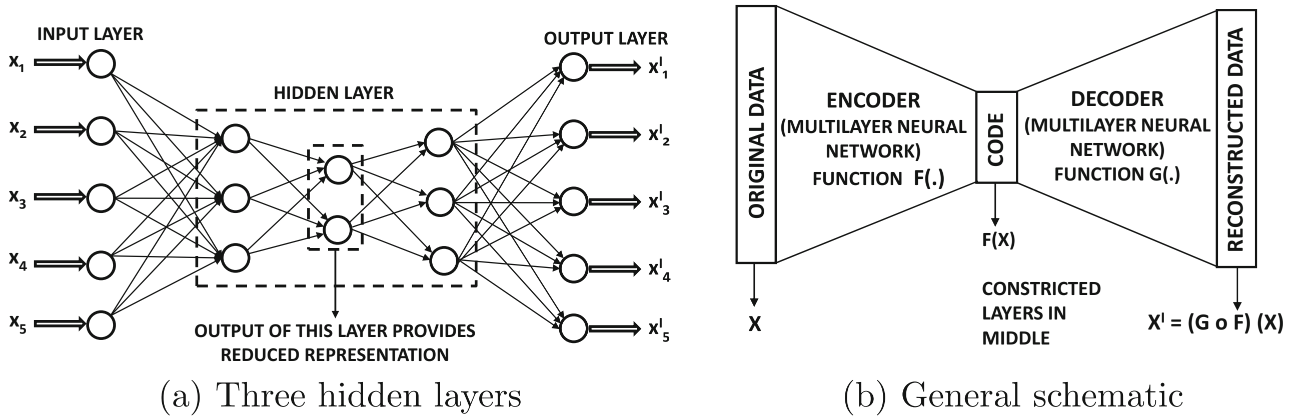
We will also discuss a number of applications such as recommender systems and outlier detection.

2.5.1 Autoencoder: Basic Principles

The basic idea of an autoencoder is to have an output layer with the same dimensionality as the inputs. The idea is to try to reconstruct each dimension exactly by passing it through the network. An autoencoder *replicates* the data from the input to the output, and is therefore sometimes referred to as a *replicator neural network*. Although reconstructing the data might seem like a trivial matter by simply copying the data forward from one layer to another, this is not possible when the number of units in the middle are *constricted*. In other words, the number of units in each middle layer is typically fewer than that in the input (or output). As a result, these units hold a reduced representation of the data, and the final layer can no longer reconstruct the data exactly. Therefore, this type of reconstruction is inherently *lossy*. The loss function of this neural network uses the sum-of-squared differences between the input and the output in order to force the output to be as similar as possible to the input. This general representation of the autoencoder is given in Figure 2.6(a), where an architecture is shown with three constricted layers. It is noteworthy that the representation of the innermost hidden layer will be hierarchically related to those in the two outer hidden layers. Therefore, an autoencoder is capable of performing hierarchical data reduction.

It is common (but not necessary) for an M-layer autoencoder to have a symmetric architecture between the input and output, where the number of units in the kth layer is the same as that in the $(M - k + 1)$th layer. Furthermore, the value of M is often odd, as a result of which the $(M + 1)/2$th layer is often the most constricted layer. Here, we are counting the (non-computational) input layer as the first layer, and therefore the minimum number of layers in an autoencoder would be three, corresponding to the input layer, constricted layer, and the output layer. As we will see later, this simplest form of the autoencoder is used in traditional machine learning for singular value decomposition. The symmetry in the architecture often extends to the fact that the weights outgoing from the

kth layer are tied to those incoming to the $(M-k)$th layer in many architectures. For now, we will not make this assumption for simplicity in presentation. Furthermore, the symmetry is never absolute because of the effect of nonlinear activation functions. For example, if a nonlinear activation function is used in the output layer, there is no way to symmetrically mirror that fact in the (non-computational) input layer.



(a) Three hidden layers

(b) General schematic

Figure 2.6: The basic schematic of the autoencoder

The reduced representation of the data is also sometimes referred to as the *code*, and the number of units in this layer is the dimensionality of the reduction. The initial part of the neural architecture before the bottleneck is referred to as the *encoder* (because it creates a reduced code), and the final part of the architecture is referred to as the *decoder* (because it reconstructs from the code). The general schematic of the autoencoder is shown in Figure 2.6(b).

2.5.1.1 Autoencoder with a Single Hidden Layer

In the following, we describe the simplest version of an autoencoder, which is used for matrix factorization. This autoencoder only has a single hidden layer of $k \ll d$ units between the input and output layers of d units each. For the purpose of discussion, assume that we have an $n \times d$ matrix denoted by D, which we would like to factorize into an $n \times k$ matrix U and a $d \times k$ matrix V:

$$D \approx UV^T \tag{2.38}$$

Here, k is the rank of the factorization. The matrix U contains the reduced representation of the data, and the matrix V contains the basis vectors. Matrix factorization is one of the most widely studied problems in supervised learning, and it is used for dimensionality reduction, clustering, and predictive modeling in recommender systems.

In traditional machine learning, this problem is solved by minimizing the *Frobenius norm* of the *residual matrix* denoted by $(D - UV^T)$. The squared Frobenius norm of a matrix is the sum of the squares of the entries in the matrix. Therefore, one can write the objective function of the optimization problem as follows:

$$\text{Minimize } J = ||D - UV^T||_F^2$$

Here, the notation $|| \cdot ||_F$ indicates the Frobenius norm. The parameter matrices U and V need to be learned in order to optimize the aforementioned error. This objective function has an infinite number of optima, one of which has mutually orthogonal basis vectors. That particular solution is referred to as *truncated singular value decomposition*. Although it is relatively easy to derive the gradient-descent steps [6] for this optimization problem

(without worrying about neural networks at all), our goal here is to capture this optimization problem within a neural architecture. Going through this exercise helps us show that SVD is a special case of an autoencoder architecture, which sets the stage for understanding the gains obtained with more complex autoencoders.

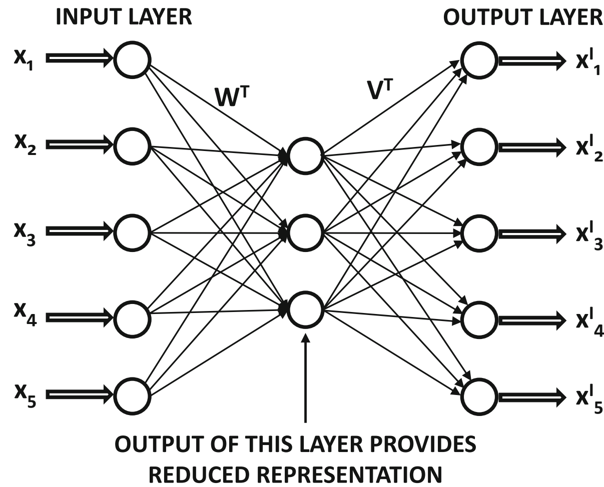


Figure 2.7: A basic autoencoder with a single layer

This neural architecture for SVD is illustrated in Figure 2.7, where the hidden layer contains k units. The rows of D are input into the autoencoder, whereas the k-dimensional rows of U are the activations of the hidden layer. The $k \times d$ matrix of weights in the decoder is V^T. As we discussed in the introduction to the multilayer neural network in Chapter 1, the vector of values in a particular layer of the network can be obtained by multiplying the vector of values in the previous layer with the matrix of weights connecting the two layers (with linear activation). Since the activations of the hidden layer are U and the decoder weights contain the matrix V^T, it follows that the reconstructed output contains the rows of UV^T. The autoencoder minimizes the sum-of-squared differences between the input and the output, which is equivalent to minimizing $||D - UV^T||^2$. Therefore, the same problem is being solved as singular value decomposition.

Note that one can use this approach to provide the reduced representation of *out-of-sample* instances that were not included in the original matrix D. One simply has to feed these out-of-sample rows as the input, and the activations of the hidden layer will provide the reduced representation. Reducing out-of-sample instances is particularly useful for nonlinear dimensionality-reduction methods, as it is more difficult for traditional machine learning methods to fold in new instances.

Encoder Weights

As shown in Figure 2.7, the encoder weights are contained in the $k \times d$ matrix denoted by W. How is this matrix related to U and V? Note that the autoencoder creates the reconstructed representation DW^TV^T of the original data matrix. Therefore, it tries to optimize the problem of minimizing $||DW^TV^T - D||^2$. The optimal solution to this problem is obtained when the matrix W contains the *pseudo-inverse* of V, which is defined as follows:

$$W = (V^TV)^{-1}V^T \tag{2.39}$$

This result is easy to show at least for non-degenerate cases in which the rows of matrix D span the full rank of d dimensions (see Exercise 14). Of course, the final solution found by the training algorithm of the autoencoder might deviate from this condition because it might not solve the problem precisely or because the matrix D might be of smaller rank.

By the definition of the pseudo-inverse, it follows that $WV = I$ and $V^T W^T = I$, where I is a $k \times k$ identity matrix. Post-multiplying Equation 2.38 with W^T we obtain the following:

$$DW^T \approx U \underbrace{(V^T W^T)}_{I} = U \tag{2.40}$$

In other words, multiplying each row of the matrix D with the $d \times k$ matrix W^T yields the reduced representation of that instance, which is the corresponding row in U. Furthermore, multiplying that row of U again with V^T yields the reconstructed version of the original data matrix D.

Note that there are many alternate optima for W and V, but in order for reconstruction to occur (i.e., minimization of loss function), the learned matrix W will always be (approximately) related to V as its pseudo-inverse and the columns of V will always span[3] a particular k-dimensional subspace defined by the SVD optimization problem.

2.5.1.2 Connections with Singular Value Decomposition

The single-layer autoencoder architecture is closely connected with singular value decomposition (SVD). Singular value decomposition finds a factorization UV^T in which the columns of V are orthonormal. The loss function of this neural network is identical to that of singular value decomposition, and a solution V in which the columns of V are orthonormal will always be one of the *possible* optima obtained by training the neural network. However, since this loss function allows alternative optima, it is possible to find an optimal solution in which the columns of V are not necessarily mutually orthogonal or scaled to unit norm. SVD is defined by an orthonormal basis system. Nevertheless, the subspace spanned by the k columns of V will be the same as that spanned by the top-k basis vectors of SVD. Principal component analysis is identical to singular value decomposition, except that it is applied to a mean-centered matrix D. Therefore, the approach can also be used to find the subspace spanned by the top-k principal components. However, each column of D needs to be mean-centered up front by subtracting its mean. One can achieve an orthonormal basis system, which is even closer to SVD and PCA by sharing some of the weights in the encoder and decoder. This approach is discussed in the next section.

2.5.1.3 Sharing Weights in Encoder and Decoder

There are many possible alternate solutions for W and V in the above discussion, in which W is the pseudo-inverse of V. One can, therefore, reduce the parameter footprint further without significant[4] loss in reconstruction accuracy. A common practice that is used in the autoencoder construction is to share some of the weights between the encoder and the

[3]This subspace is defined by the top-k singular vectors of singular value decomposition. However, the optimization problem does not impose orthogonality constraints, and therefore the columns of V might use a different non-orthogonal basis system to represent this subspace.

[4]There is no loss in reconstruction accuracy in several special cases like the single-layer case discussed here, even on the training data. In other cases, the loss of accuracy is only on the training data, but the autoencoder tends to better reconstruct out-of-sample data because of the regularization effects of parameter footprint reduction.

decoder. This is also referred to as *tying the weights*. In particular, the autoencoder has an inherently symmetric structure, in which the weights of the encoder and decoder are forced to be the same in symmetrically matching layers. In the shallow case, the encoder and decoder weights are shared by using the following relationship:

$$W = V^T \tag{2.41}$$

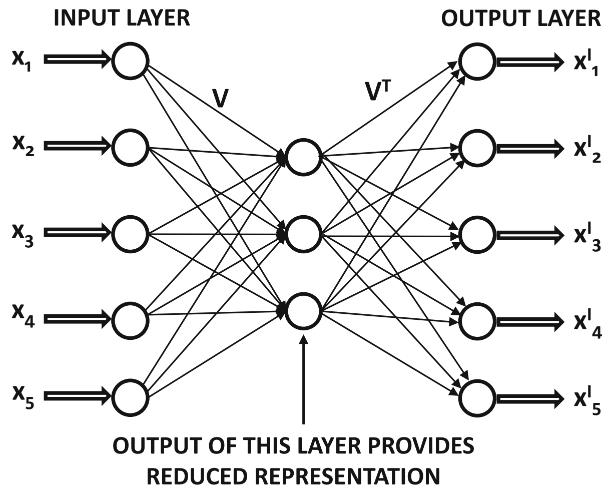


Figure 2.8: Basic autoencoder with a single layer; note tied weights (unlike the autoencoder shown in Figure 2.7).

This architecture is shown in Figure 2.8, and it is identical to the architecture of Figure 2.7 except for the presence of tied weights. In other words, the $d \times k$ matrix V of weights is first used to transform the d-dimensional data point $\overline{X}$ into a k-dimensional representation. Then, the matrix V^T of weights is used to reconstruct the data to its original representation.

The tying of the weights effectively means that V^T is the pseudo-inverse of V (see Exercise 14). In other words, we have $V^T V = I$, and therefore the columns of V are mutually orthogonal. As a result, by tying the weights, it is now possible to *exactly* simulate SVD, in which the different basis vectors need to be mutually orthogonal.

In this particular example of an architecture with a single hidden layer, the tying of weights is done only for a pair of weight matrices. In general, one would have an odd number of hidden layers and an even number of weight matrices. It is a common practice to match up the weight matrices in a symmetric way about the middle. In such a case, the symmetrically arranged hidden layers would need to have the same numbers of units. Even though it is not necessary to share weights between the encoder and decoder portions of the architecture, it reduces the number of parameters by a factor of 2. This is beneficial from the point of view of reducing overfitting. In other words, the approach would better reconstruct out-of-sample data. Another benefit of tying the weight matrices in the encoder and the decoder is that it automatically normalizes the columns of V to similar values. For example, if we do not tie the weight matrices in the encoder and the decoder, it is possible for the different columns of V to have very different norms. At least in the case of linear activations, tying the weight matrices forces all columns of V to have similar norms. This is also useful from the perspective of providing better normalization of the embedded representation. The normalization and orthogonality properties no longer hold exactly when nonlinear activations are used in the computational layers. However, there are considerable benefits in tying the weights even in these cases in terms of better conditioning of the solution.

The sharing of weights does require some changes to the backpropagation algorithm during training. However, these modifications are not very difficult. All that one has to do is to perform normal backpropagation by pretending that the weights are not tied in order to compute the gradients. Then, the gradients across different copies of the same weight are added in order to compute the gradient-descent steps. The logic for handing shared weights in this way is discussed in Section 3.2.9 of Chapter 3.

2.5.1.4 Other Matrix Factorization Methods

It is possible to modify the simple three-layer autoencoder to simulate other types of matrix factorization methods such as non-negative matrix factorization, probabilistic latent semantic analysis, and logistic matrix factorization methods. Different methods for logistic matrix factorization will be discussed in the next section, in Section 2.6.3, and in Exercise 8. Methods for non-negative matrix factorization and probabilistic latent semantic analysis are discussed in Exercises 9 and 10. It is instructive to examine the relationships between these different variations, because it shows how one can vary on simple neural architectures in order to get results with vastly different properties.

2.5.2 Nonlinear Activations

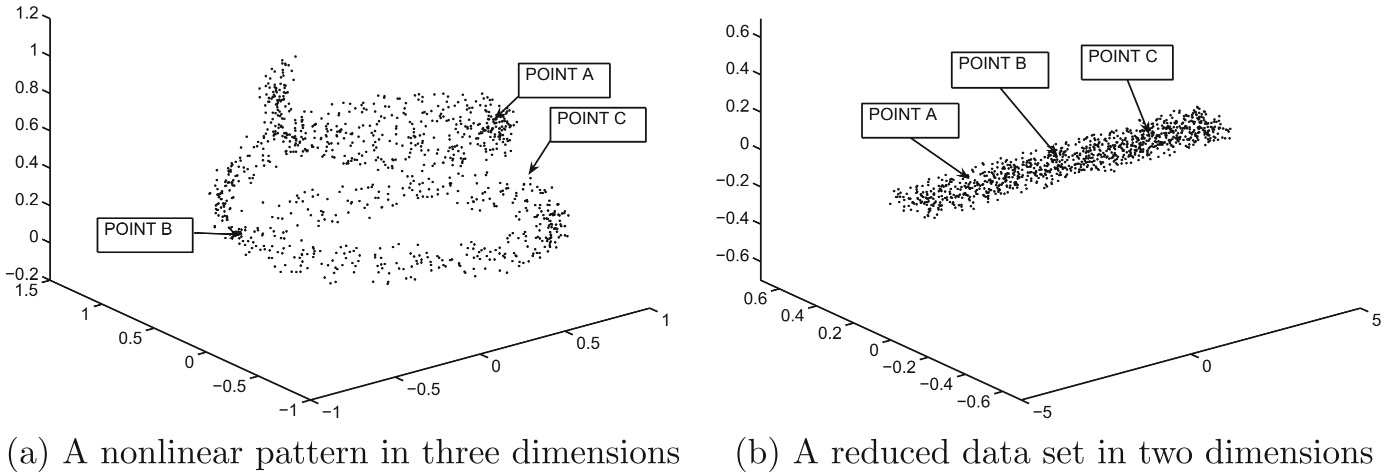
So far, the discussion has focussed on simulating singular value decomposition using a neural architecture. Clearly, this does not seem to achieve much because many off-the-shelf tools exist for singular value decomposition. However, the real power of autoencoders is realized when one starts using nonlinear activations and multiple layers. For example, consider a situation in which the matrix D is binary. In such a case, one can use the same neural architecture as shown in Figure 2.7, but one can also use a sigmoid function in the final layer to predict the output. This sigmoid layer is combined with negative log loss. Therefore, for a binary matrix $B = [b_{ij}]$, the model assumes the following:

$$B \sim \text{sigmoid}(UV^T) \tag{2.42}$$

Here, the sigmoid function is applied in element-wise fashion. Note the use of $\sim$ instead of $\approx$ in the above expression, which indicates that the binary matrix B is an instantiation of random draws from Bernoulli distributions with corresponding parameters contained in sigmoid(UV^T). The resulting factorization can be shown to be equivalent to *logistic matrix factorization*. The basic idea is that the (i, j)th element of UV^T is the parameter of a Bernoulli distribution, and the binary entry b_{ij} is generated from a Bernoulli distribution with these parameters. Therefore, U and V are learned using the log-likelihood loss of this *generative* model. The log-likelihood loss implicitly tries to find parameter matrices U and V so that the probability of the matrix B being generated by these parameters is maximized.

Logistic matrix factorization has only recently been proposed [224] as a sophisticated matrix factorization method for binary data, which is useful for recommender systems with *implicit feedback* ratings. Implicit feedback refers to the binary actions of users such as buying or not buying specific items. The solution methodology of this recent work on logistic matrix factorization [224] seems to be vastly different from SVD, and it is not based on a neural network approach. However, for a neural network practitioner, the change from the SVD model to that of logistic matrix factorization is a relatively small one, where only the final layer of the neural network needs to be changed. It is this modular nature of neural networks that makes them so attractive to engineers and encourages all types of experimentation. In fact, one of the variants of the popular *word2vec* neural approach [325, 327]

for text feature engineering is a logistic matrix factorization method, when one examines it more closely. Interestingly, *word2vec* was proposed earlier than logistic matrix factorization in traditional machine learning [224], although the equivalence of the two methods was not shown in the original work. The equivalence was first shown in [6], and a proof of this result is also provided later in this chapter. Indeed, for multilayer variants of the autoencoder,



(a) A nonlinear pattern in three dimensions (b) A reduced data set in two dimensions

Figure 2.9: The effect of nonlinear dimensionality reduction. This figure is drawn for illustrative purposes only.

an exact counterpart does not even exist in traditional machine learning. All this seems to suggest that it is often more natural to discover sophisticated machine learning algorithms when working with the modular approach of constructing multilayer neural networks. Note that one can even use this approach to factorize real-valued matrix entries drawn from $[0, 1]$, as long as the log-loss is suitably modified to handle fractional values (see Exercise 8). Logistic matrix factorization is a type of *kernel matrix factorization*.

One can also use non-linear activations in the hidden layer rather than (or in addition to) the output layer. By using the non-linearity in the hidden layer to impose non-negativity, one can simulate non-negative matrix factorization (cf. Exercises 9 and 10). Furthermore, consider an autoencoder with a single hidden layer in which sigmoid units are used in the hidden layer, and the output layer is linear. Furthermore, the input-to-hidden and the hidden-to-output matrices are denoted by W^T and V^T, respectively. In this case, the matrix W will no longer be the pseudo-inverse of V because of the non-linear activation in the hidden layer.

If U is the output of the hidden layer in which the nonlinear activation $\Phi(\cdot)$ is applied, we have:

$$U = \Phi(DW^T) \tag{2.43}$$

If the output layer is linear, the overall factorization is still of the following form:

$$D \approx UV^T \tag{2.44}$$

Note, however, that we can write $U' = DW^T$, which is a linear projection of the original matrix D. Then, the factorization can be written as follows:

$$D \approx \Phi(U')V^T \tag{2.45}$$

Here, U' is a linear projection of D. This is a different type of nonlinear matrix factorization [521, 558]. Although the specific form of the nonlinearity (e.g., sigmoid) might seem

simplistic compared to what is considered typical in kernel methods, in reality multiple hidden layers are used to learn more complex forms of nonlinear dimensionality reduction. Nonlinearity can also be combined in the hidden layers and in the output layer. Nonlinear dimensionality reduction methods can map the data into much lower dimensional spaces

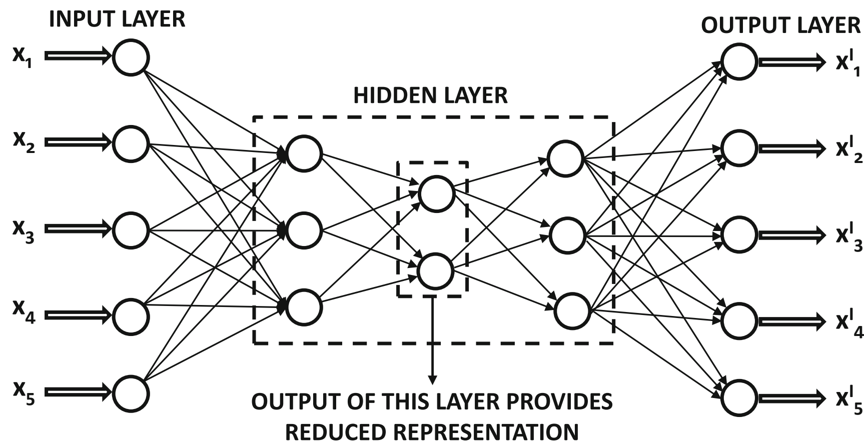


Figure 2.10: An example of an autoencoder with three hidden layers. Combining nonlinear activations with multiple hidden layers increases the representation power of the network.

(with good reconstruction characteristics) than would be possible with methods like PCA. An example of a data set, which is distributed on a nonlinear spiral, is shown in Figure 2.9(a). This data set cannot be reduced to lower dimensionality using PCA (without causing significant reconstruction error). However, the use of nonlinear dimensionality reduction methods can flatten out the nonlinear spiral into a 2-dimensional representation. This representation is shown in Figure 2.9(b).

Nonlinear dimensionality-reduction methods often require deeper networks due to the more complex transformations possible with the combination of nonlinear units. The benefits of depth will be discussed in the next section.

2.5.3 Deep Autoencoders

The real power of autoencoders in the neural network domain is realized when deeper variants are used. For example, an autoencoder with three hidden layers is shown in Figure 2.10. One can increase the number of intermediate layers in order to further increase the representation power of the neural network. It is noteworthy that it is essential for some of the layers of the deep autoencoder to use a nonlinear activation function to increase its representation power. As shown in Lemma 1.5.1 of Chapter 1, no additional power is gained by a multilayer network when only linear activations are used. Although this result was shown in Chapter 1 for the classification problem, it is broadly true for any type of multilayer neural network (including an autoencoder).

Deep networks with multiple layers provide an extraordinary amount of representation power. The multiple layers of this network provide *hierarchically* reduced representations of the data. For some data domains like images, hierarchically reduced representations are particularly natural. Note that there is no precise analog of this type of model in traditional machine learning, and the backpropagation approach rescues us from the challenges associated in computing the complicated gradient-descent steps. A nonlinear dimensionality reduction might map a manifold of arbitrary shape into a reduced representation. Although several methods for nonlinear dimensionality reduction are known in machine learning, neural networks have some advantages over these methods:

1. Many nonlinear dimensionality reduction methods have a very hard time mapping out-of-sample data points to reduced representations, unless these points are included in the training data up front. On the other hand, it is a relatively simple matter to

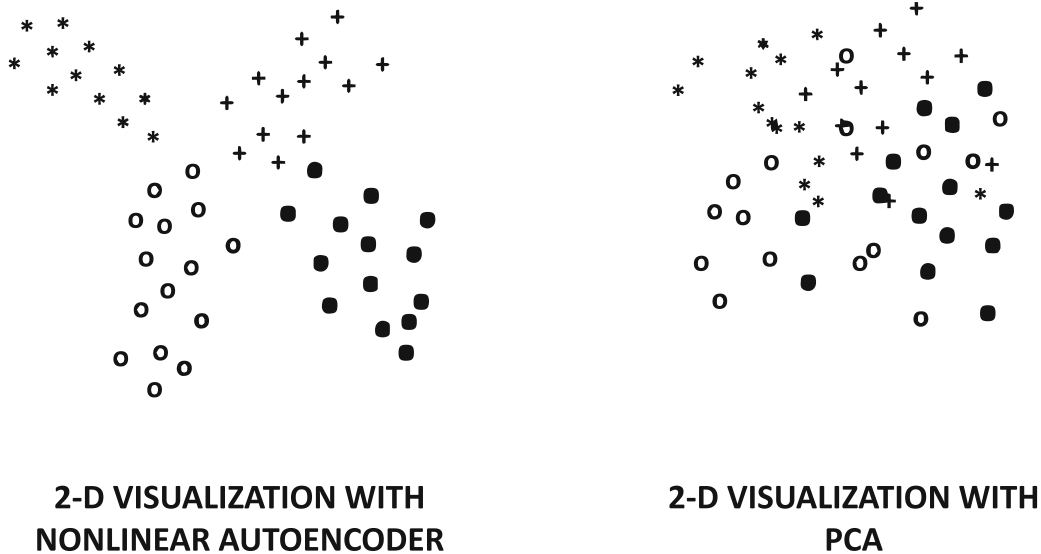


Figure 2.11: A depiction of the typical difference between the embeddings created by nonlinear autoencoders and principal component analysis (PCA). Nonlinear and deep autoencoders are often able to separate out the entangled class structures in the underlying data, which is not possible within the constraints of linear transformations like PCA. This occurs because individual classes are often populated on curved manifolds in the original space, which would appear mixed when looking at a data in any 2-dimensional cross-section unless one is willing to warp the space itself. The figure above is drawn for illustrative purposes only and does not represent a specific data set.

   compute the reduced representation of an out-of-sample point by passing it through the network.

2. Neural networks allow more power and flexibility in the nonlinear data reduction by varying on the number and type of layers used in intermediate stages. Furthermore, by choosing specific types of activation functions in particular layers, one can engineer the nature of the reduction to the properties of the data. For example, it makes sense to use a logistic output layer with logarithmic loss for a binary data set.

It is possible to achieve extraordinarily compact reductions by using this approach. For example, the work in [198] shows how one can convert a 784-dimensional representation of the pixels of an image into a 6-dimensional reduction with the use of deep autoencoders. Greater reduction is always achieved by using nonlinear units, which implicitly map warped manifolds into linear hyperplanes. The superior reduction in these cases is because it is easier to thread a warped surface (as opposed to a linear surface) through a larger number of points. This property of nonlinear autoencoders is often used for 2-dimensional visualizations of the data by creating a deep autoencoder in which the most compact hidden layer has only two dimensions. These two dimensions can then be mapped on a plane to visualize the points. In many cases, the class structure of the data is exposed in terms of well-separated clusters.

An illustrative example of the typical behavior of real data distributions is shown in Figure 2.11, in which the 2-dimensional mapping created by a deep autoencoder seems to clearly separate out the different classes. On the other hand, the mapping created by PCA does not seem to separate the classes well. Figure 2.9, which provides a nonlinear

spiral mapped to a linear hyperplane, clarifies the reason for this behavior. In many cases, the data may contain heavily entangled spirals (or other shapes) that belong to different classes. Linear dimensionality reduction methods cannot attain clear separation because nonlinearly entangled shapes are not linearly separable. On the other hand, deep autoencoders with nonlinearity are far more powerful and able to disentangle such shapes. Deep autoencoders can sometimes be used as alternatives to other robust visualization methods like t-distributed stochastic neighbor embedding (t-SNE) [305]. Although t-SNE can often provide better performance[5] for visualization (because it is specifically designed for visualization rather than dimensionality reduction), the advantage of an autoencoder over t-SNE is that it is easier to generalize to out-of-sample data. When new data points are received, they can simply be passed through the encoder portion of the autoencoder in order to add them to the current set of visualized points. A specific example of a visualization of a high-dimensional document collection with an autoencoder is provided in [198].

It is, however, possible to go too far and create representations that are not useful. For example, one can compress a very high-dimensional data point into a single dimension, which reconstructs a point from the training data very well but gives high reconstruction error for test data. In other words, the neural network has found a way to memorize the data set without sufficient ability to create reduced representations of unseen points. Therefore, even for unsupervised problems like dimensionality reduction, it is important to keep aside some points as a *validation set*. The points in the validation set are not used during training. One can then quantify the difference in reconstruction error between the training and validation data. Large differences in reconstruction error between the training and validation data are indicative of overfitting. Another issue is that deep networks are harder to train, and therefore tricks like *pretraining* are important. These tricks will be discussed in Chapters 3 and 4.

2.5.4 Application to Outlier Detection

Dimensionality reduction is closely related to outlier detection, because outlier points are hard to encode and decode without losing substantial information. It is a well-known fact that if a matrix D is factorized as $D \approx D' = UV^T$, then the low-rank matrix D' is a de-noised representative of the data. After all, the compressed representation U captures only the regularities in the data, and is unable to capture the unusual variations in specific points. As a result, reconstruction to D' misses all these unusual variations.

The absolute values of the entries of $(D - D')$ represent the outlier scores of the matrix entries. Therefore, one can use this approach to find outlier entries, or add the squared scores of the entries in each row of D to find the outlier score of that row. Therefore, one can identify outlier data points. Furthermore, by adding the squared scores in each column of D, one can find outlier features. This is useful for applications like feature selection in clustering, where a feature with a large outlier score can be removed because it adds noise to the clustering. Although we have provided the description above with the use of matrix factorization, any type of autoencoder can be used. In fact, the construction of de-noising autoencoders is a vibrant field in its own right. Refer to the bibliographic notes.

[5] The t-SNE method works on the principle is that it is impossible to preserve all pairwise similarities and dissimilarities with the same level of accuracy in a low-dimensional embedding. Therefore, unlike dimensionality reduction or autoencoders that try to faithfully reconstruct the data, it has an asymmetric loss function in terms of how similarity is treated versus dissimilarity. This type of asymmetric loss function is particularly helpful for separating out different manifolds during visualization. Therefore, t-SNE might perform better than autoencoders at visualization.

2.5.5 When the Hidden Layer Is Broader than the Input Layer

So far, we have only discussed cases in which the hidden layer has fewer units than the input layer. It makes sense for the hidden layer to have fewer units than the input layer when one is looking for a compressed representation of the data. A constricted hidden layer forces dimensionality reduction, and the loss function is designed to avoid information loss. Such representations are referred to as *undercomplete representations*, and they correspond to the traditional use-case of autoencoders.

What about the case when the number of hidden units is greater than the input dimensionality? This situation corresponds to the case of *over-complete representations*. Increasing the number of hidden units beyond the number of input units makes it possible for the hidden layer to simply learn the identity function (with zero loss). Simply copying the input across the layers does not seem to be particularly useful. However, this does not occur in practice (while learning weights), especially if certain types of regularization and *sparsity constraints* are imposed on the hidden layer. Even if no sparsity constraints are imposed, and stochastic gradient descent is used for learning, the probabilistic regularization caused by stochastic gradient descent is sufficient to ensure that the hidden representation will always scramble the input before reconstructing it at the output. This is because stochastic gradient descent is a type of noise addition to the learning process, and therefore it will not be possible to learn weights that simply copy input to output as identity functions across layers. Furthermore, because of some peculiarities of the training process, a neural network almost never uses its full modeling ability, which leads to dependencies among the weights [94]. Rather, an over-complete representation may be created, although it may not have the property of sparsity (which needs to be explicitly encouraged). The next section will discuss ways of encouraging sparsity.

2.5.5.1 Sparse Feature Learning

When explicit sparsity constraints are imposed, the resulting autoencoder is referred to as a *sparse autoencoder*. A sparse representation of a d-dimensional point is a k-dimensional point in which $k \gg d$ and most of the values in the sparse representation are 0s. Sparse feature learning has tremendous applicability to many settings like image data, where the learned features are often intuitively more interpretable from an application-specific perspective. Furthermore, points with a variable amount of information will be naturally represented by having varying numbers of nonzero feature values. This type of property is naturally true in some *input* representations like documents; documents with more information will have more non-zero features (word frequencies) when represented in multidimensional format. However, if the available input is not sparse to begin with, there are often benefits in creating a sparse transformation where such a flexibility of representation exists. Sparse representations also enable the effective use of particular types of efficient algorithms that are highly dependent on sparsity. There are many ways in which constraints might be enforced on the hidden layer to create sparsity. One approach is to add biases to the hidden layer, so that many units are encouraged to be zeros. Some examples are as follows:

1. One can impose an L_1-penalty on the activations in the hidden layer to force sparse activations. The notion of L_1-penalties for creating sparse solutions (in terms of either weights or hidden units) is discussed in Sections 4.4.2 and 4.4.4 of Chapter 4. In such a case, backpropagation must also propagate the gradient of this penalty in the backwards direction. Surprisingly, this natural alternative is rarely used.

2. One can allow only the top-r activations in the hidden layer to be nonzero for $r \leq k$. In such a case, backpropagation only backpropagates through the activated units. This approach is referred to as the r-sparse autoencoder [309].

3. Another approach is the *winner-take-all* autoencoder [310], in which only a fraction f of the activations of *each* hidden unit are allowed over the *whole training data*. In this case, the top activations are computed across training examples, whereas in the previous case the top activations are computed across a hidden layer for a single training example. Therefore node-specific thresholds need to be estimated using the statistics of a minibatch. The backpropagation algorithm needs to propagate the gradient only through the activated units.

Note that the implementations of the competitive mechanisms are almost like ReLU activations with adaptive thresholds. Refer to the bibliographic notes for pointers and more details of these algorithms.

2.5.6 Other Applications

Autoencoders form the workhorse of unsupervised learning in the neural network domain. They are used for a host of applications, which will be discussed later in the book. After training an autoencoder, it is not necessary to use both the encoder and decoder portions. For example, when using the approach for dimensionality reduction, one can use the encoder portion in order to create the reduced representations of the data. The reconstructions of the decoder might not be required at all.

Although an autoencoder naturally removes noise (like almost any dimensionality reduction method), one can enhance the ability of the autoencoder to remove specific types of noise. To perform the training of a *de-noising autoencoder*, a special type of training is used. First, some noise is added to the training data before passing it through the neural network. The distribution of the added noise reflects the analyst's understanding of the natural types of noise in that particular data domain. However, the loss is computed with respect to the *original* training data instances rather than their corrupted versions. The original training data are relatively clean, although one expects the test instances to be corrupted. Therefore, the autoencoder learns to recover clean representations from corrupted data. A common approach to add noise is to randomly set a fraction f of the inputs to zeros [506]. This approach is especially effective when the inputs are binary. The value of f regulates the level of corruption in the inputs. One can either fix f or even allow f to randomly vary over different training instances. In some cases, when the input is real-valued, Gaussian noise is also used. More details of the de-noising autoencoder are provided in Section 4.10.2 of Chapter 4. A closely related autoencoder is the *contractive autoencoder*, which is discussed in Section 4.10.3.

Another interesting application of the autoencoder is one in which we use only the *decoder* portion of the network to create artistic renderings. This idea is based on the notion of *variational autoencoders* [242, 399], in which the loss function is modified to impose a specific structure on the hidden layer. For example, one might add a term to the loss function to enforce the fact that the hidden variables are drawn from a Gaussian distribution. Then, one might repeatedly draw samples from this Gaussian distribution and use only the decoder portion of the network in order to generate samples of the original data. The generated samples often represent realistic samples from the original data distribution.

A closely related model is that of *generative adversarial networks*, which have become increasingly popular in recent years. These models pair the learning of a decoding network

with that of an adversarial discriminator in order to create generative samples of a data set. Generative adversarial networks are used frequently with image, video, and text data, and they generate artistic renderings of images and videos, which often have the flavor of an AI that is "dreaming." These methods can be used for image-to-image translation as well. The variational autoencoder is discussed in detail in Section 4.10.4 of Chapter 4. Generative adversarial networks are discussed in Section 10.4 of Chapter 10.

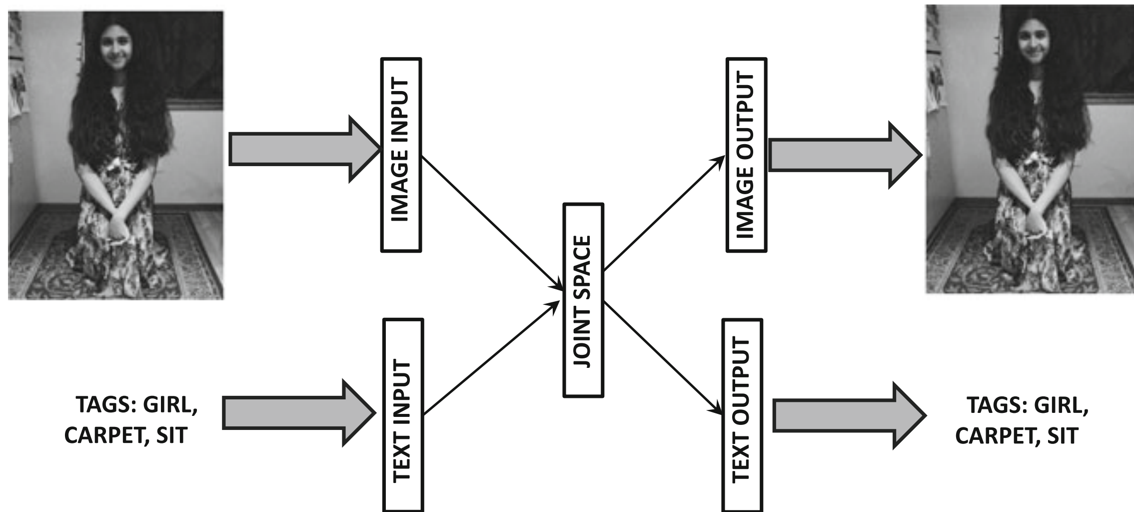


Figure 2.12: Multimodal embedding with autoencoders

One can use an autoencoder for embedding multimodal data in a joint latent space. Multimodal data is essentially data in which the input features are heterogeneous. For example, an image with descriptive tags can be considered multimodal data. Multimodal data pose challenges to mining applications because different features require different types of processing and treatment. By embedding the heterogeneous attributes in a unified space, one is removing this source of difficulty in the mining process. An autoencoder can be used to embed the heterogeneous data into a joint space. An example of such a setting is shown in Figure 2.12. This figure shows an autoencoder with only a single layer, although one might have multiple layers in general [357, 468]. Such joint spaces can be very useful in a variety of applications.

Finally, autoencoders are used to improve the learning process in neural networks. A specific example is that of *pretraining* in which an autoencoder is used to initialize the weights of a neural network. The basic idea is that learning the manifold structure of a data set is also useful for supervised learning applications like classification. This is because the features that define the manifold of a data set are often likely to be more informative in terms of their relationships to different classes. Pretraining methods are discussed in Section 4.7 of Chapter 4.

2.5.7 Recommender Systems: Row Index to Row Value Prediction

One of the most interesting applications of matrix factorization is the design of neural architectures for recommender systems. Consider an $n \times d$ ratings matrix D with n users and d items. The (i, j)th entry of the matrix is the rating of user i for item j. However, most entries in the matrix are not specified, which creates difficulties in using a traditional autoencoder architecture. This is because traditional autoencoders are designed for fully specified matrices, in which a single *row* of the matrix is input at one time. On the other hand, recommender systems are inherently suited to *elementwise* learning, in which a very

small subset of ratings from a row may be available. As a practical matter, one might consider the input to a recommender system as a set of triplets of the following form:

$$\langle \text{RowId} \rangle, \langle \text{ ColumnId } \rangle, \langle \text{ Rating } \rangle$$

As in traditional forms of matrix factorization, the ratings matrix D is given by UV^T. However, the difference is that one must learn U and V using triplet-centric input because

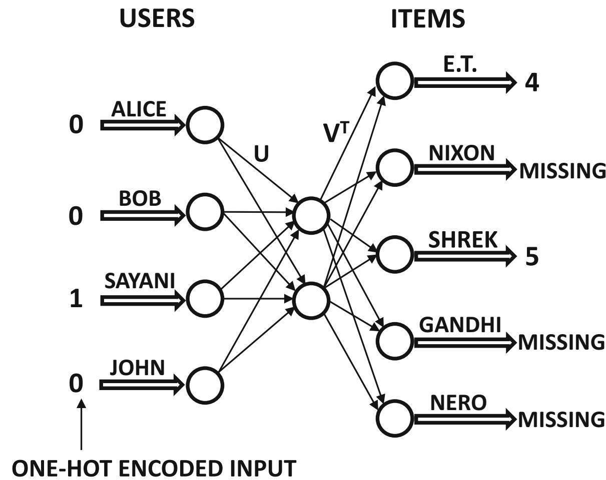


Figure 2.13: Row-index-to-value encoder for matrix factorization with missing values.

all entries of D are not observed. Therefore, a natural approach is to create an architecture in which the inputs are not affected by the missing entries and can be uniquely specified. The input layer contains n input units, which is the same as the number of rows (users). However, the input is a one-hot encoded index of the row identifier. Therefore, only one entry of the input takes on the value of 1, with the remaining entries taking on values of 0. The hidden layer contains k units, where k is the rank of the factorization. Finally, the output layer contains d units, where d is the number of columns (items). The output is a vector containing the d ratings (even though only a small subset of them are observed). The goal is to train the neural network with an incomplete data matrix D so that the network outputs all the ratings corresponding to a one-hot encoded row index after it is input. The approach is to be able to reconstruct the data by learning the ratings associated with each row index.

Consider a setting in which the $n \times k$ input-to-hidden matrix is U, and the $k \times d$ hidden-to-output matrix is V^T. The entries of the matrix U are denoted by u_{iq}, and those of the matrix V are denoted by v_{jq}. Assume that all activation functions are linear. Furthermore, let the one-hot encoded input (row) vector for the rth user be $\overline{e}_r$. This row vector contains n dimensions in which only the rth value is 1, and the remaining values are zeros. The loss function is the sum of the squares of the errors in the output layer. However, because of the missing entries, not all output nodes have an observed output value, and the updates are performed only with respect to entries that are known. The overall architecture of this neural network is illustrated in Figure 2.13. For any particular row-wise input we are really training on a neural network that is a subset of this base network, depending on which entries are specified. However, it is possible to give *predictions* for all outputs in the network (even though a loss function cannot be computed for missing entries). Since a neural network with linear activations performs matrix multiplications, it is easy to see that the vector of d outputs for the rth user is given by $\overline{e}_r UV^T$. In essence, pre-multiplication with $\overline{e}_r$ pulls out the rth row in the matrix UV^T. These values appear at the output layer and represent the

item-wise ratings predictions for the rth user. Therefore, all feature values are reconstructed in one shot.

How is training performed? The main attraction of this architecture is that one can perform the training either in row-wise fashion or in element-wise fashion. When performing the training in row-wise fashion, the one-hot encoded index for that row is input, and all *specified* entries of that row are used to compute the loss. The backpropagation algorithm is

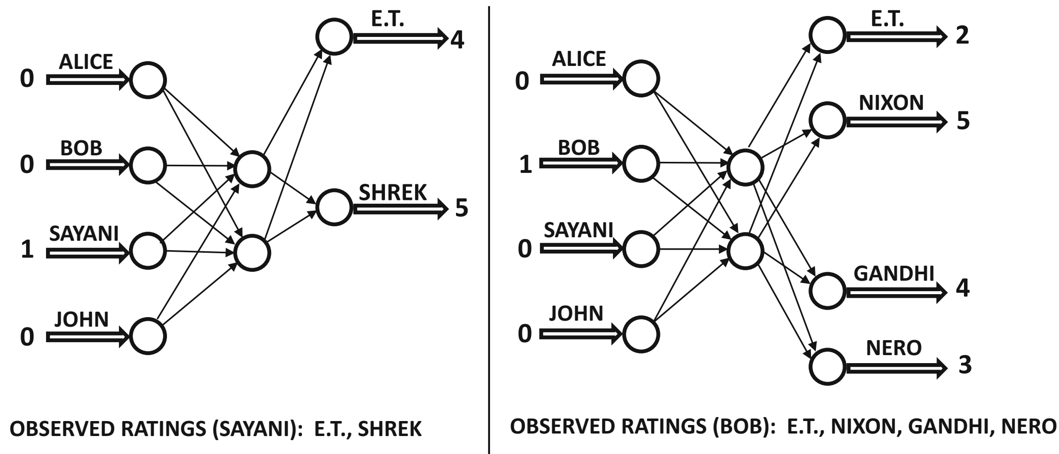


Figure 2.14: Dropping output nodes based on missing values. Output nodes are missing only at training time. At prediction time, all output nodes are materialized. One can achieve similar results with an RBM architecture as well (cf. Figure 6.5 of Chapter 6).

done only starting at output nodes where the values are specified. From a theoretical point of view, each row is being trained on a slightly different neural network with a subset of the base output nodes (depending on which entries are observed), although the weights for the different neural networks are shared. This situation is shown in Figure 2.14, where the neural networks for the movie ratings of two different users, Bob and Sayani, are shown. For example, Bob is missing a rating for *Shrek*, as a result of which the corresponding output node is missing. However, since both users have specified a rating for *E.T.*, the k-dimensional hidden factors for this movie in matrix V will be updated during backpropagation when either Bob or Sayani is processed. This ability to train using only a subset of the output nodes is sometimes used as an efficiency optimization to reduce training time even in cases where all outputs are specified. Such situations occur often in binary recommendation data sets (referred to as *implicit feedback data sets*), where the vast majority of outputs are zeros. In such cases, only a subset of zeros is sampled for training in matrix factorization methods [4]. This technique is referred to as *negative sampling*. A specific example is that of neural models for natural language processing like *word2vec*.

It is also possible to perform the training in element-wise fashion, where a single triplet is input. In such a case, the loss is computed only with respect to a single column index specified in the triplet. Consider the case where the row index is i, and the column index is j. In this specific case, and the single error computed at the output layer is $y - \hat{y} = e_{ij}$. the backpropagation algorithm essentially updates the weights on all the k paths from node j in the output layer to the node i in the input layer. These k paths pass through the k nodes in the hidden layer. It is easy to show that the update along the qth such path is as follows:

$$u_{iq} \Leftarrow u_{iq}(1 - \alpha\lambda) + \alpha e_{ij} v_{jq}$$
$$v_{jq} \Leftarrow v_{jq}(1 - \alpha\lambda) + \alpha e_{ij} u_{iq}$$

Here, α is the step-size, and λ is the regularization parameter. *These updates are identical to those used in stochastic gradient descent for matrix factorization in recommender systems.* However, an important advantage of the use of the neural architecture (over traditional matrix factorization) is that we can vary on it in so many different ways in order to enforce different properties. For example, for matrices with binary data, we can use a logistic layer in the output. This will result in *logistic matrix factorization*. We can incorporate multiple hidden layers to create more powerful models. For matrices with categorical entries (and count-centric weights attached to entries), one can use a softmax layer at the very end. This will result in *multinomial matrix factorization*. To date, we are not aware of a formal description of multinomial matrix factorization in traditional machine learning; yet, it is a simple modification of the neural architecture (implicitly) used by recommender systems. In general, it is often easy to stumble upon sophisticated models when working with neural architectures because of their modular structure. One does not need to relate the neural architecture to a conventional machine learning model, as long as empirical results establish its robustness. For example, two variations of the (highly successful) skip-gram model of *word2vec* [325, 327] correspond to logistic and multinomial matrix factorizations of word-context matrices; yet, this fact does not seem to be pointed[6] out by either by the original authors of *word2vec* [325, 327] or the broader community. In conventional machine learning, models like logistic matrix factorization are considered relatively esoteric techniques that have only recently been proposed [224]; yet, these sophisticated models represent relatively simple neural architectures. In general, the neural network abstraction brings practitioners (without too much mathematical training) much closer to sophisticated methods in machine learning, while being shielded from the details of optimization with the backpropagation framework.

2.5.8 Discussion

The main goal of this section was to show the benefits of the modular nature of neural networks in unsupervised learning. In our particular example, we started with a simple simulation of SVD, and then showed how minor changes to the neural architecture can achieve very different types of goals in intuitive settings. However, from an architectural point of view, the amount of effort required by the analyst to change from one architecture to the other is often a few lines of code. This is because modern softwares for building neural networks often provide templates for describing the architecture of the neural network, where each layer is specified independently. In a sense, the neural network is "built" with the well-known types of machine-learning units much like a child puts together building blocks of a toy. Backpropagation takes care of the details of optimization, while shielding the user from the complexities of the steps. Consider the significant mathematical differences between the specific details of SVD and logistic matrix factorization. Changing the output layer from linear to sigmoid (along with a change of loss function) can literally be a matter of changing a trivially small number of lines of code without affecting most of the remaining code (which usually isn't large anyway). This type of modularity is tremendously useful in application-centric settings. Autoencoders are also related to another type of unsupervised learning method, known as a Restricted Boltzmann Machines (RBM) (cf. Chapter 6). These methods can also be used for recommender systems, as discussed in Section 6.5.2 of Chapter 6.

[6]The work in [287] does point out a number of *implicit* relationships with matrix factorization, but not the more direct ones pointed out in this book. Some of these relationships are also pointed out in [6].