  - **Pooling statistic**. Use maximum statistic for pooling via the `MaxPool` layer.
  - **Pool size**. The square root of the feature map size (along the spatial axes) can be taken.
  - **Padding**. Follow the same principle as for convolution padding. Use `valid` padding in a shallow and `same` in a deep network.

- **Activation**. Use a nonlinear `relu` activation layer after a pooling layer to follow *convolution* → *pooling* → *activation* structure.
- **Dense layer**. After the stack of *convolution* → *pooling* → *activation* bundled layers, place a dense layer to downsize the feature map. Use a `Flatten()` layer before the dense layer to vectorize the multi-axes feature map. The number of units can be set equal to a number in the geometric series of 2 closest to the length of the flattened features divided by 4.
- **Output layer**. The output layer is dense. The number of units is equal to the number of labels in the input. For example, in a binary classifier, the number of units is equal to 1. The other configurations of the output layer follow the same rules-of-thumb as in § 4.10.

6.17 Exercises

1. **Long-term dependencies**. Recurrent neural networks, e.g., LSTMs are meant to learn long-term dependencies. However, it is found that in some high-dimensional multivariate time series problems LSTMs perform poorly if the time-steps (lookback) is increased. A convolutional neural network, on the other hand, works better even with long lookbacks.
    (a) Refer to Jozefowicz, Zaremba, and Sutskever 2015 and explain the reason behind LSTM and other RNNs limitation in learning long-term dependencies.
    (b) Explain why convolutional networks still work well with long time-steps?
    (c) Train and evaluate the baseline convolutional network in § 6.9.3 with a `TIMESTEPS` equal to `120` and `240`, and report the increase in the model accuracy, parameters, and runtime. Interpret your findings.
    (d) Plotting the feature maps and filters provide some interpretation of the model. Refer to Appendix I to plot them and report your interpretations.
2. **`Conv1D`, `Conv2D`, and `Conv3D`**. § 6.8.2 explained the different scenarios in which `Conv1D`, `Conv2D`, or `Conv3D` could be used. Next, § 6.10.1 explained that they are top-down interchangeable. Also, as shown in § 6.10.2, the interchangeability provides more modeling choices when a higher level `Conv‘x’D` is used.
    (a) Refer to § 6.10.1 and construct a convolutional network using `Conv3D` layer that is equivalent to the baseline model in § 6.9.3.
    (b) Construct another convolutional network using `Conv3D` layer equivalent to the neighborhood model in § 6.10.2.
    (c) Explain how a `Conv3D` layer can replace a `Conv2D` layer in a convolutional network for images?
3. **1×1 Convolution**. A 1×1 convolutional layer summarizes the information across the channels.

   (a) Explain how is this conceptually similar to and different from pooling?
   (b) Refer to Appendix H to construct a network with 1×1 convolutional layers. Present the improvement in model performance.
   (c) Plot the feature maps outputted by the 1×1 convolutional layer (refer Appendix I) and present your interpretations.

4. **Summary Statistics.** § 6.11 expounded the theory of minimal sufficient and complete statistics which led to a maximum likelihood estimator (MLE) shown as the best pooling statistic in § 6.13. Additionally, ancillary statistics were put forward as complementary pooling statistics to enhance a network's performance.
   (a) Prove that the maximum statistic $\max_i X_i$ is a complete minimal sufficient statistic for a uniform distribution. Also, show that it is the MLE for uniform as mentioned in § 6.13.1.
   (b) Derive the MLEs for normal, gamma, and Weibull distributions given in § 6.13.2-6.13.4.

5. **Pooling statistics**. It is mentioned in § 6.12 the feature map distribution distortion is caused by nonlinear activations. A possibility of using pooling statistics other than average and maximum arises by addressing the distortion.
   (a) Train the baseline model with max- and average pooling. Then swap activation and pooling layers, and make the convolutional layer activation as `linear` as described in § 6.12.2 in the same baseline network. Train with max- and average-pooling again. Compare the results for both pooling before and after the swap.
   (b) Construct a network by referring to Appendix J with a maximum and range ($\max - \min$) pooling statistics in parallel. The range statistic is an *ancillary* statistic that complements the maximum statistic with regards to the information drawn from a feature map. Present and discuss the results.

(c) Explain why swapping the activation and pooling layers make use of ancillary statistics such as range and standard deviation possible?

(d) (Optional) Construct a network with average and standard deviation pooling statistics in parallel. They are the MLEs of a normal distribution. If feature maps are normally distributed, they pool the most relevant information. Present and discuss the results.

(e) (Optional) Construct a network with Weibull distribution MLE as the pooling statistic. Train the network with the shape parameter k in $\{0.1, 1, 10, 100\}$. Present the results and explain the effect of k.

Chapter 7

Autoencoders

7.1 Background

An **autoencoder** is a reconstruction model. It attempts to reconstruct its inputs from itself as depicted below,

$$\boldsymbol{x} \rightarrow \underbrace{f(\boldsymbol{x}) \rightarrow \boldsymbol{z}}_{encoding} \rightarrow \underbrace{g(\boldsymbol{z}) \rightarrow \hat{\boldsymbol{x}}}_{decoding}. \tag{7.1}$$

Clearly, an autoencoder is made of two modules: an encoder and a decoder. As their names indicate, an encoder f encodes input $\boldsymbol{x}$ into encodings $\boldsymbol{z} = f(\boldsymbol{x})$, and a decoder g decodes $\boldsymbol{z}$ back to a closest reconstruction $\hat{\boldsymbol{x}}$.

Training a model to predict (construct) $\hat{\boldsymbol{x}}$ from $\boldsymbol{x}$ sometimes appear trivial. However, an autoencoder does not necessarily strive for a perfect reconstruction. Instead, the goal could be dimension reduction, denoising, learning useful features for classification, pre-training another deep network, or something else.

Autoencoders, therefore, fall in the category of **unsupervised** and **semi-supervised** learning.

They were conceptualized in the late 80's. These early works were inspired by principal component analysis. PCA, which was invented more than a century ago (Pearson 1901), has remained a popular feature representation technique in machine learning for dimension reduction and feature representation. Autoencoders developed in Rumelhart, G. E. Hinton, and R. J. Williams 1986; Baldi and Hornik 1989 provided a neural network version of PCA.

Over the past two decades, autoencoders have come far ahead. Sparse encoding with feature space larger than the input was developed in Ranzato, Poultney, et al. 2007; Ranzato, Boureau, and Cun 2008. Furthermore, denoising autoencoders based on corrupted input learning in Vincent, Larochelle, Bengio, et al. 2008; Vincent, Larochelle, Lajoie, et al. 2010 and contractive penalty in Rifai, Vincent, et al. 2011; Rifai, Mesnil, et al. 2011 came forward.

In this chapter, some of the fundamental concepts behind this variety of autoencoders are presented. It begins with explaining an autoencoder vis-à-vis a PCA in § 7.2. A simpler explanation is provided here by drawing parallels. Thereafter, the family of autoencoders and their properties are presented in § 7.3.

The chapter then switches to using autoencoders for rare event prediction. An anomaly detection approach is used in § 7.4. Thereafter, a sparse autoencoder is constructed and a classifier is trained on the sparse encodings in § 7.5.

A sparse LSTM and convolutional autoencoders are also constructed in § 7.6 for illustration. Towards the end of the chapter, § 7.7 presents autoencoder customization in TensorFlow to incorporate sparse covariance and orthogonal weights properties. The customization example here is intended to help researchers implement their ideas. Lastly, the chapter concludes with a few rules-of-thumb for autoencoders.

7.2 Architectural Similarity between PCA and Autoencoder

For simplicity, a linear single layer autoencoder is compared with principal component analysis (PCA).

There are multiple algorithms for PCA modeling. One of them is estimation by minimizing the reconstruction error (see § 12.1.2 in Bishop

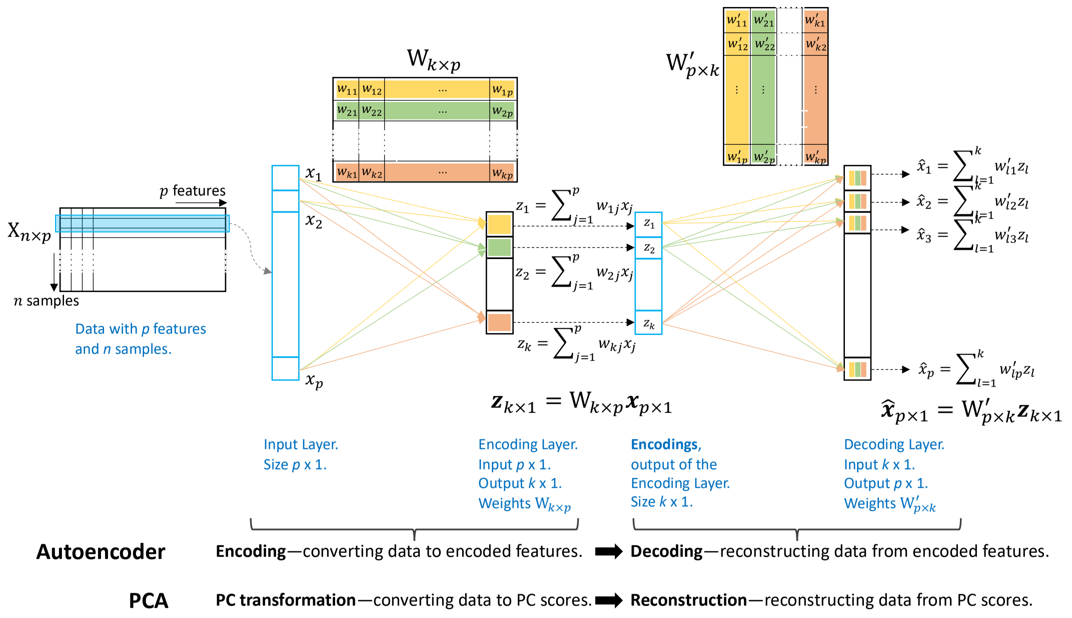

$X_{n\times p}$
p features
n samples
Data with p features and n samples.
$W_{k\times p}$
$W'_{p\times k}$
x_1
x_2
x_p
$z_1 = \sum_{j=1}^{p} w_{1j}x_j$
$z_2 = \sum_{j=1}^{p} w_{2j}x_j$
$z_k = \sum_{j=1}^{p} w_{kj}x_j$
$\hat{x}_1 = \sum_{l=1}^{k} w'_{l1}z_l$
$\hat{x}_2 = \sum_{l=1}^{k} w'_{l2}z_l$
$\hat{x}_3 = \sum_{l=1}^{k} w'_{l3}z_l$
$\hat{x}_p = \sum_{l=1}^{k} w'_{lp}z_l$
$\mathbf{z}_{k\times 1} = W_{k\times p}\boldsymbol{x}_{p\times 1}$
$\hat{\boldsymbol{x}}_{p\times 1} = W'_{p\times k}\mathbf{z}_{k\times 1}$
Input Layer. Size p x 1.
Encoding Layer. Input p x 1. Output k x 1. Weights $W_{k\times p}$
Encodings, output of the Encoding Layer. Size k x 1.
Decoding Layer. Input k x 1. Output p x 1. Weights $W'_{p\times k}$
Autoencoder
Encoding—converting data to encoded features.
Decoding—reconstructing data from encoded features.
PCA
PC transformation—converting data to PC scores.
Reconstruction—reconstructing data from PC scores.

Figure 7.1. *Illustration of PCA and Autoencoder relationship.*

2006). Following this algorithm gives a clearer understanding of the similarities between a PCA and an autoencoder.

Figure 7.1 visualizes a single-layer linear autoencoder. As shown at the bottom of the figure, the encoding process is similar to the principal component transformation.

PC transformation is projecting the original data on the principal components to yield reduced-dimension features called principal scores. This is similar to the *encoding* process in an autoencoder.

Conversely, an autoencoder's *decoding* process is similar to reconstructing the data from the Principal Scores.

In the following, their relationship is further elaborated by showcasing the key autoencoder components and their counterpart in PCA.

7.2.1 Encoding—Projection to Lower Dimension

PCA is a covariance decomposition expressed as $\Sigma = W\Lambda W^T$, where Σ is the covariance matrix of X, W is the eigenvectors matrix, and Λ is an eigenvalues diagonal matrix.

The W matrix is an orthogonal basis, also known as the principal components. The transformation of the original data X into principal features is,

$$Z = XW \tag{7.2}$$

where W is a $p \times k$ matrix with eigenvectors in each column. The eigenvectors are the top k principal components. The k is chosen as less than the original dimension p of X, i.e., $k < p$. This dimension reduction can also be seen as an **encoding**.

A similar process occurs in the encoding stage of an autoencoder. Look closely at the encoder section in Figure 7.1.

The encoding (dense) layer has k nodes. A colored cell in the layer is a computing node with p weights. The weights can also be denoted as a $p \times k$ matrix W. The weights on each node are the equivalent of an eigenvector in PCA.

The encoding output is,

$$Z = g(XW) \tag{7.3}$$

where g is an activation function. If the activation is linear, the encoding in autoencoder becomes equivalent to the principal scores in PCA (Equation 7.2).

7.2.2 Decoding—Reconstruction to Original Dimension

Reconstructing the data is also called **decoding**. The original data can be reconstructed (estimated) from the principal scores as,

$$\hat{X} = ZW^T. \tag{7.4}$$

Similarly, the decoder in an autoencoder reconstructs the data. As shown in Figure 7.1, the data is reconstructed as,

$$\hat{X} = ZW' \tag{7.5}$$

where W' is a $k \times p$ weight matrix. Unlike in encoding, the activation in decoding depends on the range of the input. For example, if the input $\boldsymbol{x} \in \mathbb{R}$ then a linear activation but if $\boldsymbol{x} \in \mathbb{R}^+$ then a ReLU-like activation. Since the inputs are usually scaled to belong in $\mathbb{R}$, it is safe to assume that the decoding has linear activation.

Note that the W in Equation 7.4 for PCA is the same as the encoding weights in Equation 7.2. But a different weight W' in Equation 7.5 is for a decoder. This is because the encoding and decoding weights are not necessarily the same in autoencoders.

But they can be *tied* to become the same. This unison is shown in Figure 7.1 using colors in the W matrices. When the encoder and decoder weights are tied, we have

$$W' = W^T \tag{7.6}$$

which means the rows of the encoder weights become equal to the columns of decoder weights.

If the encoder and decoder weights are ***tied*** *and the encoder has* ***linear*** *activation, the autoencoder becomes equivalent to a PCA model. Conversely, a nonlinearly activated encoder is a nonlinear extension of PCA.*

7.3 Autoencoder Family

There are several types of autoencoders. Table 7.1 summarizes the properties of the most common autoencoders. The rest of this section briefly describes them along with their applications.

7.3.1 Undercomplete

An undercomplete autoencoder has a smaller encoding dimension than the input. A simple example of such an autoencoder is,

$$X_{\cdot\times p} \rightarrow \underbrace{f(X_{\cdot\times p}W^{(e)}_{p\times k})}_{encoder} \rightarrow Z_{\cdot\times k} \rightarrow \underbrace{g(Z_{\cdot\times k}W^{(d)}_{k\times p})}_{decoder} \rightarrow \hat{X}_{\cdot\times p}. \tag{7.7}$$

Here the input X and the encoding Z are p- and k-dimensional, respectively, and $k < p$.

In learning a smaller representation, an undercomplete autoencoder gathers the most salient features of the data. The learning process is simply minimizing a loss function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}W^{(e)})W^{(d)})) \tag{7.8}$$

where L is a loss function, for example, mean squared error, that penalizes dissimilarity between $\boldsymbol{x}$ and $\hat{\boldsymbol{x}} = g(f(\boldsymbol{x}W^{(e)})W^{(d)})$.

Undercomplete autoencoders are more common. Perhaps because it has roots in PCA. A linearly activated (single or multilayer) undercomplete autoencoder reduces to

Table 7.1. Types of autoencoders and their properties.

	Undercomplete autoencoder	Regularized overcomplete	Sparse autoencoder	Denoising autoencoder	Contractive autoencoder	Well-posed autoencoder
(Near) Orthogonal weights, $\lambda\|\|W^TW - I\|\|_F^2$						X
Unitnorm weights, $\|\|W\|\|_2 = 1$	X	X				X
Sparse encoding covariance, $\lambda\|\|\Omega_z(1 - I)\|\|_F^2$						X
Sparse encoding, $\lambda\|\|Z\|\|_1$		X	X			
Small derivative, $\lambda\|\|\frac{\partial z}{\partial \boldsymbol{x}}\|\|_F^2$		X			X	
Small encoding, $k < p$	X					X
Corrupted input, $\boldsymbol{x} \leftarrow \boldsymbol{x} + \epsilon$				X		

$$X_{\cdot \times p} \to X_{\cdot \times p} W^{(e)}_{p \times k} \to Z_{\cdot \times k} \to Z_{\cdot \times k} W^{(d)}_{k \times p} \to \hat{X}_{\cdot \times p},$$

which is the same as PCA if $W^{(e)}$ has eigenvectors.

These roots bring the dimension reduction ability in undercomplete autoencoder. Their encodings are, therefore, used in data compression and data transformation. For example, classifiers on high-dimensional data are sometimes trained on its encodings (a data transformation).

7.3.2 Overcomplete

While undercomplete autoencoder does dimension reduction and learns the most important features, it sometimes fails to learn the true underlying process. It fails when the encoder and decoder are given too much capacity, i.e., a lot to learn in a small dimension when k is small.

An overcomplete autoencoder overcomes the issue by allowing the encoding dimension to be equal to or larger than the input. This eases the over capacity issue.

A large encoding may appear counter-intuitive to the common understanding of autoencoders. It does not offer dimension reduction! On the contrary, we have more. Even worse, an overcomplete autoencoder can easily learn to become an identity transform, i.e., a trivial model with $W^{(e)} = I_{p \times p}$.

To avoid triviality, an overcomplete autoencoder is regularized. A nonlinear and regularized overcomplete autoencoder can effectively learn the underlying data distribution due to its larger capacity.

A simple overcomplete autoencoder is similar to an undercomplete autoencoder in Equation 7.7 except $k \geq p$. The difference is in the loss function. Overcomplete autoencoders **necessarily** have regularization included in them. For example,

$$L(\boldsymbol{x}, \hat{\boldsymbol{x}}) + \lambda_1 ||\boldsymbol{z}||_1 + \lambda_2 \left|\left| \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \right|\right|_F^2 \tag{7.9}$$

where $L(\boldsymbol{x}, \hat{\boldsymbol{x}})$ is typically a mean squared error to keep the model output $\hat{\boldsymbol{x}}$ close to the input $\boldsymbol{x}$.

The second term $\lambda_1||\boldsymbol{z}||_1$ encourages sparse encodings $\boldsymbol{z}$. This is discussed more in § 7.3.5 on **sparse autoencoders**. Besides, it is argued in Goodfellow, Bengio, and Courville 2016 that the L_1-norm[1] sparsity penalty is not a "regularization" term. Instead, it is a way to approximately train a generative model.

The third term $\left|\left|\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right|\right|_F^2$ makes the derivative of the encodings with respect to the input smaller. This forces the autoencoder to learn a function that is less sensitive to minor changes in $\boldsymbol{x}$. An autoencoder with derivative regularization is called a **contractive autoencoder** discussed in § 7.3.4.

With these regularization properties taken from other autoencoders, an overcomplete autoencoder learns useful abstraction of the data distribution even if its capacity is large enough to learn a trivial identity function.

Besides, remember that Equation 7.9 is only an example of overcomplete model loss function. An overcomplete autoencoder can work with any one of the two or other types of regularization as well.

7.3.3 Denoising Autoencoder (DAE)

Observing noisy data is common in several problems. One application of autoencoders is to denoise the data. A specially designed *denoising* autoencoder serves this purpose.

A simple denoising autoencoder appears as

$$\boldsymbol{x} + \boldsymbol{\epsilon} \rightarrow f(\boldsymbol{x} + \boldsymbol{\epsilon}) \rightarrow \boldsymbol{z} \rightarrow g(\boldsymbol{z}) \rightarrow \hat{\boldsymbol{x}}, \qquad (7.10)$$

where $\boldsymbol{\epsilon}$ is a noise added to the input.

As shown in Equation 7.10, a denoising autoencoder deliberately adds noise denoted with ϵ in the input. The model is then trained to reconstruct the original $\boldsymbol{x}$ from its noisy version $\boldsymbol{x} + \boldsymbol{\epsilon}$. The loss function

[1]An L_1-norm is $||\boldsymbol{z}||_1 = \sum_i |z|_i$.

is, therefore, defined as

$$L(\boldsymbol{x}, g(f(\boldsymbol{x} + \boldsymbol{\epsilon}))). \tag{7.11}$$

In minimizing the loss, a denoising autoencoder learns to skim denoised data from noise. It is shown in Bengio et al. 2013; Alain and Bengio 2014 the model implicitly learns the structure of the data distribution $p_{data}(\boldsymbol{x})$.

Besides, a denoising autoencoder can be constructed as an overcomplete high-capacity autoencoder. It is because its loss function becomes

$$\mathbb{E}(||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2) + \sigma^2 \mathbb{E}\left[\left|\left|\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right|\right|_F^2\right] + o(\sigma^2) \tag{7.12}$$

if we have mean squared error criterion and the noise is additive, $\boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{\epsilon}$, and gaussian, $\epsilon \sim N(0, \sigma)$ (refer to Alain and Bengio 2014).

7.3.4 Contractive Autoencoder (CAE)

A contractive autoencoder is trained to make encodings robust to noisy perturbations in the input. To encourage robustness, the *sensitivity* of $\boldsymbol{z}$ to any perturbations in the input is penalized during model training.

How to measure the sensitivity? The derivative of encodings with respect to the input makes an appropriate measure.

If the input and encoding are $\boldsymbol{x} \in \mathbb{R}^p$ and $\boldsymbol{z} \in \mathbb{R}^k$, respectively, then the derivative $\dfrac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$ is a $k \times p$ matrix. The matrix of derivatives is called a Jacobian denoted by J where

$$J_{ij} = \frac{\partial z_i}{\partial x_j}, \; i = 1, \ldots, k, \; j = 1, \ldots, p. \tag{7.13}$$

The overall sensitivity of a model encodings is estimated by the Frobenius-norm of J, which in simple terms is the sum of squares of every derivative J_{ij}. The norm is expressed as $||J||_F^2 = \sum_{ij} \left(\dfrac{\partial z_i}{\partial x_j}\right)^2$.

Consequently, the CAE loss function with the penalty on derivatives is

$$L(\boldsymbol{x}, \hat{\boldsymbol{x}}) + \lambda \left|\left| \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \right|\right|_F^2. \tag{7.14}$$

The objective of a contractive autoencoder is similar to a denoising autoencoder: make the encodings robust to noise. Alain and Bengio 2014 theoretically show that they are equivalent if denoising autoencoder is trained by adding small Gaussian noise to the input.

However, they achieve robustness differently. Contractive autoencoder forces the encodings $\boldsymbol{z}$ to resist perturbations in the input. On the other hand, a denoising autoencoder forces this on the inferences $\hat{\boldsymbol{x}}$.

In doing so, a contractive autoencoder learns **better** encodings to use in other tasks such as classification. Rifai, Vincent, et al. 2011 achieved state-of-the-art classification errors on a range of datasets using contractive autoencoders learned features to train MLPs.

"The best classification accuracy usually results from applying the contractive penalty (for robustness to input perturbations) to the encodings $\boldsymbol{z}$ rather than to the inferences $\hat{\boldsymbol{x}}$,"

–Goodfellow, Bengio, and Courville 2016.

Rifai, Vincent, et al. 2011 found that the Jacobian norm penalty in contractive autoencoder loss (Equation 7.14) carve encodings that correspond to a lower-dimensional non-linear manifold. Simply put, a manifold is a house of data. The objective of autoencoders is to find the shape of the smallest such house. In doing so, its encodings learn the essential characteristics of the data.

In simple terms, a manifold is a house of data. An autoencoder finds the shape of the smallest such house.

The encodings capture the local directions of the variations along the manifold while being more invariant to a majority of directions orthogonal to the manifold.

Think of this as the manifold house has data on its walls. Variations along the walls should be captured. But variations perpendicular to a wall (and on the outside) is noise. The model should, therefore, be invariant to them.

Rifai, Mesnil, et al. 2011 went further to add a penalty on the second derivative of encodings $\frac{\partial^2 \boldsymbol{z}}{\partial \boldsymbol{x}^2}$ called the Hessian to develop a higher-order contractive autoencoder. While the first derivative Jacobian penalty ensures robustness to small noise, the Hessian penalty extends this robustness to larger noise.

From a manifold perspective, the Hessian penalty penalizes curvature. This means the manifold house will have smoother walls. Consequently, the noisier data falls out of it easily.

With its benefits, a practical issue with the contractive autoencoders is that although the Jacobian or Hessian is cheap to compute in single hidden layer autoencoders, it becomes expensive in deeper autoencoders. Rifai, Vincent, et al. 2011 addressed the issue by separately training a series of single-layer autoencoders stacked to make a deep autoencoder. This strategy makes each layer locally contractive and, consequently, the deep autoencoder contractive.

7.3.5 Sparse Autoencoder

A sparse autoencoder imposes sparsity on the encodings $\boldsymbol{z}$. The sparse encodings are typically drawn to perform another task, such as classification.

For instance, suppose we have labeled data $(\boldsymbol{x}, y)$ from a high-dimensional process. Learning a classifier on high-dimensional data is challenging. However, a sparse autoencoder can extract the relevant information along with reducing the classification features dimension. An example of such a model is

$$\begin{tikzcd}
\boldsymbol{x} \arrow[r] & f(\boldsymbol{x}) \arrow[r] & \boldsymbol{z} \arrow[r] \arrow[d] & g(\boldsymbol{z}) \arrow[r] & \hat{\boldsymbol{x}} \\
 & & {} \arrow[r] & h(\boldsymbol{z}) \arrow[r] & \hat{y}
\end{tikzcd}$$ (7.15)

where $\hat{\boldsymbol{x}}$ is the reconstructed $\boldsymbol{x}$ and $\hat{y}$ is the predicted label from the encodings $\boldsymbol{z}$, and f, g, h, denote encoder, decoder, and classifier, respectively.

Its loss function is expressed as

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \lambda||\boldsymbol{z}||_1 \tag{7.16}$$

where the penalty is an L_1-norm $\lambda||\boldsymbol{z}||_1 = \lambda \sum_i |z_i|$.

The encoding size can be smaller, equal, or larger than the input. It can vary based on the problem. Regardless, the sparsity penalty in a sparse autoencoder extracts essential statistical features in the input data.

This property is guaranteed because the sparsity penalty results in the autoencoder approximate a generative model. It is straightforward to show that the sparsity penalty is arrived at by framing a sparse autoencoder as a generative model that has latent variables (refer to § 14.2.1 in Goodfellow, Bengio, and Courville 2016). The latent variables here are the encodings $\boldsymbol{z}$.

Sparsity penalty in an autoencoder is not a "regularization" term. It is a way of approximating a generative model.

Approximating a generative model with an autoencoder ensures the encodings are useful, i.e., they describe the latent variables that explain the data.

Besides, the loss function in Equation 7.16 is if the autoencoder is trained independent of the classification task. The loss function can be modified to include classification. For example, the classifier and autoencoder in Equation 7.15 can be trained simultaneously by including the classification error in the loss as

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + L'(y, h(\boldsymbol{z})) + \lambda||\boldsymbol{z}||_1 \tag{7.17}$$

where $L'(y, h(\boldsymbol{z}))$ penalizes differences between the actual and predicted labels such as cross-entropy.

The former approach has two-stage learning. The encodings are learned in the first stage by minimizing Equation 7.16 independent of the classification task. The encodings act as latent features to train a classifier in the second stage. This approach is tractable and, therefore, easier to train. However, the encodings can sometimes be ineffective in the classifier.

The simultaneous learning approach by minimizing loss in Equation 7.17 ensures the autoencoder learns the latent features such that they are capable of classifying. However, it lacks tractability.

🔔 *Sparse autoencoders learn relevant information in a reduced dimension useful in other tasks such as classification.*

7.4 Anomaly Detection with Autoencoders

Anomaly detection is unarguably one of the best approaches in rare event detection problems. Especially, if the event is so rare that there are insufficient samples to train a classifier. Fortunately, an (extremely) rare event often appears as an anomaly in a process. They can, therefore, be detected due to their abnormality.

Petsche et al. 1996 have one of the early works in deep learning which developed anomaly detectors for rare event detection. They developed an "autoassociator" to detect an imminent motor failure.

The "autoassociator" was essentially a reconstructor. Petsche et al.

1996 showed that the autoassociator has a small reconstruction error on measurements recorded from healthy motors but a larger error on those recorded from faulty motors.

This difference in the reconstruction errors: small for normal conditions and large for a rare anomaly forms the basis of anomaly detection models for rare events.

This section presents the anomaly detection approach for rare event detection using an autoencoder. First, the approach is explained in § 7.4.1. Then an autoencoder construction and application for rare event prediction is given in § 7.4.2-7.4.5.

7.4.1 Anomaly Detection Approach

If an event is extremely rare, we can use an **anomaly detection approach** to predict the rare event. In this approach, the rare event is treated as anomalies. The model is trained to detect the anomaly.

Given a data set $(\boldsymbol{x}_t, y_t), t = 1, \ldots, T$, where y_t corresponds to the event and $\boldsymbol{x}_t \in \mathbb{R}^p$ are the process variables at time t with p dimensions. The problem is to detect a rare event, if one occurred, at a time t' when $t' > T$.

At time $t' > T$, the event $y_{t'}$ is unknown. For example, if a bank fraud happened at t' it may remain undetected until reported by the account holder. By that time, a loss is likely to be already incurred. Instead, if the fraud was detected at t', the fraudulent transaction could be withheld.

The anomaly detection works using a *reconstruction model* approach for detection. The model learns to reconstruct the process variables $\boldsymbol{x}_t$ given itself, i.e., the model is $\hat{\boldsymbol{x}}_t | \boldsymbol{x}_t$.

The approach can be broken down as follows.

1. **Training**
   - (a) Divide the process data into two parts: majority class (negatively labeled), $\{\boldsymbol{x}_t, \forall t | y_t = 0\}$, and minority class (positively labeled), $\{\boldsymbol{x}_t, \forall t | y_t = 1\}$.
   - (b) The majority class is treated as a normal state of a process.

The normal state is when the process is event-less.

(c) Train a reconstruction model on the normal state samples $\{\boldsymbol{x}_t, \forall t | y_t = 0\}$, i.e., ignore the positively labeled minority data.

2. **Inferencing**

   (a) A well-trained reconstruction model will be able to accurately reconstruct a new sample $\boldsymbol{x}_{t'}$ if it belongs to the normal state. It will, therefore, have a small reconstruction error $||\boldsymbol{x}_{t'} - \hat{\boldsymbol{x}}_{t'}||_2^2$.

   (b) However, a sample during a rare-event would be abnormal for the model. The model will struggle to reconstruct it. Therefore, the reconstruction error will be large.

   (c) Such an instance of high reconstruction error is called out as a rare event occurrence.

7.4.2 Data Preparation

As usual, we start model construction by loading the libraries in Listing 7.1.

Listing 7.1. Load libraries for dense reconstruction model

```python
import numpy as np
import pandas as pd

%tensorflow_version 2.x
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Layer
from tensorflow.keras.layers import InputSpec
from tensorflow.keras.callbacks import
    ModelCheckpoint
from tensorflow.keras.callbacks import TensorBoard
from tensorflow.keras import regularizers
from tensorflow.keras import activations
from tensorflow.keras import initializers
from tensorflow.keras import constraints
from tensorflow.keras import Sequential
from tensorflow.keras import backend as K
from tensorflow.keras.constraints import UnitNorm
from tensorflow.keras.constraints import Constraint
from tensorflow.python.framework import tensor_shape

from numpy.random import seed
seed(123)

from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import scipy

# User-defined library
import utilities.datapreprocessing as dp
import utilities.reconstructionperformance as rp
import utilities.simpleplots as sp
```

In the anomaly detection approach, an autoencoder is trained on the "normal" state of a process which makes the majority class. In the dataset, those are the `0` labeled samples. They are, therefore, separated for training and a `scaler` is fitted on it. The rest of the validation and test sets are scaled with it.

Listing 7.2. Data preparation for rare event detection

```python
# The data is taken from https://arxiv.org/abs
    /1809.10717. Please use this source for any
    citation.

df = pd.read_csv("data/processminer-sheet-break-rare
    -event-dataset.csv")
df.head(n=5)  # visualize the data.

# Hot encoding
hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
hotencoding1 = hotencoding1.add_prefix('grade_')
hotencoding2 = pd.get_dummies(df['EventPress'])
hotencoding2 = hotencoding2.add_prefix(
    'eventpress_')

df = df.drop(['Grade&Bwt', 'EventPress'],
             axis=1)

df = pd.concat([df, hotencoding1, hotencoding2],
               axis=1)

# Rename response column name for ease of
  understanding
df = df.rename(columns={'SheetBreak': 'y'})

# Sort by time.
df['DateTime'] = pd.to_datetime(df.DateTime)
df = df.sort_values(by='DateTime')

# Shift the response column y by 2 rows to do a 4-
  min ahead prediction.
df = dp.curve_shift(df, shift_by=-2)

# Drop the time column.
df = df.drop(['DateTime'], axis=1)

# Split the data and scale

DATA_SPLIT_PCT = 0.2
SEED = 123
df_train, df_test =
    train_test_split(df,
                     test_size=DATA_SPLIT_PCT,
                     random_state=SEED)
df_train, df_valid =
    train_test_split(df_train,
                     test_size=DATA_SPLIT_PCT,
                     random_state=SEED)

df_train_0 = df_train.loc[df['y'] == 0]
df_train_1 = df_train.loc[df['y'] == 1]
df_train_0_x = df_train_0.drop(['y'], axis=1)
df_train_1_x = df_train_1.drop(['y'], axis=1)

df_valid_0 = df_valid.loc[df['y'] == 0]
```

```
df_valid_1 = df_valid.loc[df['y'] == 1]
df_valid_0_x = df_valid_0.drop(['y'], axis=1)
df_valid_1_x = df_valid_1.drop(['y'], axis=1)

df_test_0 = df_test.loc[df['y'] == 0]
df_test_1 = df_test.loc[df['y'] == 1]
df_test_0_x = df_test_0.drop(['y'], axis=1)
df_test_1_x = df_test_1.drop(['y'], axis=1)

scaler = StandardScaler().fit(df_train_0_x)
df_train_0_x_rescaled =
    scaler.transform(df_train_0_x)
df_valid_0_x_rescaled =
    scaler.transform(df_valid_0_x)
df_valid_x_rescaled =
    scaler.transform(df_valid.drop(['y'],
                                   axis = 1))

df_test_0_x_rescaled =
    scaler.transform(df_test_0_x)
df_test_x_rescaled =
    scaler.transform(df_test.drop(['y'],
                                  axis = 1))
```

7.4.3 Model Fitting

An undercomplete autoencoder is constructed here. Its configurations are as follows,

- A single layer encoder and decoder.
- The encoding size is taken as approximately half of the input features. The size is chosen from the geometric series of 2 which is closest to the half.
- The encoding layer is `relu` activated.
- The encoding weights are regularized to encourage orthogonality (see § 7.7.3).
- The encodings are regularized to encourage a sparse covariance (see § 7.7.4).

- The encoding and decoding weights are constrained to have a unit norm. The encoding weights are normalized along the **rows** axis while the decoding weights on the **columns** axis.
- The decoding layer is `linear` activated. It is mandatory to have it linearly activated because it reconstructs the input that ranges in $(-\infty, \infty)$.
- The loss function is mean squared error that is appropriate because $\boldsymbol{x}$ is unbounded.
- The input `x` and output `y` in the `.fit()` function are the same in an autoencoder as the objective is to reconstruct $\boldsymbol{x}$ from itself.

Listing 7.3. Autoencoder model fitting

```
# Autoencoder for rare event detection

input_dim = df_train_0_x_rescaled.shape[1]

encoder = Dense(units=32,
                activation="relu",
                input_shape=(input_dim,),
                use_bias = True,
                kernel_regularizer=
                  OrthogonalWeights(
                    weightage=1.,
                    axis=0),
                kernel_constraint=
                  UnitNorm(axis=0),
                activity_regularizer=
                  SparseCovariance(weightage=1.),
                     name='encoder')

decoder = Dense(units=input_dim,
                activation="linear",
                use_bias = True,
                kernel_constraint=
                  UnitNorm(axis=1), name='decoder')

autoencoder = Sequential()
autoencoder.add(encoder)
autoencoder.add(decoder)

autoencoder.summary()
autoencoder.compile(metrics=['accuracy'],
                    loss='mean_squared_error',
                    optimizer='adam')

history = autoencoder.fit(x=df_train_0_x_rescaled,
                    y=df_train_0_x_rescaled,
                    batch_size=128,
                    epochs=100,
                    validation_data=
                        (df_valid_0_x_rescaled,
                         df_valid_0_x_rescaled),
                    verbose=0).history
```

7.4.4 Diagnostics

A few simple diagnostics are done on the fitted autoencoder. In Listing 7.4, it is shown that the dot product of encoding weights are nearly an identity matrix. That shows the effect of the `OrthogonalWeights()` regularizer.

The listing also shows that the weights have an exact unit-norm. A unit-norm constraint is relatively simpler to impose than orthogonality. Unit-norm does not significantly impact learning. It is simply normalizing any estimated set of weights. Unlike orthogonality, this constraint can, therefore, be a hard-constraint[2].

Listing 7.4. Near orthogonal encoding weights

```
# Near orthogonal encoding weights
w_encoder = autoencoder.get_layer('encoder').
   get_weights()[0]
print('Encoder weights dot product\n',
      np.round(np.dot(w_encoder.T, w_encoder), 2))

# Encoder weights dot product
```

[2]A hard-constraint is strictly applied while a soft-constraint is encouraged to be present in a model.

```
#   [[ 1.   0.  -0.  ...  -0.   0.  -0.]
#   [ 0.   1.   0.  ...   0.  -0.   0.]
#   [-0.   0.   1.  ...  -0.   0.   0.]
#   ...
#   [-0.   0.  -0.  ...   1.  -0.  -0.]
#   [ 0.  -0.   0.  ...  -0.   1.   0.]
#   [-0.   0.   0.  ...  -0.   0.   1.]]

# Encoding weights have unit norm
w_encoder = np.round(autoencoder.get_layer('encoder'
   ).get_weights()[0], 2).T
print('Encoder weights norm, \n',
      np.round(np.sum(w_encoder ** 2, axis = 1), 1))

# Encoder weights norm,
#   [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
#    1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
#    1. 1. 1. 1. 1. 1.]
```

After this, Listing 7.5 shows the covariance sparsity of the learned encodings. That ensures the encodings have less redundant information.

Listing 7.5. Sparse covariance of encodings

```
# Nearly-uncorrelated encoded features
encoder_model = Model(inputs=autoencoder.inputs,
                      outputs=autoencoder.get_layer(
                         'encoder').output)
encoded_features = pd.DataFrame(encoder_model.
   predict(df_train_0_x_rescaled))

print('Encoded feature correlations\n',
      np.round(encoded_features.corr(), 2))

# Encoded feature correlations
#          0      1      2   ...     29     30     31
# 0     1.00  -0.01  -0.08   ...   0.04   0.03  -0.12
# 1    -0.01   1.00   0.05   ...  -0.01   0.06  -0.02
# 2    -0.08   0.05   1.00   ...   0.11  -0.02   0.10
# 3     0.00  -0.03  -0.04   ...   0.17   0.04  -0.02
# ...
# 28  -0.02   0.09   0.15   ...  -0.10  -0.16  -0.03
# 29   0.04  -0.01   0.11   ...   1.00  -0.01  -0.08
# 30   0.03   0.06 -0.02   ...  -0.01   1.00   0.09
# 31  -0.12  -0.02   0.10   ...  -0.08   0.09   1.00
# [32 rows x 32 columns]
```

7.4.5 Inferencing

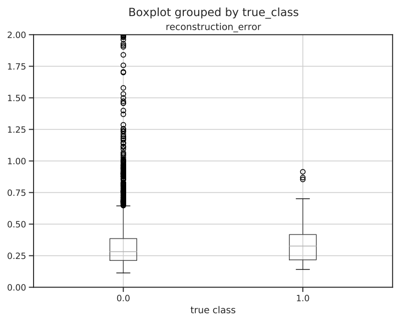
The inferencing for rare event prediction using an autoencoder is made by classifying the reconstruction errors as high or low. A high reconstruction error indicates the sample is anomalous to the normal process and, therefore, inferred as a rare event.

Listing 7.6. Autoencoder inferencing for rare event prediction

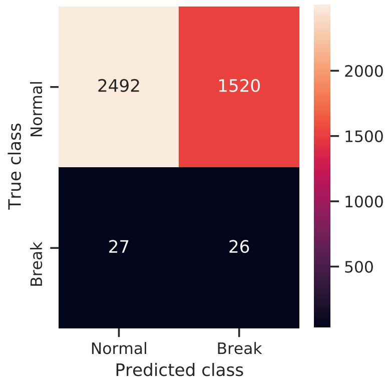
```
# Inferencing
error_vs_class_valid =
    rp.reconstructionerror_vs_class(
        model=autoencoder,
        sample=df_valid_x_rescaled,
        y=df_valid['y'])

# Boxplot
plt, fig = rp.error_boxplot(
    error_vs_class=error_vs_class_valid)

# Prediction confusion matrix
error_vs_class_test =
    rp.reconstructionerror_vs_class(
        model=autoencoder,
        sample=df_test_x_rescaled,
        y=df_test['y'])
conf_matrix, fig =
    rp.model_confusion_matrix(
        error_vs_class=error_vs_class_test,
        threshold=errors_valid1.quantile(0.50))
```

To determine the threshold of high vs low, boxplot statistics of the reconstruction error is determined for the positive and negative samples in the validation set. The boxplots are shown in Figure 7.4a.

From them, the 50-percentile of the positive sample errors is taken as the threshold for inferring a test sample as a rare event. The confusion matrix from test set predictions is shown in Figure 7.4b.



(a) *Boxplot.*



(b) *Confusion matrix.*

Figure 7.2. *Dense undercomplete autoencoder inferencing results. The boxplot of reconstruction errors for positive and negative samples in the validation set is at the top. The bottom figure shows the confusion matrix of the test inferences made based on the error threshold from the validation set.*

The test recall is as high as $\sim 50\%$ but at the cost of a high false-positive rate $\sim 38\%$. The f1-score is, therefore, as small as $\sim 3\%$.

A high false-positive rate may be undesirable in some problems. Another approach of using encodings learned from an autoencoder in a feed-forward classifier typically addresses the issue and shown in the next section.

7.5 Feed-forward MLP on Sparse Encodings

In this section, first, an overcomplete sparse autoencoder is constructed in § 7.5.1. Its encodings are then used in a feed-forward MLP classifier in § 7.5.2. This section shows that sparse autoencoders can learn useful features and help improve classification tasks.

7.5.1 Sparse Autoencoder Construction

A sparse autoencoder described in § 7.3.5 is constructed here. The model is overcomplete with the encoding dimension equal to the input dimension.

The sparsity regularization is imposed on the encodings by setting `activity_regularizer=tf.keras.regularizers.L1(l1=0.01)` in the model construction in Listing 7.7.

Listing 7.7. Sparse Autoencoder to derive predictive features

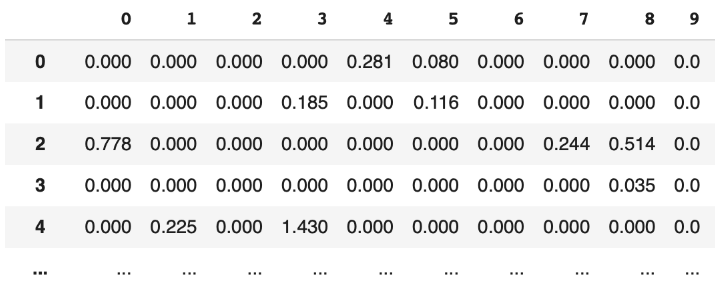
```python
# Sparse Autoencoder for Rare Event Detection
input_dim = df_train_0_x_rescaled.shape[1]

encoder = Dense(units=input_dim,
                activation="relu",
                input_shape=(input_dim,),
                use_bias = True,
                kernel_constraint=
                  UnitNorm(axis=0),
                activity_regularizer=
                  tf.keras.regularizers.L1(
                      l1=0.01),
                name='encoder')

decoder = Dense(units=input_dim,
                activation="linear",
                use_bias = True,
                kernel_constraint=
                  UnitNorm(axis=1), name='decoder')

sparse_autoencoder = Sequential()
sparse_autoencoder.add(encoder)
sparse_autoencoder.add(decoder)

sparse_autoencoder.summary()
sparse_autoencoder.compile(
    metrics=['accuracy'],
    loss='mean_squared_error',
    optimizer='adam')

history = sparse_autoencoder.fit(
    x=df_train_0_x_rescaled,
    y=df_train_0_x_rescaled,
    batch_size=128,
    epochs=100,
    validation_data=(df_valid_0_x_rescaled,
                     df_valid_0_x_rescaled),
    verbose=0).history
```

This regularization makes the autoencoder's loss function as in Equation 7.16. Due to the sparsity penalty in loss, the encoder weights shown in Listing 7.8 are not a trivial identity function even though the autoencoder is overcomplete (refer to the issue of trivial encodings in § 7.3.2).

Listing 7.8. Encoder weights are not identity in an overcomplete sparse autoencoder

```
# Weights on encoder
w_encoder = np.round(np.transpose(
    autoencoder.get_layer('encoder').get_weights()
        [0]), 3)

# Encoder weights
#  [[ 0.088  0.063  0.009 ...  0.096 -0.261  0.103]
#  [ 0.076  0.227  0.071 ...  0.083 -0.007  0.094]
#  [ 0.055 -0.1   -0.218 ... -0.019 -0.067  0.011]
#  ...
#  [ 0.039 -0.042  0.078 ... -0.216 -0.066  0.178]
#  [ 0.088 -0.169  0.084 ... -0.333 -0.109  0.171]
#  [ 0.057  0.081 -0.086 ...  0.26   0.201 -0.1  ]]
```

Moreover, the effect of the encodings sparsity penalty is visible in Figure 7.3. The figure shows several encodings are pushed to zero. In doing so, the autoencoder learns only the essential data features.



	0	1	2	3	4	5	6	7	8	9
0	0.000	0.000	0.000	0.000	0.281	0.080	0.000	0.000	0.000	0.0
1	0.000	0.000	0.000	0.185	0.000	0.116	0.000	0.000	0.000	0.0
2	0.778	0.000	0.000	0.000	0.000	0.000	0.000	0.244	0.514	0.0
3	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.035	0.0
4	0.000	0.225	0.000	1.430	0.000	0.000	0.000	0.000	0.000	0.0
...	...	...	...	...	...	...	...	...	...	...

Figure 7.3. *Encodings of the sparse autoencoder.*

The sparse encodings did prove to improve the model's performance. The boxplots in Figure 7.4a show a better separation of positive and negative samples compared to the previous model. The confusion matrix in Figure 7.4b also shows an improvement.

With the sparse autoencoder, the recall increased to $\sim$ 60%, the false-positive rate decreased to $\sim$ 34%, and f1-score increased to $\sim$ 4%. The false-positive rate is still high. The next section shows a feed-forward MLP on the sparse encodings to resolve the issue.

7.5.2 MLP Classifier on Encodings

The encodings learned in sparse autoencoders typically make excellent input features to a feed-forward classifier in deep learning.

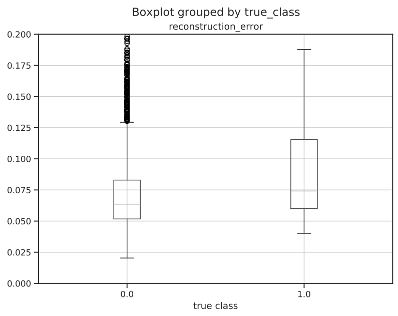
In this section, a two-stage approach: 1. learn a sparse encoder, and 2. train a classifier on the encodings, is demonstrated.

Listing 7.9 first derives the encodings of the training, validation, and test $\boldsymbol{x}$'s by passing them through the sparse encoder learned in the previous § 7.5.1.

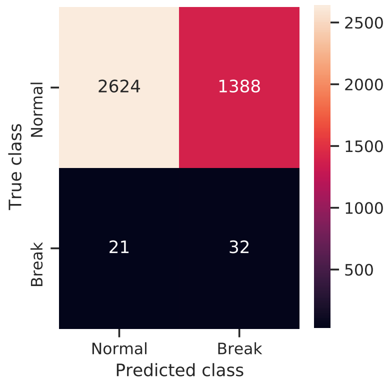
An MLP model with the same architecture as the baseline MLP in § 4.4 is constructed except that the network is trained on the encodings.

Listing 7.9. MLP feed-forward classifier on sparse encodings for rare event prediction

```
# Classifier on the sparse encodings

```



(a) *Boxplot.*



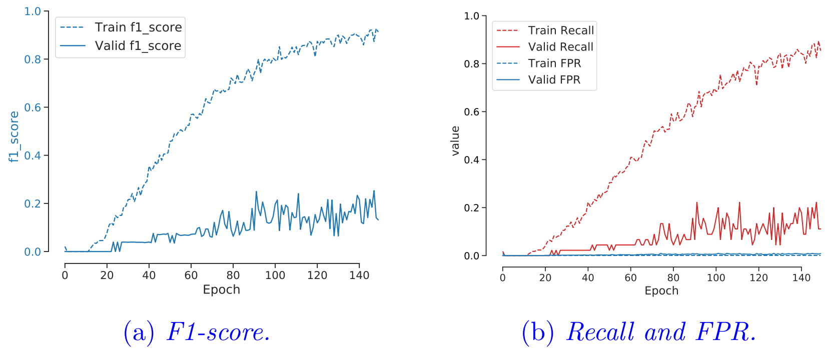
(b) *Confusion matrix.*

Figure 7.4. *Sparse overcomplete autoencoder inferencing results. The boxplot (top) shows the sparse autoencoder could better separate positive and negative samples. The confusion matrix (bottom) confirms it with an improvement in the accuracy measures.*

```
# Data preparation
import utilities.performancemetrics as pm
df_train_x = df_train.drop(['y'], axis=1)
df_train_y = df_train['y'].values

df_train_x_rescaled = scaler.transform(df_train_x)

df_valid_y = df_valid['y'].values
df_test_y = df_test['y'].values

```

```
# Obtain encodings to use as features in classifier
encoder_model =
    Model(inputs=sparse_autoencoder.inputs,
          outputs=sparse_autoencoder.get_layer('
             encoder').output)
X_train_encoded_features =
    encoder_model.predict(df_train_x_rescaled)
X_valid_encoded_features =
    encoder_model.predict(df_valid_x_rescaled)
X_test_encoded_features =
    encoder_model.predict(df_test_x_rescaled)

# Model
classifier = Sequential()
classifier.add(Input(
    shape=(X_train_encoded_features.shape[1], )))
classifier.add(Dense(units=32,
                     activation='relu'))
classifier.add(Dense(units=16,
                     activation='relu'))
classifier.add(Dense(units=1,
                     activation='sigmoid'))

classifier.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy',
                     tf.keras.metrics.Recall(),
                     pm.F1Score(),
                     pm.FalsePositiveRate()]
           )

history = classifier.fit(
    x=X_train_encoded_features,
    y=df_train_y,
    batch_size=128,
    epochs=150,
    validation_data=(X_valid_encoded_features,
                     df_valid_y),
    verbose=0).history
```

The model's f1-score, recall, and FPR are shown in Figure 7.5a and 7.5b. These accuracy measures when compared with the MLP model results on original data in Figure 4.5b and 4.5c show a clear improvement. The f1-score increased from $\sim 10\%$ to $\sim 20\%$, recall increased from $\sim 5\%$ to $\sim 20\%$, and false-positive remained close to zero.



(a) *F1-score.* (b) *Recall and FPR.*

Figure 7.5. *MLP feed-forward classifier on sparse encodings results.*

7.6 Temporal Autoencoder

A dense autoencoder encodes flat data. Sometimes it is required to encode them as sequences or time series. Sequence constructs such as LSTM and Convolution can be used to build autoencoders for them.

However, there are a few differences in this autoencoder construction compared to a dense autoencoder. They are discussed below along with constructing LSTM and convolutional autoencoders.

7.6.1 LSTM Autoencoder

An LSTM autoencoder can be used for sequence reconstruction, sequence-to-sequence translation (Sutskever, Vinyals, and Le 2014), or feature extraction for other tasks. The simple sparse autoencoder is constructed here in Listing 7.10 which can be modified based on the objective.

In this model,

- an overcomplete autoencoder is modeled.
- The model is split into `encoder` and `decoder` modules.

- The encoder has the same number of units as the number of input features (overcompleteness).
- The LSTM layer output in the encoder is flattened (in line 18). This is to provide encoded vectors. This is optional.
- The encoded vector is L_1-regularized in line 22 for sparsity.
- The encoder and decoder modules end with a linearly activated `Dense` layer. The ending dense layer is not for feature extraction. Instead, its task is similar to **calibration**. Complex layers such as LSTM and convolution yield features that may be in a different space than the original data. A linear dense layer brings these features back to the original space for reconstruction.
- The ending `Dense` layer in the decoder is applied directly to the LSTM sequence outputs. Flattening the sequences is unnecessary because the dense layer automatically estimates a weights tensor compatible with the shape of the sequences.

Listing 7.10. A sparse LSTM autoencoder

```
 1  # LSTM Autoencoder Model
 2  # Encoder
 3
 4  inputs = Input(shape=(TIMESTEPS,
 5                        N_FEATURES),
 6                 name='encoder-input')
 7
 8  x = LSTM(units=N_FEATURES,
 9           activation='tanh',
10           return_sequences=True,
11           name='encoder-lstm')(inputs)
12
13  # Shape info needed to build Decoder Model
14  e_shape = tf.keras.backend.int_shape(x)
15  latent_dim = e_shape[1] * e_shape[2]
16
17  # Generate the latent vector
18  x = Flatten(name='flatten')(x)
19  latent = Dense(units=latent_dim,
20                 activation='linear',
               activity_regularizer=
                  tf.keras.regularizers.L1(l1=0.01),
               name='encoded-vector')(x)

# Instantiate Encoder Model
encoder = Model(inputs=inputs,
                outputs=latent,
                name='encoder')
encoder.summary()

# Decoder
latent_inputs = Input(shape=(latent_dim,),
                      name='decoder_input')

x = Reshape((e_shape[1], e_shape[2]),
            name='reshape')(latent_inputs)

x = LSTM(units=N_FEATURES,
         activation='tanh',
         return_sequences=True,
         name='decoder-lstm')(x)

output = Dense(units=N_FEATURES,
               activation='linear',
               name='decoded-sequences')(x)

# Instantiate Decoder Model
decoder = Model(inputs=latent_inputs,
                outputs=output,
                name='decoder')
decoder.summary()

# Instantiate Autoencoder Model using Input and Output
autoencoder = Model(inputs=inputs,
                    outputs=decoder(inputs=encoder(inputs)),
                    name='autoencoder')
autoencoder.summary()
```

Linearly activated dense layer in encoder and decoder is necessary for calibration.

7.6.2 Convolutional Autoencoder

An autoencoder that encodes data using convolutions is called a **convolutional autoencoder**. Unlike most other autoencoders, a convolutional autoencoder uses different types of convolution layers for encoding and decoding. This and other specifications of a convolutional autoencoder constructed in Listing 7.11 are given below.

- The encoder and decoder modules are encapsulated within definitions. The encapsulation approach makes it easier to construct more complex modules. Moreover, the benefit is in visualizing the network structure as shown in Figure 7.6.
- A sparsity constraint on the encodings (for a sparse autoencoder) is added in the encoder in line 44.
- The decoder in a convolutional autoencoder cannot be made with `Conv` layers. A convolution layer is meant for extracting abstract features. In doing so, it downsizes the input. A decoder, on the other hand, is meant to reconstruct the input by upsizing (encoded) features. This is possible with `ConvTranspose` layers, also known as the **deconvolutional** layer (refer Shi, Caballero, Theis, et al. 2016).
- Pooling should strictly be not used in the decoder. Pooling is for data summarization. But decoding is for reconstruction. Any summarization with pooling can obstruct reconstruction.
- Decoding with (transpose) convolutional layers sometimes causes feature inflation. A batch normalization layer is usually added to stabilize the reconstruction.
- Similar to an LSTM autoencoder and for the same reason, a `Dense` layer is added at the end of both encoder and decoder. Its purpose is to **calibrate** the encoding and decoding.

🔔 *`Pooling` and `Conv` layers should **not** be used in a decoder of convolutional autoencoders.*

🔔 *`ConvTranspose` is used in place of a `Conv` layer in a decoder of convolutional autoencoders.*

Listing 7.11. A sparse convolutional autoencoder

```
inputs = Input(shape=(TIMESTEPS,
                      N_FEATURES),
               name='encoder_input')

def encoder(inp):
  '''
  Encoder.

  Input
  inp    A tensor of input data.

  Process
  Extract the essential features of the input as
  its encodings by filtering it through
  convolutional layer(s). Pooling can also
  be used to further summarize the features.

  A linearly activated dense layer is added as the
  final layer in encoding to perform any affine
  transformation required. The dense layer is not
  for any feature extraction. Instead, it is only
  to make the encoding and decoding connections
  simpler for training.

  Output
  encoding   A tensor of encodings.
  '''

  # Multiple (conv, pool) blocks can be added here
  conv1 = Conv1D(filters=N_FEATURES,
                 kernel_size=4,
                 activation='relu',
                 padding='same',
                 name='encoder-conv1')(inp)
  pool1 = MaxPool1D(pool_size=4,
                    strides=1,
                    padding='same',
                    name='encoder-pool1')(conv1)

  # The last layer in encoding
  encoding = Dense(units=N_FEATURES,
                   activation='linear',
                   activity_regularizer=
                    tf.keras.regularizers.L1(l1
                       =0.01),
                   name='encoder')(pool1)

  return encoding

def decoder(encoding):
  '''
  Decoder.

  Input
  encoding    The encoded data.

  Process
  The decoding process requires a transposed
  convolutional layer, a.k.a. a deconvolution
  layer. Decoding must not be done with a
  regular convolutional layer. A regular conv
  layer is meant to extract a downsampled
  feature map. Decoding, on the other hand,
  is reconstruction of the original data from
  the downsampled feature map. A regular
  convolutional layer would try to extract
  further higher level features from
  the encodings instead of a reconstruction.

  For a similar reason, pooling must not be
  used in a decoder. A pooling operation is
  for summarizing a data into a few summary
```

```
    statistics which is useful in tasks such as
    classification. The purpose of decoding is
    the opposite, i.e., reconstruct the original
    data from the summarizations. Adding pooling
    in a decoder makes it lose the variations
    in the data and, hence, a poor reconstruction.

    If the purpose is only reconstruction, a
    linear activation should be used in decoding.
    A nonlinear activation is useful for
    predictive features but not for reconstruction.

    Batch normalization helps a decoder by
    preventing the reconstructions
    from exploding.

    Output
    decoding    The decoded data.

    '''

    convT1 = Conv1DTranspose(filters=N_FEATURES,
                             kernel_size=4,
                             activation='linear',
                             padding='same')(encoding)

    decoding = BatchNormalization()(convT1)

    decoding = Dense(units=N_FEATURES,
                     activation='linear',
                     name='decoder')(decoding)

    return decoding

autoencoder = Model(inputs=inputs,
                    outputs=decoder(encoder(inputs))
                    )

autoencoder.summary()

autoencoder.compile(loss='mean_squared_error',
                    optimizer = 'adam')
```

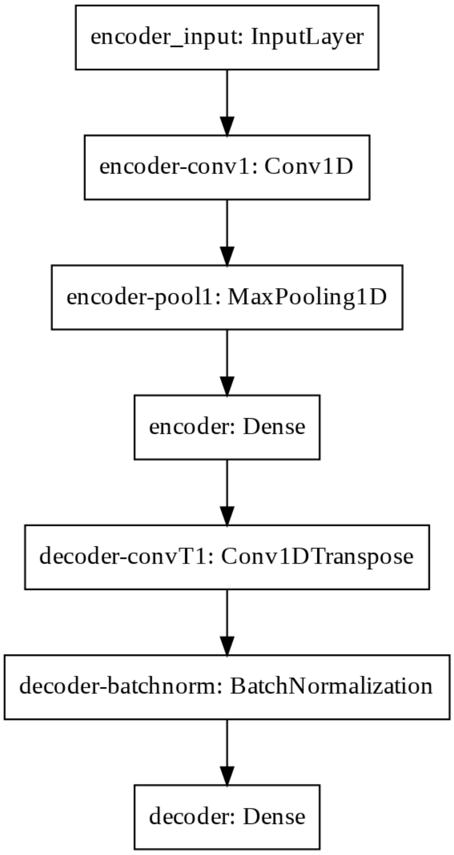

encoder_input: InputLayer
encoder-conv1: Conv1D
encoder-pool1: MaxPooling1D
encoder: Dense
decoder-convT1: Conv1DTranspose
decoder-batchnorm: BatchNormalization
decoder: Dense

Figure 7.6. *A baseline convolutional autoencoder.*

```

history = autoencoder.fit(x=X_train_y0_scaled,
                          y=X_train_y0_scaled,
                          epochs=100,
                          batch_size=128,
                          validation_data=
                              (X_valid_y0_scaled,
                               X_valid_y0_scaled),
                          verbose=1).history
```

A convolutional autoencoder can be used for image reconstruction, image denoising, or, like the other autoencoders, feature extraction for other tasks (Shi, Caballero, Huszár, et al. 2016).

In this chapter, three different programmatic paradigms in TensorFlow is used for constructing a dense, an LSTM, and a convolutional autoencoder. The paradigms are interchangeable for simple models. However, the functional approach used for the convolutional autoencoder in Listing 7.11 is preferable.

7.7 Autoencoder Customization

Autoencoders have proven to be useful for unsupervised and semi-supervised learning. Earlier, § 7.3 presented a variety of ways autoencoders can be modeled. Still, there is significant room for new development.

The section is intended for researchers seeking new development. It presents an autoencoder customization idea in § 7.7.1. A customized autoencoder is then constructed in § 7.7.2-7.7.4.

7.7.1 Well-posed Autoencoder

A mathematically well-posed autoencoder is easier to tune and optimize. The structure of a well-posed autoencoder can be defined from its relationship with principal component analysis.

As explained in the previous § 7.2.1 and 7.2.2, a linearly activated autoencoder approximates PCA. And, conversely, autoencoders are a nonlinear extension of PCA. In other words, an autoencoder extends PCA to a nonlinear space. Therefore, an Autoencoder should ideally have the properties of PCA. These properties are,

- **Orthonormal weights**. It is defined as follows for encoder weights,

$$W^T W = I, \text{and} \tag{7.18}$$

$$\sum_{j=1}^{p} w_{ij}^2 = 1,\ i = 1, \ldots, k \tag{7.19}$$

where I is a $p \times p$ identity matrix, p is the number of input features, and k is the number of nodes in an encoder layer.

- **Independent features**. The principal component analysis yields independent features. This can be seen by computing the covariance of the principal scores $Z = XW$,

$$\begin{aligned}\text{cov}(Z) &\propto (XW)^T(XW) \\ &= W^T X^T XW \\ &\propto W^T W \Lambda W^T W \\ &= \Lambda\end{aligned}$$

  where Λ is a diagonal matrix of eigenvalues and W are the eigenvectors. But autoencoder weights are **not** necessarily eigenvectors. Therefore, this property is not present by default and can be incorporated by adding a constraint,

$$\text{correlation}(Z_{encoder}) = I. \tag{7.20}$$

- **Tied layer**. An autoencoder typically has an hour-glass like symmetric structure[3]. In such a network, there is a mirror-layer in the decoder for every layer in the encoder. These layers can be tied by their weights and activations as

$$W^{(-l)} = (W^{(l)})^T \tag{7.21}$$
$$f^{(-l)} = (f^{(l)})^{-1} \tag{7.22}$$

  where $W^{(l)}$ and $f^{(l)}$ are the weights and activation on the l-th encoder layer, and $W^{(-l)}$ and $f^{(-l)}$ are on its mirror layer in the decoder. The weights transpose relationship in Equation 7.21 comes from Equation 7.6. The activations' inverse relationship is required due to their nonlinearity.

[3]An autoencoder does not necessarily have an hour-glass like structure. The decoder can be structured differently than the encoder.

Tying the weights without the activations inverse relationship can cause poor reconstruction. Consider a simple autoencoder flow to understand this issue:

$$X \rightarrow \underbrace{f^{(l)}(XW^{(l)})}_{encoder} \rightarrow Z \rightarrow \underbrace{f^{(-l)}(ZW^{(-l)})}_{decoder} \rightarrow \hat{X}.$$

In this flow, if $W^{(-l)}$ becomes $W(^{(l)})^T$ then $f^{(-l)}$ must become the inverse of $f^{(l)}$. Otherwise, the model is improperly posed. For example, if $f^{(-l)}$ is the same as $f^{(l)}$ then the model will be expected to yield the estimation of the input X using the same weights learned for encoding and the same nonlinear activation but $X \rightarrow f^{(l)}(XW^{(l)}) \rightarrow Z \rightarrow f^{(l)}(Z(W^{(l)})^T) \not\rightarrow \hat{X}$

Therefore, layers can only be tied in presence of a nonlinear activation and its inverse. However, defining such activation is nontrivial. In its absence, layers can be tied only if they are linearly activated in which case we lose the multilayer and nonlinear benefits of autoencoders.

7.7.2 Model Construction

In this section, an autoencoder with sparse encoding representation covariance (the features independence property in § 7.7.1) and orthonormal weights regularization is constructed and fitted on a random data. The custom definitions for these properties and the regularization effects are then illustrated in § 7.7.3 and 7.7.4.

The model is constructed and fitted on random data. The encoder layer has `kernel_regularizer`, `kernel_constraint`, and `activity_regularizer` defined. These are discussed in the next sections.

Listing 7.12. An illustrative example of a regularized autoencoder.

```python
# Generate Random Data for Testing
n_dim = 5

# Generate a positive definite
# symmetric matrix to be used as
# covariance to generate a random data.
cov = datasets.make_spd_matrix(n_dim,
                               random_state=None)

# Generate a vector of mean for generating the
    random data.
mu = np.random.normal(loc=0,
                      scale=0.1,
                      size=n_dim)

# Generate the random data, X.
n = 1000
X = np.random.multivariate_normal(mean=mu,
                                  cov=cov,
                                  size=n)

# Autoencoder fitted on random data
nb_epoch = 100
batch_size = 16
input_dim = X.shape[1]
encoding_dim = 4
learning_rate = 1e-3

encoder = Dense(units=encoding_dim,
                activation="relu",
                input_shape=(input_dim, ),
                use_bias = True,
                kernel_regularizer=
                    OrthogonalWeights(
                        weightage=1.,
                        axis=0),
                kernel_constraint=
                    UnitNorm(axis=0),
                activity_regularizer=
                    SparseCovariance(
                        weightage=1.), name='encoder
                            ')

decoder = Dense(units=input_dim,
                activation="linear",
                use_bias = True,
                kernel_constraint=
                         UnitNorm(axis=1), name='decoder'
                             )

autoencoder = Sequential()
autoencoder.add(encoder)
autoencoder.add(decoder)

autoencoder.compile(metrics=['accuracy'],
                         loss='mean_squared_error',
                         optimizer='sgd')
autoencoder.summary()

autoencoder.fit(X, X,
                    epochs=nb_epoch,
                    batch_size=batch_size,
                    shuffle=True,
                    verbose=0)
```

7.7.3 Orthonormal Weights

Orthonormality of encoding weights is an essential property. Without it the model is ill-conditioned, i.e., a small change in the input can cause a significant change in the model.

As shown in Equation 7.18 and 7.19, this property has two parts: a. **orthogonality**, and b. **unit norm**.

The latter is easily incorporated in an encoder layer using `kernel_constraint=UnitNorm(axis=0)` constraint[4].

The former property can be incorporated with a custom regularizer defined in Listing 7.13. This is used in `kernel_regularizer` that acts as a soft constraint. Meaning, the estimated weights will be only **nearly** orthogonal.

The weights can be made strictly orthogonal by an input covariance decomposition. But due to decomposition computational complexity, a regularization method is preferred.

[4]In a decoder layer, the same constraint should be applied on the columns as `kernel_constraint=UnitNorm(axis=1)`.

Listing 7.13. A custom constraint for orthogonal weights.

```python
# Orthogonal Weights.
class OrthogonalWeights (Constraint):
    def __init__(self,
                 weightage = 1.0,
                 axis = 0):
        self.weightage = weightage
        self.axis = axis

    def weights_orthogonality(self,
                              w):
        if(self.axis==1):
            w = K.transpose(w)

        wTwminusI = K.dot(K.transpose(w), w) -
            tf.eye(tf.shape(w,
                    out_type=tf.float32)[1])

        return self.weightage * tf.math.sqrt(
            tf.math.reduce_sum(tf.math.square(
                wTwminusI)))

    def __call__(self, w):
        return self.weights_orthogonality(w)
```

The learned encoder and decoder weights for the autoencoder in Listing 7.12 is shown in Listing 7.14.

Listing 7.14. Encoder and decoder weights on an illustrative regularized autoencoder

```python
w_encoder = np.round(autoencoder.get_layer('encoder'
    ).get_weights()[0], 3)
w_decoder = np.round(autoencoder.get_layer('decoder'
    ).get_weights()[1], 3)
print('Encoder weights\n', w_encoder.T)
print('Decoder weights\n', w_decoder.T)

# Encoder weights
#   [[-0.301 -0.459  0.56   -0.033  0.619]
#    [-0.553  0.182 -0.439 -0.644  0.231]
#    [-0.036  0.209 -0.474  0.61   0.599]
#   [-0.426 -0.683 -0.365  0.288 -0.367]]
# Decoder weights
#   [[-0.146 -0.061  0.436 -0.72   0.517]
#   [-0.561  0.199 -0.431 -0.633  0.242]
#   [-0.059  0.681 -0.345  0.613  0.194]
#   [ 0.086 -0.769 -0.22   0.198 -0.56 ]]
```

The (near) orthogonality of encoding weights is shown in Listing 7.15. The weights are nearly orthogonal with small non-zero off-diagonal elements in $W^T W$. As mentioned earlier, the orthogonality is added as a soft-constraint due to which $W^T W \approx I$ instead of strict equality.

Listing 7.15. Encoder weights dot product show their near orthogonality. The orthogonality regularization is not applied on the decoder—its dot product is therefore not diagonal heavy

```
w_encoder = autoencoder.get_layer('encoder').
    get_weights()[0]
print('Encoder weights dot product\n',
      np.round(np.dot(w_encoder.T, w_encoder), 2))

w_decoder = autoencoder.get_layer('decoder').
    get_weights()[1]
print('Decoder weights dot product\n',
      np.round(np.dot(w_decoder.T, w_decoder), 2))

# Encoder weights dot product
#   [[1.   0.   0.   0.  ]
#   [0.   1.   0.01 0.  ]
#   [0.   0.01 1.   0.  ]
#   [0.   0.   0.   1.  ]]
# Decoder weights dot product
#   [[ 1.    0.46 -0.52 -0.49]
#   [ 0.46  1.   -0.02 -0.37]
#   [-0.52 -0.02  1.   -0.44]
#   [-0.49 -0.37 -0.44  1.  ]]
```

The orthogonality regularization is not added to the decoder. Due to that, the decoder weights dot product is not diagonally dominant.

Besides, the unit-norm constraint is added on both encoder and decoder weights. Their norms are shown in Listing 7.16.

Listing 7.16. Norm of encoder and decoder weights

```
print('Encoder weights norm, \n',
      np.round(np.sum(w_encoder ** 2,
                      axis = 0),
               2))
print('Decoder weights norm, \n',
      np.round(np.sum(w_decoder ** 2,
                      axis = 1),
               2))

# Encoder weights norm,
#  [1. 1. 1. 1.]
# Decoder weights norm,
#  [1. 1. 1. 1.]
```

It is important to note that decoder unit-norm is applied along the columns. Also, if the orthogonality regularization is to be added on decoder, it should be such that $WW^T \approx I$ unlike in encoder (where it is $W^TW \approx I$).

7.7.4 Sparse Covariance

Independence of the encoded features (uncorrelated features) is another desired property. It is because correlated encoding means we have redundant information spilled over the encoded dimensions.

However, unlike in PCA, weights orthogonality does not necessarily result in independent features. This is because $Z^TZ = W^TX^TXW \neq I$ even if $W^TW = I$. Orthogonal principal components, on the other hand, leads to independent features because they are drawn from the matrix decomposition of the input, i.e., $X^TX = W\Lambda W^T$ where Λ is a diagonal eigenvalues matrix.

If the orthogonal encoder weights were drawn by covariance decomposition of the input, the features will be independent. However, due to the computational complexity, the (near) feature independence is incorporated by regularization.

An approach to incorporate this is in Listing 7.17. Similar to weight orthogonality, the feature independence is incorporated as a soft constraint to have a sparse covariance.

Listing 7.17. A custom regularization constraint for encoded feature covariance sparsity

```python
# Near-Independent Features
class SparseCovariance (Constraint):

    def __init__(self, weightage=1.0):
        self.weightage = weightage

    # Constraint penalty
    def uncorrelated_feature(self, x):
        if(self.size <= 1):
            return 0.0
        else:
            output = K.sum(K.square(
                self.covariance -
                tf.math.multiply(self.covariance,
                                 tf.eye(self.size)))
                                    )
            return output

    def __call__(self, x):
        self.size = x.shape[1]
        x_centered = K.transpose(tf.math.subtract(
            x, K.mean(x, axis=0, keepdims=True)))

        self.covariance = K.dot(
            x_centered,
            K.transpose(x_centered)) / \
                tf.cast(x_centered.get_shape()[0],
            tf.float32)

        return self.weightage * self.
           uncorrelated_feature(x)
```

The covariance of the encoded random data is shown in Listing 7.18. As expected, the covariance is sparse. Moreover, similar to PCA behavior, most of the input data variability is taken by a small number of encodings. In this example, the last encoding dimension has more than 99% of the variance.