Chapter 6

Deep Latent Variable Models

In the previous chapter, we discussed how autoregressive models leverage previous values in a sequence to represent highly complex, high dimensional probability distributions. In contrast, latent variable models adopt a distinct approach to approximating the probability distribution p_{data}. Latent variable models do not use previous values in the sequence as AR models do; instead, they assume the existence of an unobservable hidden variable that influences every data point in the dataset.

In this chapter, we focus on models where the knowledge of the hidden or latent variables **partially explains the observed data.** Here, the relationship between the hidden variables and the observations is not fully deterministic—we introduce some uncertainty in the modeling step regarding how the latent variables generate the observed data. On the other hand, models where the latent variables completely determine the observed data—where a deterministic, invertible mapping exists between the hidden variables and the data—will be the subject of Chapter 7. These are known as flow models or invertible models. In Chapter 8, we will introduce another type of latent variable model: Generative Adversarial Networks (GANs). Unlike the models discussed here or in the next chapter, GANs do not explicitly model the probability distribution of the data.

In this chapter, we focus on models where the dimensionality of this latent variable is significantly smaller than that of the observed data. Introducing hidden variables in our models helps model a high-dimensional multimodal distribution as the integral of the product of two simpler distributions. Commonly, these simpler distributions belong to the exponential family, which simplifies the overall modeling process while retaining flexibility in capturing the underlying data structure. The Hidden Markov

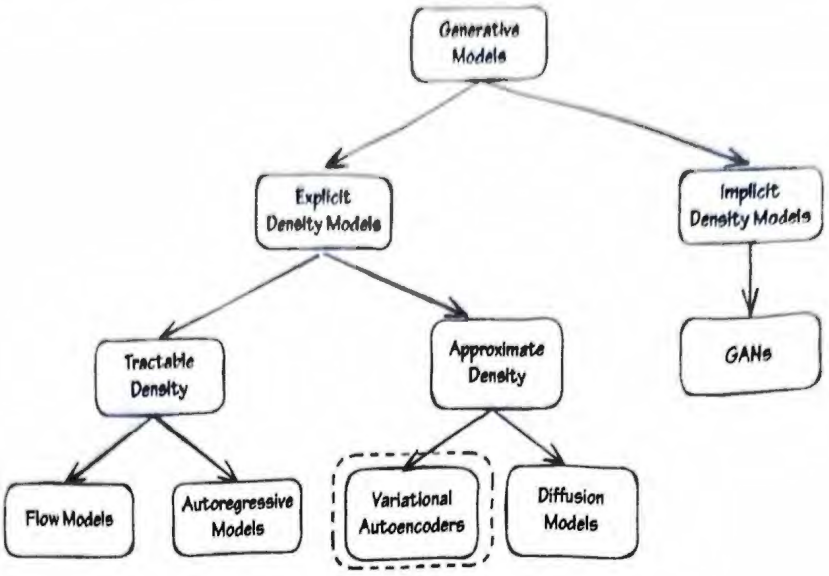

Figure 6.1 Model taxonomy: variational autoencoders.

Model that we introduced in Chapter 3 is obviously one simple example of such latent variables model (where there is some dynamic between the latent variables), but we shall see that Principal Component Analysis (PCA) introduced there can also be viewed as an example of latent variables model. Deep latent variable models, such as the well-known variational autoencoders discussed in this chapter, belong to the category of explicit models that estimate densities approximately, as shown in Figure 6.1. This chapter will leverage the full machinery of deep learning to create a much richer variety of these models.

6.1 Introduction

To illustrate the utility of latent variable models, consider their application in finance, particularly for modeling and estimating the covariance matrix of multivariate time series. Covariance matrices are fundamental in finance, utilized for tasks such as regression analysis, risk estimation, portfolio optimization, and scenario simulation through Monte Carlo methods (López de Prado, 2020).

Let's assume we have a lengthy history of returns from D time series, each representing an asset's historical returns. A straightforward approach to modeling these multivariate time series is to ignore individual time-series serial correlation and assume a constant covariance across assets over time. At any time index t, the returns could be modelled as a draw from a constant probability distribution, for example, $\mathbf{x}_t \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\mathbf{x}_t \in \mathbb{R}^D$ represent the return at time index t, $\boldsymbol{\mu}$ is the mean return vector, and $\boldsymbol{\Sigma}$ is the covariance matrix of returns.

To provide context, consider modeling the daily returns of the S&P 500 index components, thereby setting the number of time series to $D = 500$. Here, the simplest estimator for the covariance matrix would require computing $D(D+1)/2 = 500(501)/2 = 125{,}250$

covariance values. This large number of parameters can become problematic if the available daily data are limited.

Another way to think about modeling $\mathbf{x}_t$ is to assume it is influenced by a smaller set of underlying, unobserved factors that partially explain the observations.

These factors may look similar to traditional multi-factor models, such as the Fama-French three-factor model we discussed in Chapter 2, which explains asset returns using factors like the market, Small Minus Big (SMB), and High Minus Low (HML) factors. However, a key difference between these traditional models and the ones we will explore in this chapter is that, in the traditional case, the factors are observable—they can be measured, and often, they can even be traded, providing significant value to traders. In contrast, the models introduced in this chapter deal with unobservable factors. These are hidden, or latent, variables, terms we will use interchangeably in this chapter, and which are common in the literature.

For instance, Factor Analysis (FA), a type of latent variable model, decomposes the covariance matrix as $\mathbf{\Sigma} = \Lambda\Lambda^T + \mathbf{\Sigma}_0$, where $\Lambda \in \mathbb{R}^{D \times M}$ is known as the loading matrix, M being the number of factors (a hyperparameter in FA) with $M \ll D$, and $\mathbf{\Sigma}_0$ is a $D \times D$ diagonal matrix. In financial terminology, we can think of $\Lambda\Lambda^T$ as the systematic covariance matrix, representing the part of the asset's returns variance explained by the factors, and Σ_0 as the idiosyncratic covariance matrix—the variance of asset's returns not explained by the factors. This type of variance decomposition is particularly useful for risk modeling in portfolio construction applications, where we might be interested in reducing the exposure of our strategy to certain factors or placing an upper limit on systematic variance, etc. This formulation of variance decomposition allows constraints like these to be easily incorporated into the portfolio optimization process.

This method significantly reduces the number of parameters to estimate, from $D(D+1)/2$ to $D \times M + D$, from $\mathcal{O}(D^2)$ to $\mathcal{O}(D \times M)$. For example, if $M = 10$, the parameter count drops to 5,010 compared to 125,250—a 25-fold reduction. Because this modeling approach assumes that observations are explained by a smaller set of factors, the covariance matrix is inherently low rank. This not only reduces the number of parameters to estimate but also decreases the memory required to store the covariance matrix compared to simpler modeling approaches.

What is fascinating about latent variable models is that, even though the factors are not directly observed, by making probabilistic assumptions about how they behave, we can infer their values and estimate the associated model parameters. Hamlet remembers when he first studied these kinds of models; I was absolutely fascinated by them. In the context of FA, things get even more interesting when we introduce dynamics into the latent variables, leading to models such as the well-known Kalman Filter.

As we will see in this chapter, when we consider Deep Latent Variable Models, the algorithms used to estimate these models become even more interesting. The concepts and tools required to develop them are applied in many fields, including statistics, machine learning, optimization, information theory, etc., and have applications across many areas.

6.2 Latent Variable Models

Latent variable models aim to represent high-dimensional data distributions using two simpler distributions. The model involves a latent variable $\mathbf{z}$ and an observed variable $\mathbf{x}$, where $\mathbf{z}$ is an M-dimensional random vector and $\mathbf{x}$ is a D-dimensional random vector. By design, we choose $M < D$ to ensure a lower dimensionality for the latent space.

A latent variable model is defined by the joint probability distribution of the latent and observed variables as $p(\mathbf{z}, \mathbf{x}) = p(\mathbf{z})p(\mathbf{x} \mid \mathbf{z})$. The model is specified by two components: a probability distribution over the latent variables, $p(\mathbf{z})$, called the prior distribution, and a conditional distribution over the observed data given the latent variables, $p(\mathbf{x} \mid \mathbf{z})$, which maps $\mathbf{z}$ from the latent space to the data space. Sampling in latent variable models is relatively straightforward, as it follows the data generation process described by:

$$\begin{aligned} \mathbf{z}_t &\sim p(\mathbf{z}_t) \\ \mathbf{x}_t \mid \mathbf{z}_t &\sim p(\mathbf{x}_t \mid \mathbf{z}_t) \end{aligned} \tag{6.1}$$

The challenge lies in choosing the prior and conditional distributions to represent our hypotheses about the data generation process. These choices also determine the specific methods used for performing latent variable inference and parameter estimation.

As examples of these particular choices, we will briefly examine traditional latent variable models such as Factor Analysis (FA), Probabilistic Principal Component Analysis (PPCA), and Gaussian Mixture Models (GMM) before moving on to the main topic of this chapter: Deep Latent Variable Models.

6.3 Examples of Traditional Latent Variable Models

6.3.1 Factor Analysis

As we have already seen, Factor Analysis (FA), see Bishop (2006), is a foundational model that expresses the modeling assumption that data can be explained by a reduced number of factors. This approach has interesting applications, such as covariance matrix estimation and dimensionality reduction we just saw, among others. Similarly, PPCA, which we will also examine later, builds on this probabilistic framework. In this section, we will go into the probabilistic formulation of these methods.

In FA, we assume that both the marginal and conditional distributions are Gaussian and are defined as follows:

$$\begin{aligned} p(\mathbf{z}_t) &= \mathcal{N}(\mathbf{z}_t; \mathbf{0}, \mathbf{I}) \\ p(\mathbf{x}_t \mid \mathbf{z}_t) &= \mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu} + \boldsymbol{\Lambda}\mathbf{z}_t, \boldsymbol{\Sigma}_0) \end{aligned} \tag{6.2}$$

where $p(\mathbf{z}_t)$ is the prior distribution over the latent variable, and $\mathbf{z}_t$, and $p(\mathbf{x}_t \mid \mathbf{z}_t)$ represents the conditional probability over the latent variable $\mathbf{z}_t$ given the observed data $\mathbf{x}_t$.

In the FA literature, the matrix $\Lambda \in \mathbb{R}^{D \times M}$ is referred to as the loading matrix, and it captures the correlations between the observed variables. The vector of means $\boldsymbol{\mu} \in \mathbb{R}^D$ can be thought as a bias term. The matrix $\boldsymbol{\Sigma}_0$ is diagonal with D elements and represents the noise covariance matrix. The values along the diagonal are the independent noise variances for each variable. In finance, this is commonly referred to as idiosyncratic variance, as described in Bodie et al. (2018).

To obtain the marginal distribution over $\mathbf{x}_t$, we integrate out all possible values of the latent variable $\mathbf{z}_t$:

$$p(\mathbf{x}_t) = \int p(\mathbf{x}_t, \mathbf{z}_t) d\mathbf{z}_t = \int p(\mathbf{x}_t | \mathbf{z}_t) p(\mathbf{z}_t) d\mathbf{z}_t$$

In FA, where both $p(\mathbf{x}_t)$ and $p(\mathbf{x}_t | \mathbf{z}_t)$ are Gaussian, this integral has a closed-form solution. However, a simpler way to compute the marginal distribution and to avoid the previous integral is to express $\mathbf{x}_t$ as:

$$\mathbf{x}_t = \boldsymbol{\mu} + \Lambda \mathbf{z}_t + \boldsymbol{\epsilon}_t$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_0)$.

Using the property that Gaussians are closed under addition and multiplication, the marginal distribution over $\mathbf{x}_t$ is also Gaussian. Therefore, we only need to compute the mean and covariance of $\mathbf{x}_t$, which are given by:

$$\begin{aligned} \mathbb{E}[\mathbf{x}] &= \boldsymbol{\mu} \\ \text{Cov}[\mathbf{x}] &= \boldsymbol{\Sigma} = \Lambda \Lambda^T + \boldsymbol{\Sigma}_0 \end{aligned}$$

Thus, the marginal distribution over $\mathbf{x}_t$ is:

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}),$$

where $\boldsymbol{\Sigma} = \Lambda \Lambda^T + \boldsymbol{\Sigma}_0$, retrieving the formulation presented in the introduction of this chapter.

Parameter estimation in this model follows a well-known approach: the parameters $\boldsymbol{\mu}$, Λ, and $\boldsymbol{\Sigma}_0$ are estimated from data using the Expectation-Maximization (EM) algorithm. For more details on parameter estimation for FA, see Bishop (2006).

6.3.2 *Probabilistic Principal Component Analysis*

Probabilistic Principal Component Analysis, or Probabilistic PCA (PPCA), is the probabilistic version of PCA proposed by Bishop and Tipping (2001). As we know, PCA is a widely used technique in data analysis and processing, yet it does not rely on a probabilistic model. Particularly in finance, PCA is employed to identify statistical factors that can be utilized for asset prediction (Chan, 2017) or risk modeling, where these factors have shown performance comparable to fundamental ones (see Paleologo [2021]).

This model is closely related to the Factor Analysis. In PPCA, both the prior and conditional distributions are Gaussian and are defined by the same equations as in Equation 6.2. However, the key difference between the two methods is that PPCA assumes Σ_0 is a constant diagonal matrix, denoted as $\Sigma_0 = \sigma^2 I$.

This assumption provides a computational advantage while preserving the interpretability of Factor Analysis.

6.3.2.1 Example: Comparing PCA and Factor Analysis for Latent Space Recovery. As an illustrative use case for Factor Analysis (FA) versus Probabilistic PCA (PPCA), let's explore an example adapted from the sklearn documentation (https://scikit-learn.org/stable/auto_examples/decomposition/plot_pca_vs_fa_model_selection.html). As we have seen, FA and PPCA are closely related: both assume Gaussian prior and Gaussian conditional distributions. The key difference lies in their treatment of **idiosyncratic variance**.

- PPCA assumes homoscedastic noise, meaning the variance is the same for all variables (identity covariance matrix).
- Factor Analysis assumes heteroscedastic noise, where each variable has its own variance (diagonal covariance matrix).

This example illustrates the ability of PCA (the limiting case of PPCA with zero variance) and Factor Analysis to recover the correct latent dimension in a low-rank simulated dataset with additive noise. We examine two cases, one with homoscedastic noise, with the same variance for each variable (identity covariance), and a second case with heteroscedastic noise, with different variance for each variable (diagonal covariance).

For the experiment, we generate 100-dimensional data with an underlying low-rank structure (latent dimension = 10).

Results regarding model selection for homoscedastic noise are illustrated in Figure 6.2. In this case, the low-rank data has been corrupted with noise that has equal variance across all variables. Both PCA (blue) and Factor Analysis (red) successfully recover the correct latent dimension of 10 when performing model selection.

On the other hand, results regarding model for heteroscedastic noise are illustrated in Figure 6.3. Here, the low-rank data are corrupted with noise that has varying variance across variables. This leads to a more interesting result:

- Factor Analysis (red) correctly identifies the latent dimension as 10, aligning with its assumption of heteroscedastic noise.
- PCA (blue), on the other hand, overestimates the latent dimension.

These results demonstrate how the assumptions behind each method influence their ability to capture the true latent structure in the presence of different noise characteristics.

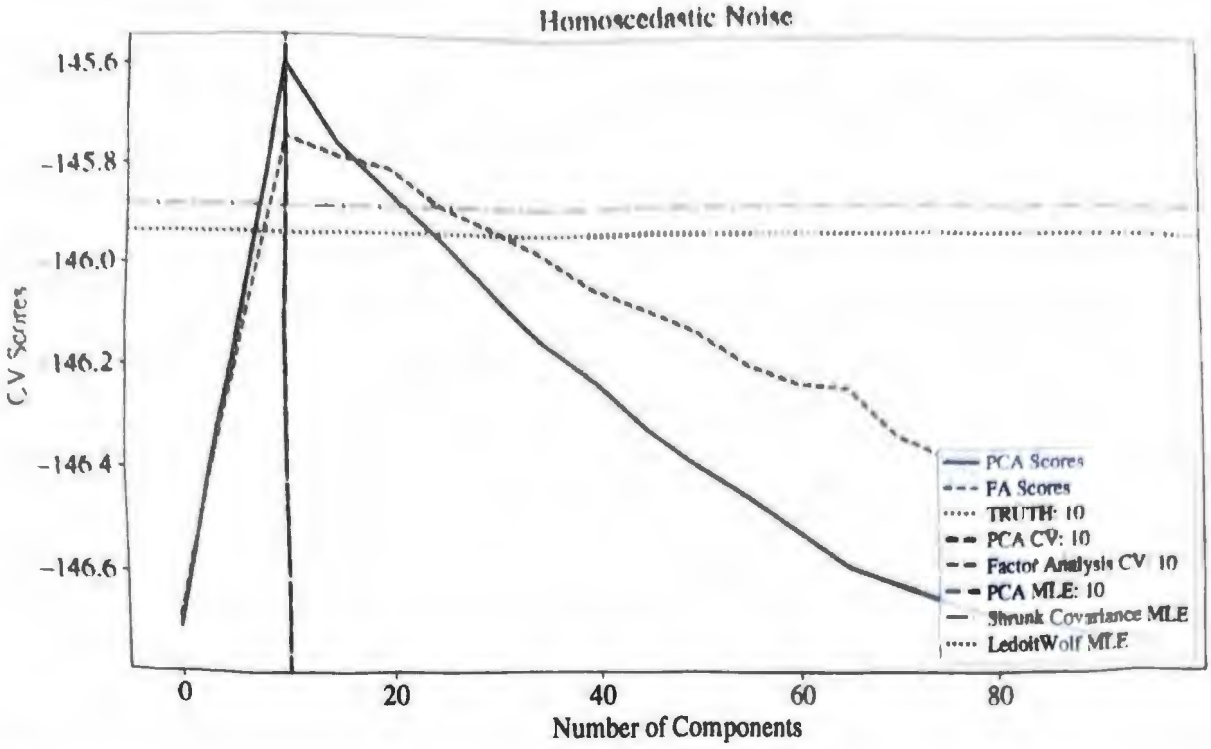

Figure 6.2 Illustration of model selection for homoscedastic noise.

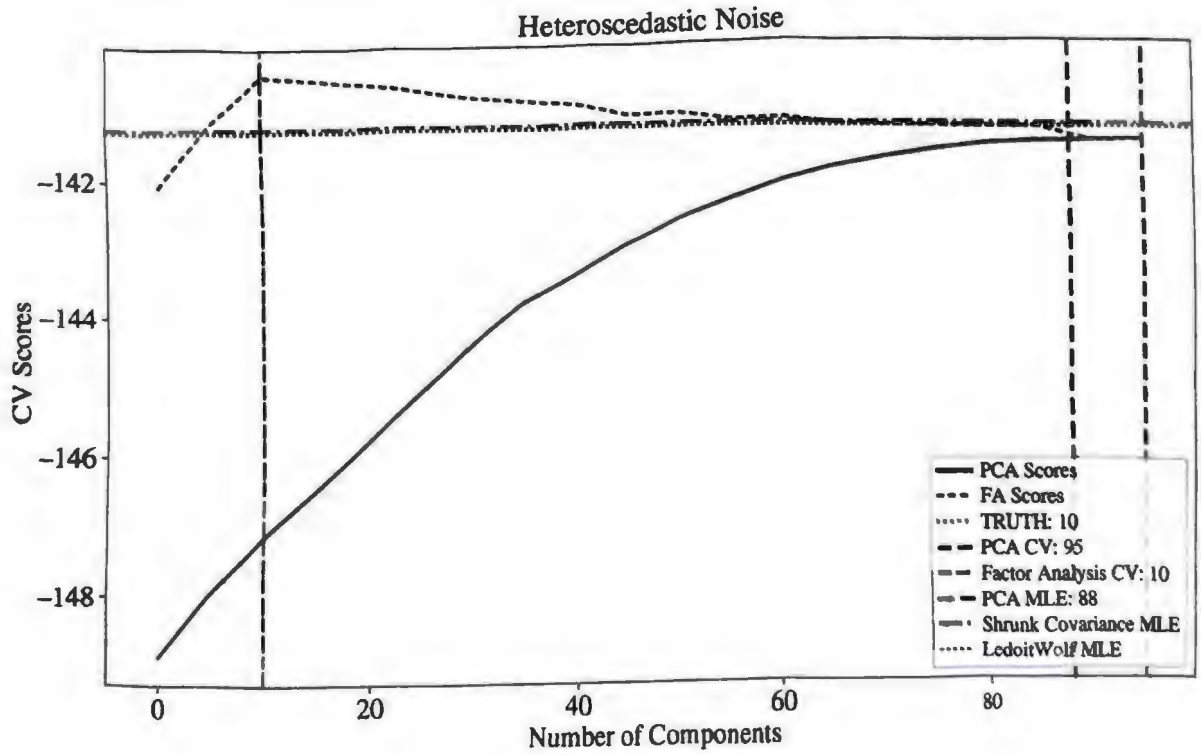

Figure 6.3 Illustration of model selection for heteroscedastic noise.

6.3.2.2 Advantages of Probabilistic Approaches (PPCA/FA) over PCA There are several advantages of using the probabilistic approach, either with PPCA or FA, over PCA, also called the non-probabilistic PCA. Some of the main ones include the following:

- **Estimation with Missing Data:** PPCA and FA provide a natural framework for parameter estimation when some entries of a vector in the dataset $\mathbf{x}_n \in \mathbb{R}^D$ are missing. This contrasts with the non-probabilistic PCA, where parameter estimation with missing data might requires data imputation techniques.
- **Mixtures of Probabilistic Principal Component Analysis Models:** Sometimes, a single PCA projection is insufficient to explain the data. Why not use a mixture of PCA models? This idea is analogous to Gaussian Mixture Models (GMM), detailed in the next section, where the data distribution is approximated using a mixture of Gaussians rather than a single Gaussian. Similarly, the probabilistic framework allows that a mixture of PPCA models can be easily implemented to better capture the structure of the data, which is an advantage over the non-probabilistic approach.
- **Data Generation and Likelihood Computation:** If your task involves data generation or the computation of densities (e.g., finding the likelihood of a point in your dataset), the probabilistic versions (PPCA/FA) are the way to go, as they explicitly model the data distribution.

For more details on these advantages, as well as the estimation procedures for FA, PPCA, and PCA using the Expectation-Maximization (EM) algorithm, refer to the excellent reference paper by Bishop and Tipping (2001) on PPCA.

In practice, choosing the appropriate method can be done using cross-validation or domain knowledge. Also, you might:

- Use explained variance to determine the dimension of the latent space.
- You might decide between PPCA and FA based on whether you believe the idiosyncratic variance (residual noise) of each variable should be equal (homoscedastic), requiring a single σ^2 to estimate, or different (heteroscedastic), requiring D variances to estimate $\sigma_1^2,...,\sigma_D^2$.
- For initial data exploration or modeling insight, you might use PCA, which is computationally more efficient to estimate than its probabilistic counterpart. Depending on your needs, PCA might offer a good trade-off between performance and computational complexity, especially when the amount of data to process is extremely large.

By understanding or making assumptions about the noise structure in our data we can make a better informed decision about our modeling choices.

6.3.3 Gaussians Mixture Models

In the Gaussian Mixture Model (GMM), the latent variable z is discrete, taking values in the set $\{1,...,K\}$, while the observed value $\mathbf{x}$ is continuous, taking values in $\mathbb{R}^D$. The latent variable follows a Categorical distribution, denoted as $p(z) = \text{Categorical}(1,...,K;\pi)$,

where $\boldsymbol{\pi} = [\pi_1, ..., \pi_K]$, and, for example, the probability that the variable z takes the value k is given by $p(z = k) = \pi_k$. The conditional distribution of $\mathbf{x}$ given z are Gaussian, defined by $p(\mathbf{x}|z = k) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$.

The mixture of Gaussian becomes apparent when computing the marginal over the observed variable by marginalizing the joint probability between the latent and observed variables over the latent variable, as follows:

$$\begin{aligned} p(\mathbf{x}) &= \sum_{k=1}^{K} p(z = k, \mathbf{x}) \\ &= \sum_{k=1}^{K} p(z = k) p(\mathbf{x} \mid z = k) \\ &= \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \end{aligned} \tag{6.3}$$

The result is simply a weighted sum of Gaussians (hence the adjective "mixture"). We can create different mixture models by altering the probability mass over the latent variables.

The posterior probability distribution of the latent variables given the observations quantifies the probability that the data point $\mathbf{x}$ was generated by the k-th mixture component. It is given by the conditional distribution:

$$p(z = k \mid \mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)} \tag{6.4}$$

For an illustrative example of GMM applied to clustering, see Figure 6.4. For more details on GMM see the excellent references Bishop (2006) and Murphy (2022).

6.3.3.1 Gaussian Mixture Model (GMM) for Market Regime Detection. Let's explore how Gaussian Mixture Models (GMM) can be applied to a well-known problem: detecting market regimes. As we know, markets change over time, and traders and investors often use domain knowledge to label certain periods. Common labels we hear in the press, on the news, or in interviews include terms like "bull market," "bear market," "trending market," or "mean-reverting market," and so on. In this case, however, we are going to take a data-driven approach to estimate market regimes.

Modeling market regimes is crucial for traders and investors, as for example, it might help them deploy the most appropriate strategy from a pool of potential strategies for the current market conditions or incorporate regime information into portfolio construction models for risk management. As we will see, GMM serves as a powerful building block for developing more sophisticated models that can be tailored to your specific market insights.

6.3.3.2 Example: Low and High Volatility Regimes. In this hypothetical scenario, based on domain knowledge, let's say we are interested in identifying two distinct clusters that may correspond to two different market regimes: one with low volatility

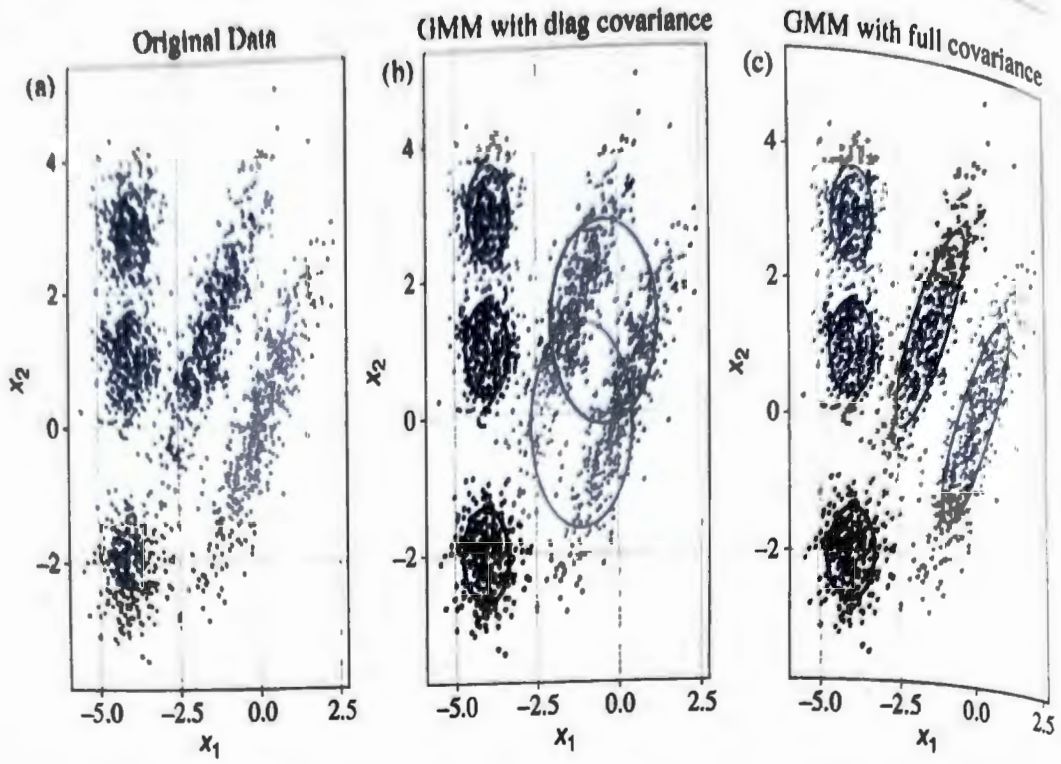

Figure 6.4 Illustration of clustering using Gaussian Mixture Models (GMMs). (a) Generated 2D data samples. (b) A possible clustering of the same data with $K = 5$ using GMMs with a diagonal covariance matrix. (c) A possible clustering with $K = 5$ using GMMs with a full covariance matrix. Cluster assignments (indicated by colors) are determined by selecting the cluster with the highest posterior probability for each point. Illustration based on Figure 3.12 in Murphy (2022), with additional details.

and the other with high volatility. These regimes might be identified entirely based on domain knowledge or through exploratory data analysis. For instance, it's unlikely that the returns of an asset like the SPY could be well explained over long periods using a single Gaussian distribution with constant mean and constant variance. Consider recent periods in history, such as the time before, during, and after the COVID-19 pandemic. During the pandemic, volatility was exceptionally high, while the months before and after exhibited much lower volatility. For simplicity, we might build a model to capture this high-volatility period separately from the low-volatility periods, which we could then visualize as "clusters" in the SPY returns time series. From an application point of view, identifying volatility regimes can help traders make decisions such as adjusting portfolio allocations, modifying leverage levels, or implementing volatility-specific trading strategies.

In the following example, we use the SPY ETF return data for market regime detection. In this setup, the latent variable z represents the market regime and can take two possible values: "low volatility" or "high volatility." Conditioned on the market regime (or state of z), we assume that returns follow a Gaussian distribution with regime-specific means and variances.

In this case, the data generation process can be described as follows. Since we have only two states for the latent variable, 0 or 1, at time t we draw the value of the state from a Bernoulli distribution with parameter π, where $P(z = 1) = \pi$:

$$z_t \sim \text{Bernoulli}(\pi)$$

Once the value of the latent variable is determined, at time t we generate data from the corresponding Gaussian distribution, with regime-specific means and variances, as follows:

$$x_t \sim (1 - z_t)\mathcal{N}(\mu_0, \sigma_0) + z_t\mathcal{N}(\mu_1, \sigma_1)$$

Here, the state $z = 0$ corresponds to the first cluster, or low-volatility regime, while $z = 1$ represents the second cluster, or high-volatility regime. In the accompanying notebook [*GMM-Market-Regime-Detection*], we will demonstrate how to estimate the parameters of this model, namely $\theta = \{\pi, \mu_0, \mu_1, \sigma_0, \sigma_1\}$, using data from the SPY ETF.

Figure 6.5 is a time-series plot of SPY returns from January 2000 to March 2024, along with the most probable market regimes estimated from the posterior probability distribution (see Equation 6.4) for this period. In this plot, individual points represent SPY returns, while vertical frames indicate the most probable regime at each time point. Light orange corresponds to regime 0 (low volatility), and blue corresponds to regime 1 (high volatility) (see Figure 6.5). As shown in Figure 6.5, this simple model identifies well-known high-volatility periods over the timeline, such as the 2008 financial crisis and the pandemic period in 2020. It is worth noting that in this example, we are addressing the problem of market regime detection or identification, not market regime forecasting or prediction.

Two Sigma, one of the largest hedge funds in the world, ranked fourth in Forbes Advisor's "Top 10 US Hedge Funds of December 2024" (Baldridge, 2024) with $67.47

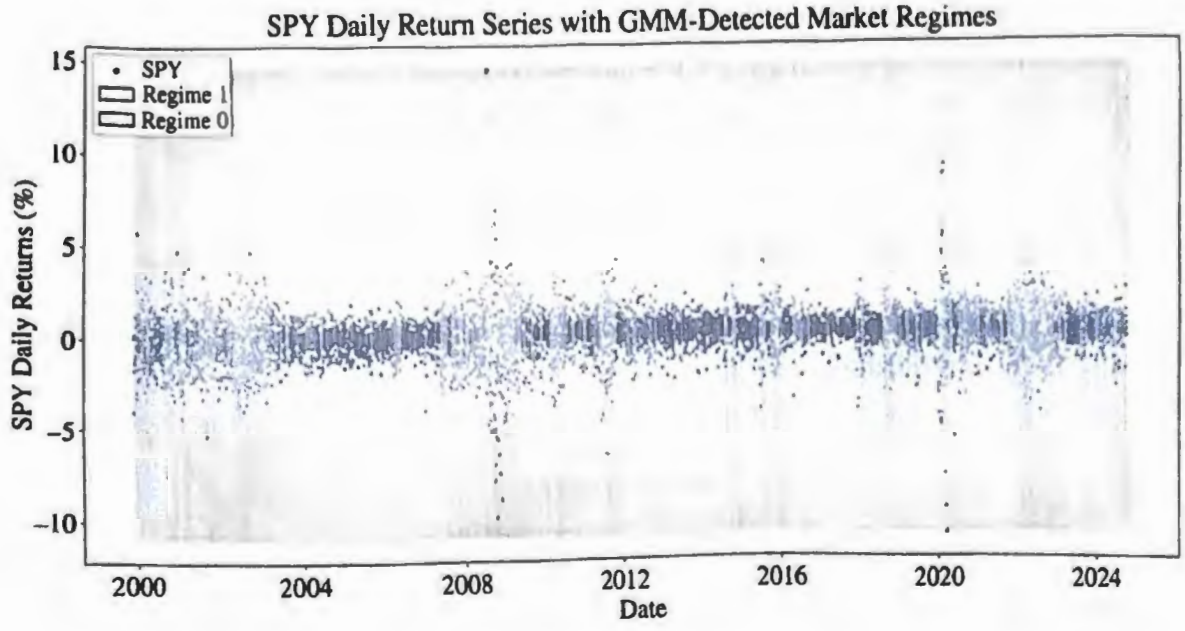

Figure 6.5 Gaussian Mixture Model for market regime detection.

billion in assets under management (AUM) at the time of writing this chapter, shared a similar approach in their blog post, "A Machine Learning Approach to Regime Modeling," see Botte and Bao (2021). They used GMM to detect market regimes and identified four clusters they linked to different market conditions: Crisis, Steady State, Inflation, and Walking on Ice. They backed these labels with historical analysis and applied the method to custom-built factors developed in-house.

Even though their setup uses more clusters and proprietary data, the underlying method is very similar. GMM is a strong starting point for building more powerful models, which you can tailor with your unique insights into the market.

There are many ways to improve our modeling approach for regime detection. For example, one limitation of the basic GMM setup is that it assumes the latent variables are independent over time. This assumption ignores the well-known stylized fact of volatility clustering, where periods of high volatility tend to be followed by more high-volatility periods, and the same is true for low-volatility periods.

To capture this, you can add dynamics to the latent variables. This essentially turns GMM into a Hidden Markov Model (HMM), which has been widely used for market regime detection (see Chan [2017] Kinlaw et al., [2021]). Unlike GMM, HMM adds "memory," allowing it to better reflect how market conditions evolve over time.

As a fun historical note, Leonard Baum (https://en.wikipedia.org/wiki/Leonard_E._Baum), one of the first employees of Monemetrics (the precursor to Renaissance Technologies, considered the most successful quantitative hedge fund ever), co-developed the Baum-Welch algorithm. This algorithm, a special case of Expectation-Maximization (EM), is used to estimate the parameters of an HMM.

According to Gregory Zuckerman's book (Zuckerman, 2019), *The Man Who Solved the Market,* James Simons, founder of Renaissance Technologies, said about his early days at Monemetrics: "Once I got Lenny (Leonard) involved, I could see the possibilities of building models."

Now, in the rest of this chapter, we will focus on cases where the parameters of the conditional distribution of observations given the latent variables are functions of the latent variables and are modeled using Deep Neural Networks.

6.3.4 Deep Latent Variable Models

In this book, we will focus on specific representations for the latent variable model we have discussed so far, namely:

$$\mathbf{z} \sim p(\mathbf{z})$$
$$\mathbf{x}|\mathbf{z} \sim p(\mathbf{x}|\mathbf{z})$$

Here, $p(\mathbf{z})$ represents the prior over the latent variable $\mathbf{z}$. Unlike the earlier examples, the parameters of the conditional probability of $\mathbf{x}$ given $\mathbf{z}$ are determined by Deep Neural Network function of $\mathbf{z}$. Typically, as before, the relationship is represented as $p(\mathbf{x}|\mathbf{z}) = \text{ExponentialFamily}(\mathbf{x}; \eta)$, but now η is defined as $\eta = \text{DNN}_\theta(\mathbf{z})$.

Note that $p(\mathbf{x} \mid \mathbf{z})$ is also known as the **decoder**, a term that you will find all over the deep learning and LLM literature. Such models are referred to as Deep Latent Variable Models (DLVM). When the prior over the latent variable is Gaussian, the model is known as a Deep Latent Gaussian Model (DLGM); see Murphy (2023).

Contrary to the examples previously discussed, the posterior computation $p(\mathbf{z} \mid \mathbf{x})$ is intractable. Likewise, the marginal over $\mathbf{x}$ is also intractable:

$$\begin{aligned} p(\mathbf{x}) &= \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\ &= \int p(\mathbf{z}) p(\mathbf{x} \mid \mathbf{z}) d\mathbf{z} \\ &= \int p(\mathbf{z}) \text{ExponentialFamily}(\mathbf{x}; \text{DNN}_\theta(\mathbf{z})) d\mathbf{z} \end{aligned}$$

We will need to approximate this quantity to compute the density of $\mathbf{x}$ and, in turn, use it to estimate the parameters of the model, In the next section, we will explore methods for approximating this integral to effectively train Deep Latent Variable Models.

6.4 Learning

6.4.1 *Training Objective*

As we discussed in the previous chapter on autoregressive (AR) models, once we define a family of probability distributions parametrized by θ, denoted by $\{p_\theta\}_\theta$, our goal is to establish an objective function that will allow us to select the best model from this family that most accurately approximates the data distribution, p_{data}.

Similar to AR models, we will use the log-likelihood to define our objective function:

$$\begin{aligned} \mathcal{L} &= \frac{1}{N} \sum_n \log p_\theta(\mathbf{x}_n) \\ &= \frac{1}{N} \sum_n \log \int p(\mathbf{x}_n \mid \mathbf{z}_n) p(\mathbf{z}_n) d\mathbf{z}_n \end{aligned}$$

however, as we have observed, this calculation is intractable for most interesting cases.

6.4.2 *The Variational Inference Approximation*

Given that the exact likelihood is intractable, we will derive an approximation for it and pick the best model by optimizing this quantity. The following derivations were inspired by the exposition style presented in Levine (2021). Let's focus on a *single* observation $\mathbf{x}$

$$\begin{aligned} \log p(\mathbf{x}) &= \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\ &= \log \int \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} q(\mathbf{z}) d\mathbf{z} \\ &= \log \mathbb{E}_{\mathbf{z} \sim q} \left[\frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} \right] \end{aligned} \tag{6.5}$$

where $\mathbb{E}_{\mathbf{z}\sim q}$ indicates expectation value over the distribution $q(\mathbf{z})$. In this derivation, we introduced an arbitrary distribution $q(\mathbf{z})$ by applying an identity—multiplying and dividing the integrand by this quantity. This technique allowed us to rewrite the integral as an expectation with respect to arbitrary distribution $q(\mathbf{z})$. This technique is known as importance sampling, and it provides a framework for approximating expectations with respect to distributions that are difficult to sample from but easy to evaluate. It does so by using a distribution that is easier to sample from, such as $q(\mathbf{z})$, which is a design choice (see Bishop [2006]). In the importance sampling literature, $q(\mathbf{z})$ is referred to as the proposal distribution.

Applying Jensen's inequality to Equation 6.5, $\log\mathbb{E}[x] \geq \mathbb{E}[\log x]$, we can push the log inside the expectation, obtaining:

$$\begin{aligned}
\log p(\mathbf{x}) &= \log \mathbb{E}_{\mathbf{z}\sim q}\left[\frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z})}\right] \\
&\geq \mathbb{E}_{\mathbf{z}\sim q}\log\left[\frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z})}\right] \\
&\geq \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{x},\mathbf{z}) - \log q(\mathbf{z})\right] \\
&\geq \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{x}\mid\mathbf{z}) + \log p(\mathbf{z}) - \log q(\mathbf{z})\right]
\end{aligned} \tag{6.6}$$

Here, the last expression in Equation 6.6 is known as the Evidence Lower Bound (ELBO, pronounced as "elbow"), defined as follows:

$$\mathcal{L}(q,p) = \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{x}\mid\mathbf{z}) + \log p(\mathbf{z}) - \log q(\mathbf{z})\right] \tag{6.7}$$

The ELBO can be written or interpreted in two ways (Levine, 2021). The first one uses the entropy formulation:

$$\begin{aligned}
\mathcal{L}(q,p) &= \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{x}\mid\mathbf{z}) + \log p(\mathbf{z})\right] - \mathbb{E}_{\mathbf{z}\sim q}\log q(\mathbf{z}) \\
&= \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{x}\mid\mathbf{z}) + \log p(\mathbf{z})\right] + \mathbb{H}[q]
\end{aligned} \tag{6.8}$$

here, $\mathbb{H}[q]$ represents the Entropy of the probability distribution q, defined $\mathbb{H}[q] = \mathbb{E}_{\mathbf{z}\sim q}\left[-\log q(\mathbf{z})\right]$.

The second commonly used interpretation involves the Kullback-Leibler Divergence:

$$\begin{aligned}
\mathcal{L}(q,p) &= \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{x}\mid\mathbf{z}) + \log p(\mathbf{z})\right] - \mathbb{E}_{\mathbf{z}\sim q}\log q(\mathbf{z}) \\
&= \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{x}\mid\mathbf{z})\right] - \mathbb{E}_{\mathbf{z}\sim q}\left[\log\frac{q(\mathbf{z})}{p(\mathbf{z})}\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{x}\mid\mathbf{z})\right] - \mathrm{KL}(q(\mathbf{z})\|p(\mathbf{z}))
\end{aligned} \tag{6.9}$$

This formulation highlights the common interpretation given to this objective function in the variational inference literature: the first term represents the reconstruction error (log likelihood), and the second term serves as a regularization term. The advantage of this approach is that the KL divergence between $q(\mathbf{z})$ and $p(\mathbf{z})$ often has

a closed-form solution, especially when both distributions are Gaussian—a common design choice that also simplifies the gradient computation.

6.4.2.1 How to Choose the Proposal Distribution One critical decision in this formulation is to pick the proposal distribution, $q(\mathbf{z})$. We could say that it makes sense to align the proposal distribution close to the posterior distribution of the latent variable $\mathbf{z}$ given the observed data $\mathbf{x}$, represented as $p(\mathbf{z} \mid \mathbf{x})$. Therefore, we can opt to minimize the KL divergence between both distributions to make $q(\mathbf{z})$ close to $p(\mathbf{z} \mid \mathbf{x})$, which involves:

$$\begin{aligned}
\mathrm{KL}(q(\mathbf{z}) \| p(\mathbf{z} \mid \mathbf{x})) &= \mathbb{E}_{\mathbf{z} \sim q}\left[\log \frac{q(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{x})}\right] \\
&= \mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z}) - \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{z} \mid \mathbf{x}) \\
&= \mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z}) - \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{x}, \mathbf{z}) + \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{x}) \\
&= \mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z}) - \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{z}) - \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{x} \mid \mathbf{z}) + \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{x}) \\
&= -\mathbb{E}_{\mathbf{z} \sim q}[\log p(\mathbf{z}) + \log p(\mathbf{x} \mid \mathbf{z}) - \log q(\mathbf{z})] + \log p(\mathbf{x}) \\
&\quad -\mathcal{L}(q, p) + \log p(\mathbf{x}) \qquad (6.10)
\end{aligned}$$

here, minimizing $\mathrm{KL}(q(\mathbf{z}) \| p(\mathbf{x} \mid \mathbf{z}))$ with respect to q is equivalent to maximizing $\mathcal{L}(q, p)$ with respect to q, since $\log p(\mathbf{x})$ is independent of $\mathbf{z}$:

$$\operatorname{argmin}_q \mathrm{KL}(q(\mathbf{z}) \| p(\mathbf{z} \mid \mathbf{x})) = \operatorname{argmax}_q \mathcal{L}(q, p) \qquad (6.11)$$

Hamlet remembers the first time he encountered this derivation and how amazed he was by its elegant result. It enables us to design a proposal distribution using tractable quantities, even when the quantities involved in the derivation, such as the posterior distribution, are intractable. In most practical cases, where deep neural networks are used to parameterize conditional distributions, computing the posterior $p(\mathbf{z} \mid \mathbf{x})$ directly is infeasible. If we had direct access to this posterior, we could use it instead of relying on $q(\mathbf{z})$.

Remarkably, the derivation shows that we do not actually need access to the posterior $p(\mathbf{z} \mid \mathbf{x})$ to find a proposal distribution $q(\mathbf{z})$ that is close to it, that minimizes the KL divergence between both distributions. By developing the optimization problem, we discover that making $q(\mathbf{z})$ as close as possible to the posterior is equivalent to maximizing Equation 6.7, the ELBO with respect to q. This is a powerful result because the ELBO consists of quantities that are tractable and can be computed efficiently.

What surprised Hamlet most is the realization that by optimizing the same objective—the ELBO—with respect to p, we can simultaneously find the best parameters for approximating the data distribution. In other words, finding the optimal q that best approximates the posterior and the optimal p that best models the data involves alternating between two steps:

1. Maximizing the ELBO with respect to q to make $q(\mathbf{z})$ as close as possible to the posterior $p(\mathbf{z} \mid \mathbf{x})$ in KL divergence sense.
2. Maximizing the ELBO with respect to p to increase the likelihood of the observed data $\mathbf{x}$.

This alternating optimization approach unifies the goals of posterior approximation and data likelihood maximization under the same objective function.

Recall that we have constructed all these equations for a single, fixed, observation $\mathbf{x}$. One of the challenges with this formulation is that for each $\mathbf{x}_n$ in our dataset $\mathcal{D}$, we need to solve an inference problem as specified by Equation 6.11 to determine q for the specific $\mathbf{x}_n$ denoted as q_n. For example, if we represent $q_n(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mu_n, \sigma_n)$, this will involves finding a unique μ_n and σ_n for every data point. Consequently, the number of parameters to be estimated are:

$$|\theta| + N(|\mu| + |\sigma|)$$

here, the notation $|\cdot|$ represent the number of parameters. As indicated by the previous formula, the number of parameters to estimate is linear with respect to the size of the dataset N, which in current practical applications can be huge.

6.4.2.2 Amortized Inference. The main idea behind amortized inference is that if we repeatedly face the same inference problem, could a parameterized function solve it for us? This leads us to consider using a family of neural networks for this task.

Instead of determining a distinct q by solving the optimization problem in Equation 6.11 for every individual $\mathbf{x}$, we might think of finding a single q_ϕ where the parameters are determined by a Deep Neural Network that takes $\mathbf{x}$ as input, and that minimizes the average KL divergence across the entire dataset:

$$\min \sum_n \mathrm{KL}\left(q_\phi(\mathbf{z}_n|\,\mathbf{x}_n) \| p_\theta(\mathbf{z}_n|\,\mathbf{x}_n)\right) = \max \sum_n L\left(q_\phi(\mathbf{z}_n|\,\mathbf{x}_n), p_\theta(\mathbf{x}_n|\,\mathbf{z}_n)\right) \quad (6.12)$$

This formulation enables us to use another Deep Neural Network to define the parameters of q_ϕ and solve the problem in an amortized manner. This network takes an input $\mathbf{x}$ and outputs the parameters of a probability distribution over the latent variable $\mathbf{z}$ is commonly referred to as the **encoder** in the literature.

6.4.3 Optimization

As previously seen, to optimize for ϕ and θ, we need to maximize the same training objective, the ELBO, with respect to both θ and ϕ. Now, considering the computation of:

$$\nabla_\phi \mathbb{E}_{\mathbf{z}\sim p_\phi}[p_\theta(\mathbf{x}|\,\mathbf{z})] \quad (6.13)$$

we face the challenge of computing gradients with respect to ϕ, which involves taking the derivative with respect to the parameters of the distribution over which the expected value is taken. There are two common methods to address this problem: the famous REINFORCE algorithm (Williams, 1992) and the reparametrization trick (Kingma, 2013).

6.4.3.1 The Likelihood Gradient, REINFORCE. The REINFORCE algorithm can be derived as follows:

$$\begin{aligned}\nabla_{\phi}\mathbb{E}_{\mathbf{z}\sim q_{\phi}}[p_{\theta}(\mathbf{x}\mid\mathbf{z})] &= \nabla_{\phi}\int q_{\phi}(\mathbf{z}\mid\mathbf{x})\,p_{\theta}(\mathbf{x}\mid\mathbf{z})\,d\mathbf{z}\\ &= \int\nabla_{\phi}q_{\phi}(\mathbf{z}\mid\mathbf{x})\,p_{\theta}(\mathbf{x}\mid\mathbf{z})\,d\mathbf{z}\\ &= \int q_{\phi}(\mathbf{z}\mid\mathbf{x})\,\nabla_{\phi}\log q_{\phi}(\mathbf{z}\mid\mathbf{x})\,p_{\theta}(\mathbf{x}\mid\mathbf{z})\,d\mathbf{z}\\ &= \mathbb{E}_{\mathbf{z}\sim p_{\phi}}\left[\nabla_{\phi}\log q_{\phi}(\mathbf{z}\mid\mathbf{x})\,p_{\theta}(\mathbf{x}\mid\mathbf{z})\right]\end{aligned} \tag{6.14}$$

This method computes the gradient of an expectation by exploiting the identity $\nabla_x \log f(x) = \frac{\nabla_x f(x)}{f(x)}$. So, the gradient of an expectation becomes the expectation of a gradient. REINFORCE uses a Monte Carlo approach to estimate the gradient of the expectation in Equation 6.14. One advantage of this method is that it works with latent variables whose value spaces can be either discrete or continuous. However, one of the primary disadvantages with this approach is that it tends to suffer from high variance. For more details about the algorithm, see Williams (1992).

6.4.3.2 Reparameterization Trick. An alternative approach is the reparameterization trick (Kingma, 2013), which can be utilized for the computation of Equation 6.13.

For expectations involving sampling $\mathbf{z}$ from $q_{\phi}(\mathbf{z}\mid\mathbf{x}) = \mathcal{N}\left(\mathbf{z};\boldsymbol{\mu}_{\phi}(x),\boldsymbol{\Sigma}_{\phi}(x)\right)$, we can reparametrize $\mathbf{z}$ as:

$$\mathbf{z} = \boldsymbol{\mu}_{\phi}(x) + \boldsymbol{\Sigma}_{\phi}(x)^{1/2}\boldsymbol{\epsilon} \tag{6.15}$$

where $\boldsymbol{\epsilon}\sim\mathcal{N}(0,\mathbf{I})$. With the reparametrization trick, all randomness is absorbed by $\boldsymbol{\epsilon}$, and for computing the gradient with respect to $\boldsymbol{\phi}$, we just need to compute the gradient of the deterministic functions $\boldsymbol{\mu}_{\phi}$ and $\boldsymbol{\Sigma}_{\phi}$.

$$\nabla_{\phi}\mathbb{E}_{\mathbf{z}\sim q_{\phi}}[p_{\theta}(\mathbf{x}\mid\mathbf{z})] = \nabla_{\phi}\mathbb{E}_{\boldsymbol{\epsilon}\sim\mathcal{N}(0,\mathbf{I})}\left[p_{\theta}\left(\mathbf{x}\mid\boldsymbol{\mu}_{\phi}(x)+\boldsymbol{\Sigma}_{\phi}(x)^{1/2}\boldsymbol{\epsilon}\right)\right]$$

This gradient can be computed efficiently using automatic differentiation tools such as PyTorch or TensorFlow.

6.4.4 Mind the Gap!

Another way to write Equation 6.10 is:

$$\log p(\mathbf{x}) = \mathcal{L}(q,p) + \mathrm{KL}(q(\mathbf{z})\|p(\mathbf{x}\mid\mathbf{z})) \tag{6.16}$$

In this formulation, the term $\mathrm{KL}(q(\mathbf{z})\|p(\mathbf{x}\mid\mathbf{z}))$ is known as the gap or error that arises when approximating the log likelihood by the ELBO. If we truly match $q(\mathbf{z})$ to $p(\mathbf{x}\mid\mathbf{z})$, the KL will be zero and the log likelihood is equal to the ELBO. Consequently, the error in our approximation is quantified by the KL divergence between our approximation q_{ϕ} to $p(\mathbf{z}\mid\mathbf{x})$.

6.5 Variational Autoencoder (VAE)

The Variational Autoencoder (VAE) is a latent variable model where the marginal distribution over the latent variable is Gaussian, and both the encoder and decoder (to be discussed shortly) are modeled as conditional Gaussians, as described by Kingma (2013). The observations can be images, multivariate time series, etc.

The encoder takes as input $\mathbf{x}$ and outputs the parameters of $q_{\phi}(\mathbf{z}|\mathbf{x})$, specifically the mean vector and covariance matrix. Similarly, the decoder takes $\mathbf{z}$ as input and outputs the mean vector and covariance matrix for $p_{\theta}(\mathbf{x}|\mathbf{z})$. The classical VAE is typically trained using the reparametrization trick. A schematic representation of these operations is illustrated in Figure 6.6.

More formally, the classical VAE (Kingma, 2013) is defined by the following representations:

- The prior over $\mathbf{z}$, $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$
- The encoder, $q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\boldsymbol{\mu}_{\phi}(\mathbf{x}), \boldsymbol{\Sigma}_{\phi}(\mathbf{x})\right)$
- The decoder $p_{\theta}(\mathbf{x}|\mathbf{z}) = \mathcal{N}\left(\boldsymbol{\mu}_{\theta}(\mathbf{z}), \boldsymbol{\Sigma}_{\theta}(\mathbf{z})\right)$

VAEs are typically trained using the reparametrization trick. During training, the ELBO is used as training objective:

$$\mathcal{L}(q,p) = \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\mathbf{I})}[\log p_{\theta}(\mathbf{x}|\mathbf{z})] - \mathrm{KL}(q_{\phi}(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))$$

with $\mathbf{z} = \boldsymbol{\mu}_{\phi}(x) + \boldsymbol{\Sigma}_{\phi}(x)^{1/2}\boldsymbol{\epsilon}$. To optimize for $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$, gradients $\nabla_{\theta} L(q,p)$ and $\nabla_{\phi} L(q,p)$ are computed, typically using Stochastic Gradients. Parameters $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ can be then updated using gradient ascent.

After training the VAE, new points can be generated by sampling a value from the prior and feeding this value into the decoder to generate a new sample:

$$\begin{aligned} \mathbf{z} &\sim p(\mathbf{z}) \\ \mathbf{x}|\mathbf{z} &\sim p_{\theta}(\mathbf{x}|\mathbf{z}) \end{aligned}$$

In Figure 6.7, we see visualizations from the paper Kingma (2013) of the learned latent space and the corresponding data space, generated by a VAE after training on the Frey Faces dataset.

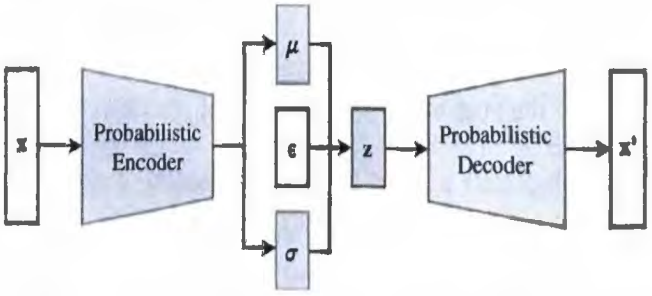

Figure 6.6 VAE. *Source:* EugenioTL / https://en.m.wikipedia.org/wiki/File:Reparameterized_Variational_Autoencoder.png, last accessed on 13 January 2025 / CC BY 30.

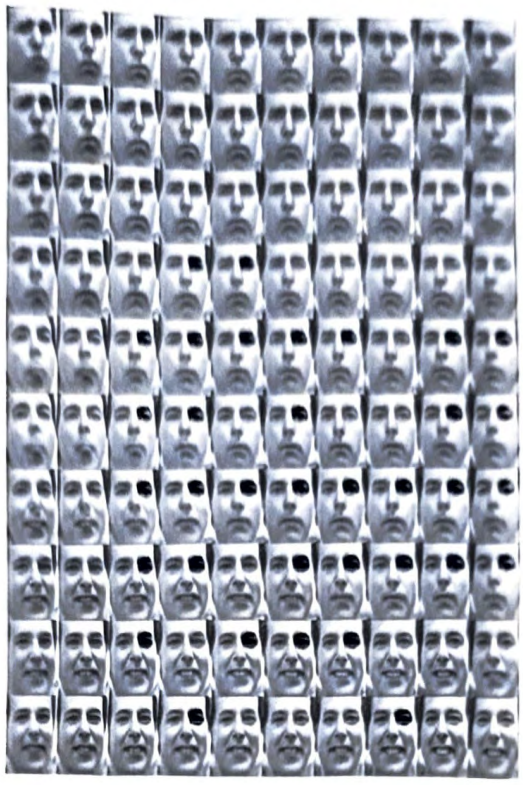

Figure 6.7 Illustration of the learned data manifold for generative models with a two-dimensional independent Gaussian latent space. Varying one coordinate generates smiling faces, while varying the other alters head poses, both independently. The model was trained on the Frey Faces dataset. *Source:* Kingma (2013).

A remarkable aspect is that since the prior of the latent space are independent Gaussian variables (identity covariance), using a two-dimensional latent space illustrates disentanglement in the data space. Varying one coordinate can generate smiling faces, while varying the other can alter head poses, all independently, as shown in Figure 6.7.

The VAE can also serve as a representation learning tool for downstream machine learning tasks. For example, if $\mathbf{x}$ represents an image, the compressed representation $\mathbf{z}$ could be used as input for an image classifier. In fact, this representation learning approach is utilized in Stable Diffusion, one of the most powerful techniques for image generation currently available.

In the accompanying notebook for this chapter ([*VAE Notebook*]), we implement and test the traditional VAE model introduced by Kingma (2013). While this example uses image data, it serves as an important step for building a solid understanding of VAEs, which we will extend to capture time-series dynamics.

6.6 VAEs for Sequential Data and Time Series

As we saw in the previous section based on the work of Kingma (2013), and illustrated in Figure 6.7, VAEs are really good at modeling the correlations within an image $\mathbf{x}_n$ in a dataset, where $\mathbf{x}_n$ is a D-dimensional vector of pixel intensities. If we want to apply standard VAEs to time-series data, we might simply reinterpret the vector $\mathbf{x}_n$ as a

multivariate time series with D series at time n and use VAEs to sample cross-sectional time series, for example.

However, in the standard latent variable model framework, the data generation process assumes that we sample $\mathbf{z}_n$ from $p(\mathbf{z})$ and then generate $\mathbf{x}_n$ from the conditional distribution $p(\mathbf{x} \mid \mathbf{z}_n)$. By design, this structure assumes independence between consecutive observations. This independence assumption can significantly limit the performance of VAEs in scenarios where temporal dependencies between observations are significant.

There are several interesting ways to extend the standard VAE to capture temporal dependencies between observations. This is particularly important for modeling time series at certain time scales, where past prices/returns, market conditions, and other factors significantly influence future prices/returns. These extensions can take inspiration from traditional time-series models, such as incorporating autoregressive structures in the observations or latent variables, similar to classical AutoRegressive models or Dynamic Linear Models (DLMs), see Prado and West (2010). Next, we mention a few alternatives for this.

6.6.1 Extending VAEs for Time Series

6.6.1.1 Sequential Encoders and Decoders. One straightforward extension is to replace the standard VAE encoder and decoder used in Kingma (2013) with sequential models, such as RNNs, or Transformers. One example of this approach is the Variational Recurrent Autoencoder (VRAE) by Fabius and Van Amersfoort (2014). Following is a brief description of the main idea behind VRAE:

- VRAE uses the final hidden state of the encoder RNN as a representation or summary of the input sequence. This hidden state is then used to determine the parameters of the probability distribution of the latent variable $\mathbf{z}$. In other words, the distribution over $\mathbf{z}$ is determined from the last state of the RNN.
- The latent variable $\mathbf{z}$ is then sampled using the reparameterization trick.
- Finally, the sampled latent variable $\mathbf{z}$ is passed into the decoder RNN, which reconstructs the input sequence.

6.6.1.2 Superposition of Time-series Components. Another approach is to model time series as a superposition of components—an idea commonly used in dynamic linear models (DLMs), see Prado and West (2010). In this framework, the observed time series might be composed as a combinations of the following:

- Local level or intercept (e.g., a random walk)
- Long-term trend components
- Seasonal patterns (e.g., weekly, monthly, or yearly periodicities)
- Short-term autocorrelated processes

Here, each time-series component can have its own state vector (latent variables), which is then projected linearly into the data space.

This approach provides both the interpretability and the flexibility to incorporate domain expertise, address data limitations, and adapt to the specific needs of the problem at hand. By using individual time-series components, modelers can design solutions that are tailored to their specific needs. In the next section, we will describe a solution based on this principle.

If you want to learn more about DLM models, check out the introduction to *Bayesian Structural Time-series Models* (Medina Ruiz, 2019), which explores time-series modeling for predicting Criteo's internet traffic load in the context of server infrastructure capacity planning. This example highlights how injecting domain knowledge into the modeling process can address data limitations and improve the overall performance.

6.6.1.3 TimeVAE: A Flexible VAE for Time-series Generation. A more recent example of extending VAEs to time series is TimeVAE, introduced by Desai et al. (2021). TimeVAE adapts VAEs specifically for multivariate time-series generation, combining traditional deep learning layers with time-series-specific components.

By using specialized decoders, TimeVAE allows the inclusion of temporal structures—such as trends and seasonal components—directly into the modeling process. This approach combines the flexibility and interpretability of traditional methods like Dynamic Linear Models (DLMs), but with the advantage of capturing nonlinear relationships in high-dimensional data. TimeVAE makes use of both traditional learning layers (e.g., dense and convolutional layers) and custom time-series-specific layers to capture components like levels, multi-polynomial trends, and seasonalities.

The TimeVAE architecture is divided into two main parts: **Base TimeVAE**, which does not incorporate domain knowledge for modeling the time series, and **Interpretable TimeVAE**, which allows domain knowledge to be injected and is tailored to model specific time-series components. Following is a description of the architecture.

6.6.1.3.1 Architecture of TimeVAE

1. Base TimeVAE encoder and decoder
   - The encoder processes the input time series through a series of convolutional layers with ReLU activation.
   - The output of the encoder is flattened and passed through a fully connected (dense) linear layer to determine the parameters of the distribution over the latent variable $\mathbf{z}$, which is modeled as a multivariate Gaussian.
   - A realization of the latent variable $\mathbf{z}$ is sampled using the reparametrization trick, which is then feed into the decoder.
   - The decoder tries to invert the encoding process. It consists of a fully connected linear layer, followed by reshaping and a series of transposed convolutional layers with ReLU activation. Finally, the data passes through a time-distributed fully connected layer to produce the reconstructed time series.

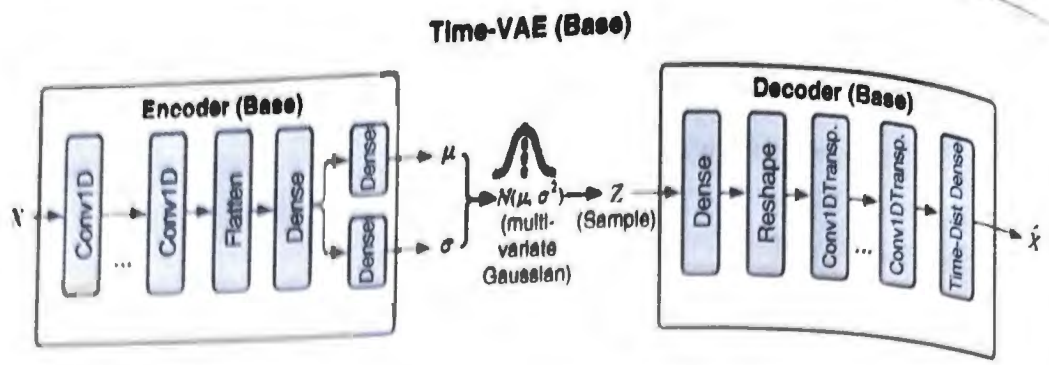

Figure 6.8 Illustration of the encoder-decoder architecture of Base TimeVAE. The input X is processed through a series of operations, including convolutional layers, reshaping, and a dense layer, to determine the parameters of the latent variable distribution Z. The decoder takes a realization of Z and tries to reconstruct the input by inverting the encoding process. Figure 1 from Desai et al. (2021).

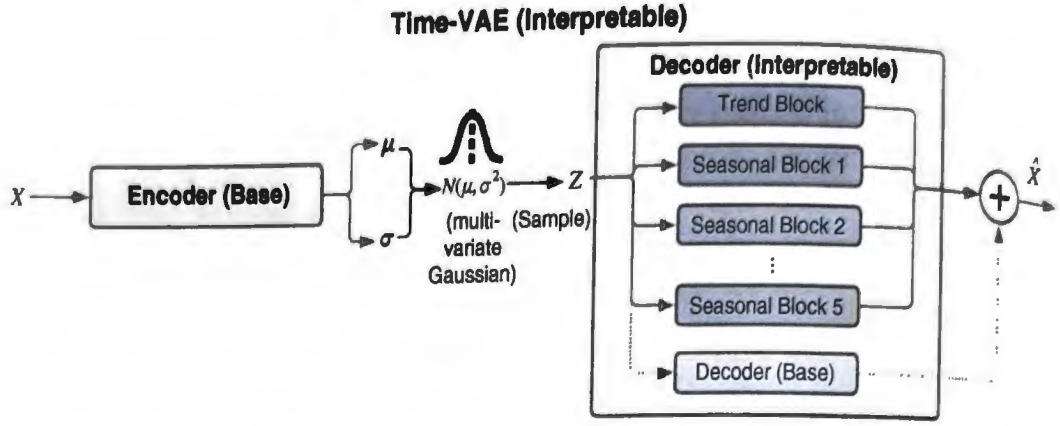

Figure 6.9 Illustration of the main components in Interpretable TimeVAE. Specialized decoders, such as the **Trend Block**, **Seasonal Block 1**, and **Seasonal Block 2**, take the latent variable Z as input. The outputs of these decoders are combined in parallel through addition to produce the final reconstruction of the time series. Seasonal Block 1 and Seasonal Block 2 can represent different seasonal components, capturing multiple seasonalities present in the time series. Figure 2 from Desai et al. (2021).

A schematic representation of Base TimeVAE can be found in Figure 6.8.

2. Interpretable TimeVAE: Time-series-specific components

As mentioned, TimeVAE introduces specialized decoders to model individual components of a time series, such as trends and seasonalities.

Each specialized decoder takes the latent variable **z** as input (which can be produced by the Base TimeVAE encoder). The outputs of these decoders are combined in parallel by addition to produce the final reconstruction of the time series. A schematic representation of Interpretable TimeVAE can be found in Figure 6.9.

TimeVAE provides the following advantages:

- Flexibility: You can easily model multiple temporal components, such as polynomial trends of different degrees, and different seasonalities (e.g., weekly, monthly, yearly), making it highly adaptable to capture different time-series characteristics.

- Interpretability: By explicitly modeling components like trends and seasonality, TimeVAE makes it easier to understand the underlying structure of the time series.
- Nonlinear Modeling: unlike traditional models like DLM, TimeVAE can capture complex nonlinear relationships in high dimensional data.

In the accompanying notebook for this chapter ([*TimeVAE Notebook*]), we implement and test TimeVAE on financial data. We believe that the best way to learn is by hands-on practice, and this notebook gives you the opportunity to explore and experiment with the specific decoder architectures used in the paper for modeling trends and seasonalities. You can also extend these architectures to fit your specific needs.

If you are not yet familiar with the standard VAE implementation, we recommend starting with the VAE notebook, where the encoder and decoder architectures are easier to follow. This will provide a solid foundation before diving into TimeVAE.

6.7 Conclusion

In this chapter, we explored how VAEs provide a powerful framework for modeling complex, multimodal, and high-dimensional data.

We saw how VAEs are highly efficient at sampling and flexible for modeling high-dimensional data using a smaller latent space representation. This latent space provides meaningful data representations that can be used for applications such as compression, data generation, and other downstream tasks. However, one trade-off for this flexibility is that the likelihood computation becomes intractable for most practical cases involving deep models. To address this, we rely on likelihood approximations, which enable parameter estimation and tasks like model selection.

We also discussed how VAEs can be extended to handle sequential data, particularly for time-series dynamics. For example, we introduced approaches like the VRAE and TimeVAE, which combine the VAE framework with sequential architectures like RNNs and time-series-specific components. These models offer a practical starting point for building solutions tailored to your own applications, whether for forecasting financial data, generating financial data, or other sequential problems or data. The latent representation learned by VAEs is highly versatile and can be used to train downstream models to solve a wide variety of applications. While we have seen its use in time-series generation, it can also be used for tasks such as classification, anomaly detection, and forecasting.

In the next chapters, we will continue expanding our toolbox of deep generative models for high-dimensional data—models that were originally successful in NLP and computer vision. More importantly, for our use cases in Trading and Asset Management, we will explore techniques to show how these models can be further extended and adapted for time-series and sequential data applications.