Chapter 5

Deep Autoregressive Models for Sequence Modeling

Autoregressive (AR) models are a familiar presence in the quantitative investor's toolbox. After all, the most basic financial data is a price series (sequence), and who doesn't want to predict the next price? In Chan (2017), Ernie discussed the various linear predictive AR models such as AR(p), ARMA(p,q), VAR, VEC, and state space models such as the Kalman filter. In this chapter, we will find out whether we can push the envelope by adopting nonlinear techniques from deep learning for such a "sequence modeling" problem. Beyond predictions, AR can be used for anomaly detection and data generation applications as well. It falls under the category of explicit density models, as illustrated in Figure 5.1. This means it proposes a tractable probability density model to approximate the data distribution p_{data}.

Sequential data means that data appear in sequences where a natural or artificial order is present. This principle enables the construction of probabilistic models for any variable in the sequence by conditioning on its preceding values.

This feature makes them useful for applications involving inherently ordered data, such as audio signals, meteorological phenomena (e.g., precipitation patterns), text data for Natural Language Processing (NLP) applications, or human-created time series like the ones found in financial markets. Remarkably, AR models also apply to datasets lacking of inherent order, such as images. By imposing an order among the pixels within the image, AR models can generate complete images one pixel at a time, as demonstrated with the PixelCNN architecture, see Oord et al. (2016b) for more details.

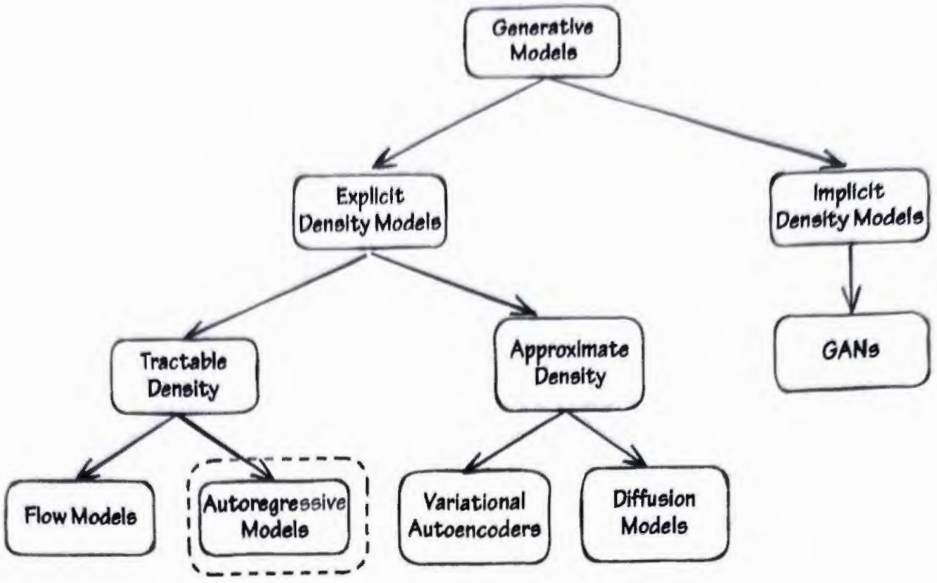

Figure 5.1 Model taxonomy: Autoregressive models.

5.1 Representation Complexity

Consider a sequence of random variables $x_1, x_2, ..., x_N$. A probabilistic model of the sequence is specified by its joint probability distribution, $p_{X_1, X_2, ..., X_N}(x_1, x_2, ..., x_N)$. Through the application of the product rule in probability, we can decompose the joint distribution into a product of conditionals:

$$\begin{aligned} p_{\mathbf{X}}(\mathbf{x}) = p_{X_1, X_2, ..., X_N}(x_1, ..., x_N) = & \; p_{X_1}(x_1) \\ & \times p_{X_2|X_1}(x_2|x_1) \\ & \times p_{X_3|X_1, X_2}(x_3|x_1, x_2) \\ & \cdots \\ & \times p_{X_N|X_1, ..., X_{N-1}}(x_N|\mathbf{x}_{1:N-1}) \end{aligned} \tag{5.1}$$

AR models simplify the modeling of high-dimensional distributions by exploiting the right-hand side of Equation 5.1, modeling a series of conditional distributions.

These distributions provide probabilistic descriptions for each variable based on its predecessors in the sequence, transforming complex high-dimensional modeling into a series of more manageable conditional probabilities that are easier to represent and estimate.

Despite AR models' simplification of high-dimensional problem into more manageable one-dimensional conditional distributions, the computational cost of inference and storage remains a challenge. For example, in financial time series like the S&P500, discretizing values into a finite states, let's say, five states—a common practice in financial applications, such as the NUMERAI Hedge Fund competition—illustrates the complexity. As an example, consider modeling the conditional distribution $p_{X_{n+1}|\mathbf{X}_{1:n}}(x_{n+1}|\mathbf{x}_{1:n})$ given the previous $n = 21$ values in the sequence. Think of this

example as forecasting the next day's returns of a security based on the last month of daily returns, similar to trading applications where the past month's data is used for prediction. This approach would require a huge probability table with 5^{21} entries, which is clearly impractical. Instead of specifying conditional probabilities through table lookups, a parameterized model is necessary. However, even with a parameterized approach, modeling interactions between all variables can result in a prohibitive number of parameters. Therefore, we need to find ways to reduce the number of parameters.

5.2 Representation and Complexity Reduction

One way to reduce the number of parameters in the model is by making assumptions about the structure of the joint distribution in Equation 5.1. These assumptions might include specifications about variable interactions, density forms, etc.

One approach to reduce complexity is to impose conditional probability assumptions, such that to model x_n we do not need the knowledge of the entire sequence of previous values $x_1,...,x_{n-1}$. For example, we could say that to model x_n, all we need to know is the value of x_{n-1}, suggesting that all relevant information is contained in this variable. (There is no loss of generality in assuming that x_n only depends on the previous state x_{n-1}, because we could just redefine a state to include information from a longer lookback period.) Mathematically, we express this as $p_{X_n|X_1,...,X_{n-1}}(x_n|x_1,...,x_{n-1}) = p_{X_n|X_{n-1}}(x_n|x_{n-1})$, also known as the Markov Property. We touched on this in Chapter 3 when we discussed the Hidden Markov Model.

Another technique involves weight sharing, where, in the case of the previous example, the conditional probability $p_{X_n|X_{n-1}}(x_n|x_{n-1})$ is independent of the sequence position n, imposing a kind of invariance or stationarity across the sequence.

These simplifications also reduce the expressiveness of the model by narrowing the types of probability distributions that can be represented.

To illustrate the concept of conditional independence and model stationarity in sequence modeling, let's consider a practical example involving an AR model or order 2, denoted as AR(2). An AR(2) model factorizes the joint probability of a sequence $x_1,...,x_N$ as follows:

$$p(x_1,...,x_N) = p(x_1)p(x_2|x_1)p(x_3|x_1,x_2)p(x_4|x_2,x_3)...p(x_N|x_{N-2},x_{N-1}) \tag{5.2}$$

In this scenario, we assume that x takes values in the space $\chi = \mathbb{R}$ and that we can further model each conditional probability using a Gaussian distribution. Specifically, for any given x_n, the conditional probability $p(x_n|x_{n-2},x_{n-1})$ is expressed as:

$$p(x_n|x_{n-2},x_{n-1}) = \mathcal{N}(w_0 + w_1 x_{n-1} + w_2 x_{n-2}, \sigma^2) \tag{5.3}$$

here $\mathcal{N}(\cdot, \sigma^2)$ denotes a Gaussian distribution with the mean determined by the linear combination of the two preceding values in the sequence and a constant term, and with constant variance σ^2. As an example of using a linear AR models, the accompanying

notebook [*AR Notebook*] demonstrates forecasting the quarterly growth rate of the US gross national product (GNP).

This Gaussian assumption is for illustration only. We do not need to constrain ourselves to any particular form of the density function $p_\theta(\cdot\,|\cdot)$. We could even choose to model variables taking values in either continuous or discrete spaces. The choice of the density function may determine whether you frame the task as a classification problem or a regression problem, each with its own distinct characteristics. These representations will have their advantages and disadvantages in terms of efficient training and model representation, expressiveness and generalization, sampling quality and speed, and compression rate.

In this chapter, we will explore various model classes that are commonly used to model conditional probabilities, starting from traditional to more cutting-edge architectures. We will start with model families such as Logistic Regression to lay the groundwork and clarify essential concepts. Next we explore Neural Networks and Recurrent Neural Network leading to more complex families. We then advance to innovations that serve as the cornerstone of modern architectures, including Causal Masked Neural Networks, specifically WaveNet, and transformers. (The discussion on Causal Masked Neural Networks is inspired by insights gained from the lecture slides of Abbeel et al. [2020]).

To illustrate main concepts and techniques for image applications, we will use the standard Binary MNIST dataset (Larochelle and Murray, 2011), a canonical example in Computer Vision. Even though we use simple datasets, these models have the potential to extend far beyond these basic applications.

These models can be used to develop trading strategies utilizing alternative data. For instance, analyzing satellite images of crop fields can help predict commodity prices (Guida, 2019; Denev and Amen, 2020), while observations of retail parking lots can forecast retail company earnings (Guida, 2019), among other applications. Of course, we can also apply these models to price, or more appropriately, returns series.

By the end of the chapter, we will provide examples of how these models can be applied for asset management and trading, while also exploring how Transformer models can be adapted for time-series forecasting tasks. In Chapter 9, "Leveraging Large Language Models (LLMs) for Text Data Analysis in Trading," we will demonstrate how to construct a simple high-frequency trading strategy using Auto Regressive Generative Models to process speech signals from Federal Reserve announcements to inform trading decisions.

5.3 A Short Tour of Key Model Families

5.3.1 Logistic Regression Model

The binary MNIST dataset (Larochelle and Murray, 2011) is a simplified version of the original MNIST dataset, which has been widely used as a benchmark in computer vision.

Figure 5.2 Sample from the Binarized MNIST dataset. Larochelle and Murray (2011).

This dataset consists of 28×28 pixel images, with each pixel falling into $\{0,1\}$, representing the color black or white at that pixel location. The binary constraints means that there are $D = 2^{764}$ possible images, highlighting the fact that any practical training set can cover only a very small portion of all potential image configurations. Indeed, the dataset contains only 60,000 images for training and 10,000 images for testing.

In this context, an image $\mathbf{x}$ is an element of the space $\mathcal{X} = \{0,1\}^D$.

As typical examples of digits in the dataset, see Figure 5.2.

For a given image $\mathbf{x}$, we can write its joint probability as follows:

$$\begin{aligned} p_{\mathbf{X}}(\mathbf{x}) = p_{X_1, X_2, \ldots, X_{768}}(x_1, \ldots, x_{768}) &= p_{X_1}(x_1) \\ &\times p_{X_2|X_1}(x_2|x_1) \\ &\times p_{X_3|X_1,X_2}(x_3|x_1,x_2) \\ &\ldots \\ &\times p_{X_{768}|X_1,\ldots,X_{767}}(x_{768}|\mathbf{x}_{1:767}) \end{aligned} \tag{5.4}$$

This factorization is presented verbosely to highlight two key aspects of this model:

- Lack of parameters sharing
- No assumptions of conditional independence

In this model, each conditional probability is modeled using a logistic model defined as:

$$p(x_d|\mathbf{x}_{1:d-1}) = f(\mathbf{x}_{1:d-1}; \theta^{(d)}) = \text{sigmoid}(\mathbf{x}_{1:d-1}^T \mathbf{w}^{(d)} + b^{(d)}) \tag{5.5}$$

(Recall from Chapter 3 that the logistic model is defined in terms of the sigmoid function.) Here, a unique set of parameters $\theta^d = \{\mathbf{w}^d, b^d\}$ is assigned to each conditional. This model is known as a *fully visible sigmoid belief network* (FVSBN); see Frey et al. (1995).

So far, we have discussed several model representations and will introduce a few more. Keep in mind, however, that model training and parameter estimation will be covered in detail in Section 5.4 of this chapter.

5.3.1.1 Sampling from FVSN. The sampling process involves generating one pixel at a time, sequentially. For example, the first pixel is sample from its distribution, $x_1 \sim p_{X_1}$. Subsequent pixels are generated conditioned on the pixel values realized previously, e.g., $x_2 \sim p_{X_2|X_1}(\cdot, x_1)$, continuing in this manner for the rest of the pixels. This method is known as Ancestral Sampling (Bishop, 2006).

Instead of modeling the conditionals using a logistic model, we could explore using a different function to enhance model expressiveness, such as a Neural Network. Here,

$$p(x_d|\mathbf{x}_{1:d-1}) = f\left(\mathbf{x}_{1:d-1}; \theta^{(d)}\right) \tag{5.6}$$

where now $f = \text{NeuralNetwork}\left(\mathbf{x}_{1:d-1}; \theta^{(d)}\right)$. Of course, modeling with a Neural Network will increase the number of parameters to estimate.

5.3.2 Masked Autoencoder for Density Estimation (MADE)

The Masked Autoencoder for Density Estimation (MADE), see Germain et al. (2015), represents a significant advancement in the field of Generative AI by enhancing the capabilities of traditional autoencoders. Traditional autoencoders compress input data into a latent representation and then attempt to reconstruct it, introducing and denoising noise in the process. However, they are limited in their ability to generate new data points or to model the probability distribution of the data.

MADE addresses these limitations by introducing a sequential dependency among the input variables, following the chain rule of probability. This method ensures that the generation of any data point in the sequence does not depend on future data points, adhering to a predefined ordering. This innovation allows MADE to output probabilities and sample new data points in a way that respects the underlying data distribution, marking a departure from traditional autoencoders.

Key features of MADE include its flexibility in handling input variables of varying dimensionality and its use of a softmax output for each variable, representing a probability distribution from which new data points can be sampled. The model's structure, which incorporates masked connections between nodes, enables it to maintain the autoregressive property essential for accurate density estimation.

Illustrating the application of MADE, experiments on datasets like MNIST demonstrate its capability to generate digit images that increasingly resemble authentic data points with extended training. Figure 5.3 shows an image generated in the left and its nearest neighbor in the right, which is a very good practice to verify the model is not just learning the images of the dataset, and that it is actually generating new ones.

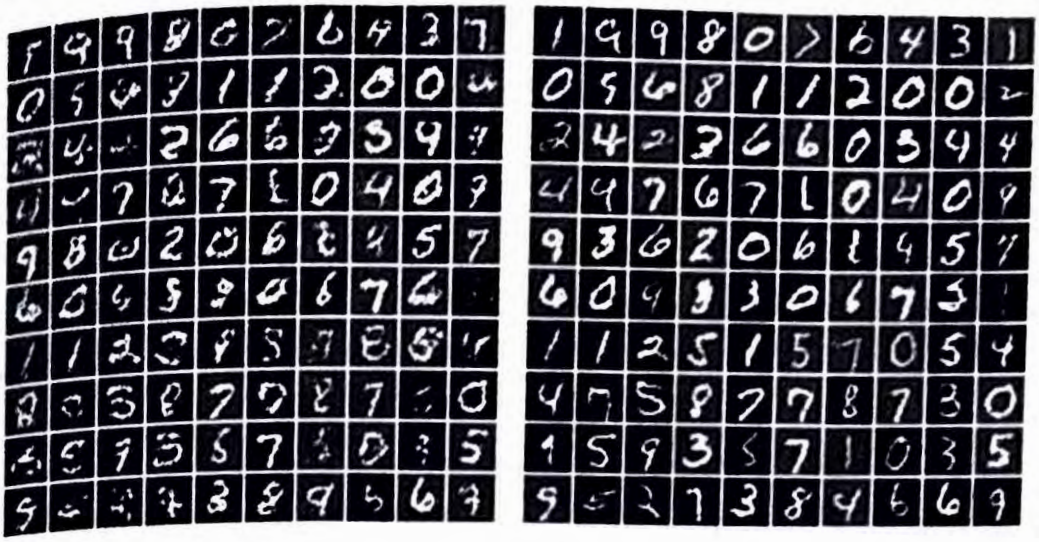

Figure 5.3 MADE Generation on MNIST. Left: samples from a MADE model. Right: Nearest neighbor in binarized MNIST. *Source:* Germain et al. (2015).

In conclusion, MADE's innovative approach to combining autoencoder frameworks with probabilistic modeling techniques has established it as a foundational model in Generative AI. Its contribution lies in its ability to generate new data points and model data distributions more accurately, paving the way for further advancements in the field.

5.3.3 Causal Masked Neural Network Models

Causal Masked Neural Networks have emerged as a leading approach in the field of Generative AI, building on the foundational concepts introduced by models like MADE. This approach has advanced the state of generative modeling by implementing efficient parameter sharing and introducing coordinate coding to incorporate location information about data sequences.

At the heart of Causal Masked Neural Networks is the principle of parameterizing conditionals with a neural network, similar to MADE, but with a crucial innovation: the use of the same neural network parameters across different stages of the generation process. This approach is akin to applying a "sliding window" over the data, allowing the repetitive use of a set of parameters, significantly increasing model efficiency by reducing the total number of parameters needed while still modeling high-dimensional data.

To address the challenges associated with parameter sharing, particularly the potential loss of positional information in data such as images, these networks incorporate coordinate coding. This technique involves feeding the location coordinates of data points (e.g., pixels in an image) into the model along with the data itself, enabling the network to retain spatial awareness despite the uniform use of parameters across different positions. Coordinate coding can be implemented in various forms, such as one-hot encodings or relative position encodings, adding a layer of flexibility to the model's design.

5.3.3.1 WaveNet. One of the first and most successful implementations of this approach was WaveNet (see Oord et al., 2016a), a model primarily designed for generating speech. Its influence extends beyond academic circles, with variations of the model being implemented in real-world systems such as personal assistants on smartphones. WaveNet is recognized for its superior audio generation quality, making it a cornerstone in the evolution of audio synthesis technologies.

WaveNet utilizes 1D convolutional layers and introduces additional techniques like dilation in convolutions to capture information from further back in the sequence, see Figure 5.4. This means WaveNet can capture longer-range dependencies within the audio data more efficiently, improving the quality and naturalness of the generated audio. This model has showcased the potential of this approach not only for audio data but also for a wide range of sequential data types.

In practice, WaveNet utilizes a stack of dilated causal convolutional layers to enable networks to have very large receptive fields, thereby allowing the network to consider inputs located far away in the sequence. For example, Figure 5.5 illustrates dilated causal convolutions with dilations of 1, 2, 4, and 8, as shown in the paper by Oord et al. (2016a).

As an example of using WaveNet, the accompanying notebook [*WaveNet Notebook*] demonstrates its application for forecasting financial data, providing an opportunity for hands-on learning with this type of models

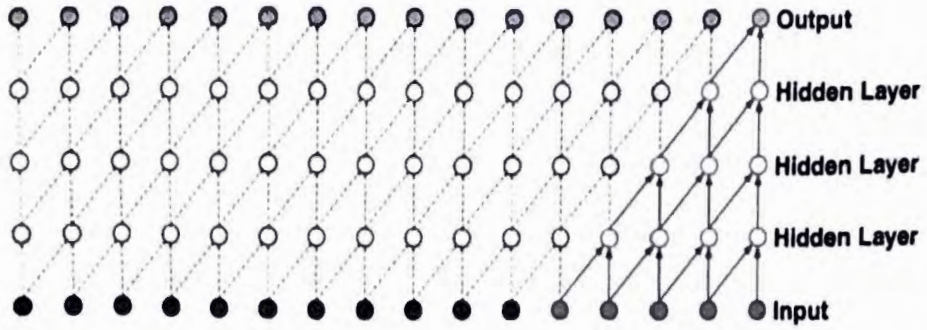

Figure 5.4 Visualization of a stack of causal convolutional layers. Figure 2 from Oord et al. (2016a).

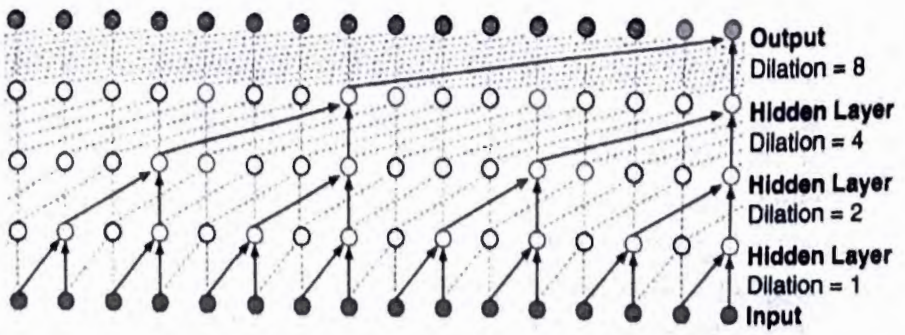

Figure 5.5 Visualization of a stack of dilated causal convolutional layers. Figure 3 from Oord et al. (2016a).

Despite the efficiency and expressiveness of Causal Masked Neural Network Models, challenges remain, particularly in the context of sampling. Generating new data points requires processing each element sequentially, which can be slow. However, the training process benefits from parallelism, significantly speeding up model learning. Moreover, concerns about the finite context window—how far back the model can look when generating the next part of a sequence—have been mitigated by advancements in attention mechanisms.

5.3.4 *Recurrent Neural Networks (RNN)*

As discussed in Chapter 3, Recurrent Neural Networks (RNNs) are well known for their ability to process sequential data with an effectively infinite look-back capability. This feature distinguishes them from other models that are limited to fixed-size inputs, enabling RNNs to dynamically incorporate all prior information in a sequence. This capability makes RNNs exceptionally well-suited for complex tasks such as speech recognition, natural language processing, and other time-series related applications.

To recap, at the core of an RNN is the concept of a hidden network state, often referred to simply as the "state." This state acts as the network's memory, theoretically holding all necessary past information to make future predictions. In a typical RNN, the hidden state at any sequence index t, denoted by h_t, is updated based on the previous state h_{t-1} and the current input x_t, according to a function g. The output y_t at each timestep depends on the current state h_t using a typical nonlinear function f,

$$\begin{aligned} h_t &= g(h_{t-1}, x_t) \\ y_t &= f(h_t) \end{aligned} \tag{5.7}$$

Here, f and g are typical nonlinear neural network layers that are independent of the specific index location t. Consequently, RNNs effectively employ parameter sharing across the sequence.

This recursive nature allows RNNs to handle variable input lengths and provides them with the theoretical capacity to manage an infinite window of context, which is ideal for generating coherent and contextually rich sequences over time.

However, the practical application of RNNs is often challenged by issues related to their deep and recursive structure. Notably, the training of RNNs can be subject to the phenomena of vanishing and exploding gradients. These occur when the gradient signals, essential for learning through backpropagation, diminish to insignificant levels (vanish) or grow too large (explode) as they are propagated back through each "timestep" in the sequence. Such issues can severely limit the RNN's ability to learn long-term dependencies within the data affecting the overall efficacy of the model in real-world applications.

In generative tasks, RNNs operate by predicting the next item in a sequence based on the history it has processed, with each output dependent on the preceding computational state. For example, when trained on specific datasets like the works of Shakespeare or encoded Wikipedia text, RNNs can generate text that mimics the style of the training data, capturing nuances such as punctuation, grammar, and even the specific formatting of text.

In the accompanying notebook [*RNN Notebook*], we demonstrate an experiment using RNNs to forecast price movements and returns of the S&P 500.

5.3.4.1 Practical Considerations and Limitations. While theoretically capable of handling sequences of arbitrary length and complexity, RNNs in practice face significant operational challenges:

- **Training Speed:** Training RNNs is slow due to the sequential nature of their computations, which prevents parallel processing. Each step depends on the completion of the previous step, leading to longer training times.
- **Gradient Issues:** The training of RNNs is notoriously difficult due to vanishing and exploding gradient problems, which can severely impact learning effectiveness. Techniques like Long Short-Term Memory (LSTM) units have been developed to mitigate these issues but do not entirely eliminate them.
- **Memory Requirements:** Due to their recursive nature, RNNs often require substantial memory for training, as they need to maintain information across many time steps.

5.3.5 Transformers

Transformer models have revolutionized both industry and academia in recent years. Since their introduction in the highly influential paper *Attention Is All You Need* by Vaswani et al. (2023), they have been successfully applied to a wide range of data types, including text, audio, time series, image, and video. As autoregressive models, Transformers offer several advantages over traditional RNNs. One of the most significant is their ease of training and the ability to leverage parallelization, which accelerates both training and inference. Additionally, Transformers mitigate issues such as vanishing and exploding gradients, which are more prevalent in earlier models like RNNs and LSTMs.

In this section, we will explore the Transformer architecture in detail, breaking down its core components. By the end of this section, we will have covered all the essential building blocks necessary to implement two of the most influential Transformer models: BERT, introduced by (Devlin et al. 2019) and developed by Google, and GPT, introduced by Radford et al. (2018) and developed by OpenAI. BERT, which stands for Bidirectional Encoder Representations from Transformers, revolutionized natural language processing, redefining how both industry and academia tackle a wide range of NLP tasks. Similarly, the GPT family of models (Generative Pre-trained Transformers)

by OpenAI has been groundbreaking, pushing the boundaries of what's possible with language generation and understanding.

For clarity of exposition, let's focus on text data, as it makes these models easier to understand. Keep in mind, however, that the same framework may be applicable to numerical time series, as both text and time series are considered "sequences." We will elaborate on this point further in Section 5.3.6. We will briefly cover the essential steps involved in a typical preprocessing pipeline for Natural Language Processing (NLP) applications, providing just enough information to understand Transformers. To do this, we will use a combination of code snippets and mathematical notation to illustrate how these models work.

5.3.5.1 Attention Mechanism. To understand the concept of attention, let's consider the following two sentences:

- *The startup achieved a new* ***round*** *of funding to expand operations.*
- *He invited his friends for another* ***round*** *of drinks at the bar.*

In the first sentence, the word "round" refers to a phase of funding for a company, while in the second sentence, it refers to a sequence of drinks being ordered. Intuitively, these different meanings suggest that the word "round" should be represented by different vectors, also known as vector embeddings—or simply embeddings in the NLP literature—depending on the context.

However, in the initial steps of the NLP pipeline—when raw text is transformed into token embeddings—the model assigns the same vector representation to the words "round" in both sentences, regardless of the context. This is because token embeddings, at this point, are **context-independent** and rely solely on the token ID assigned to the word.

The NLP Pipeline: From Raw Data to Token Embeddings:

1. **Tokenization:** The first step in modern NLP pipelines involves converting raw text into smaller units called **tokens.** Tokens can be full words, subwords, or even characters, depending on the tokenizer. Each token is mapped to a unique token ID.
2. **Token Embeddings:** The model assigns an initial embedding to each token based on the token ID. At this stage, the embeddings are purely **context-independent**—the same word will always have the same embedding, regardless of the surrounding text.

To illustrate this, we will use the Hugging Face (https://huggingface.co) `transformers` python library, which simplifies working with Transformer models like BERT. In the following code, we examine the tokenization process:

```
from transformers import BertTokenizer, BertModel
import torch

# Initialize BERT tokenizer and model
model_ckpt = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_ckpt)
model = BertModel.from_pretrained(model_ckpt)
```

```
# Two sentences with different meanings for "round"
sentence_1 = "The startup achieved a new round of funding to expand
operations."
sentence_2 = "He invited his friends for another round of drinks at the bar."

# Tokenize the sentences
inputs_1 = tokenizer(sentence_1, return_tensors='pt')
inputs_2 = tokenizer(sentence_2, return_tensors='pt')

# Get the token ID for "round"
token_id = tokenizer.convert_tokens_to_ids('round')
print(f"Token ID of word round: {token_id}")

# Print token IDs to show tokenization process
print("Token ids for sentence 1:", inputs_1.input_ids)
print("Token ids for sentence 2:", inputs_2.input_ids)



# Access the token embeddings directly from BERT's embedding matrix
# The embedding matrix is model.embeddings.word_embeddings
token_embedding_matrix = model.embeddings.word_embeddings.weight

# Extract the token embedding for "round"
token_embedding_matrix[token_id]


# Print results to verify that token embeddings are identical
print(f"Token embedding for 'round': {token_embedding_for_round_1[:5]}...")
# First 5 vavlues
>>>
Token ID of word round: 2461
Token ids for sentence 1: tensor([[
 101,  1996, 22752,  4719,
 1037,  2047,  2461,  1997,
 4804,  2000, 7818,  3136,
1012,  102]])
Token ids for sentence 2: tensor([[
101, 2002, 4778, 2010,
2814, 2005, 2178, 2461,
1997, 8974, 2012, 1996,
3347, 1012,  102]])
```

Here, the Token ID for the word "round" corresponds to the ID 2461, which is located at position 6 and position 7 (0-indexed) in `inputs_1.input_ids` for sentence 1 and `inputs_2.input_ids` for sentence 2, respectively.

To determine the token embeddings assigned to the word "round" let's access the token embedding matrix of BERT:

```
# Access the token embeddings directly from BERT's embedding matrix
# The embedding matrix is model.embeddings.word_embeddings
token_embedding_matrix = model.embeddings.word_embeddings.weight

# Extract the token embedding for "round"
token_embedding_for_round = token_embedding_matrix[token_id]
```

```
print(f"Embedding matrix size: {token_embedding_matrix.size()}")
>>>
Embedding matrix size: torch.Size([30522, 768])

print(f"Token embedding for 'round': {token_embedding_for_round[:5]}...")
>>
Token embedding for 'round': tensor([ 0.0109,  0.0577,
0.0310, -0.0615,  0.0571], grad_fn=<SliceBackward0>)...
```

The embedding matrix size for the BERT model is `torch.Size([30522, 768])`, meaning that each token ID is mapped to a 768-dimensional vector representation, and the vocabulary—the number of unique tokens for this specific model—consists of 30,522 tokens. More details about how these embedding representations are created and learned will be provided later. Additionally, more information about this process and the typical NLP pipeline can be found in the Hugging Face tutorial, located in the *chapter10* folder of the book repository.

The purpose of the attention mechanism is to refine the representation of each word in a sentence, making it more context-dependent. After applying attention, we obtain a more refined embedding that captures the context in which the word appears. The importance of the attention mechanism is so profound that the authors of the influential paper "Attention Is All You Need" (Vaswani et al., 2023) emphasized it in the very title. But how exactly does attention work to refine these embeddings?

Before diving into the mathematical details, let's explore some insightful visualizations using BertViz, a tool introduced by Vig (2019), which helps us understand how models like BERT, GPT2, or T5 update word representation through attention. This visualization will help us to grasp the concept before looking at the equations.

Most attention mechanisms refine embeddings by computing a linear combination of all input embedding using specific "attention weights." The attention mechanism makes these "attention weights" or "attention scores" dependent on the input, allowing the model to create context-dependent representations. To clarify this concept, we will use BertViz to see how BERT assign attention weights across different words in a sentence.

In the visualizations generated by the following code, namely Figures 5.6a and b, it is displayed the attention scores for two input sentences:

```
from transformers import AutoTokenizer
from transformers import AutoModel
from bertviz import head_view

model_ckpt = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModel.from_pretrained(model_ckpt, output_attentions=True)
sentence_a = 'The startup achieved a new round of funding to expand operations'
sentence_b = 'He invited his friends for another round of drinks at the bar'
```

```
viz_inputs = tokenizer(sentence_a, sentence_b,  return_tensors='pt')
attention = model(**viz_inputs).attentions
sentence_b_start = (viz_inputs.token_type_ids == 0).sum(dim=1)
tokens = tokenizer.convert_ids_to_tokens(viz_inputs.input_ids[0])

head_view(attention, tokens, sentence_b_start, heads=[8])
```

Here, we instructed BERT to process the two input sequences simultaneously. Both sentences are shown in the left and right column, separated by the special `[SEP]` token.

In these figures, the right column represents the initial input embeddings, while the left column shows the output embeddings after applying the attention mechanism. Although the visualizations do not show the final word embeddings, they display the

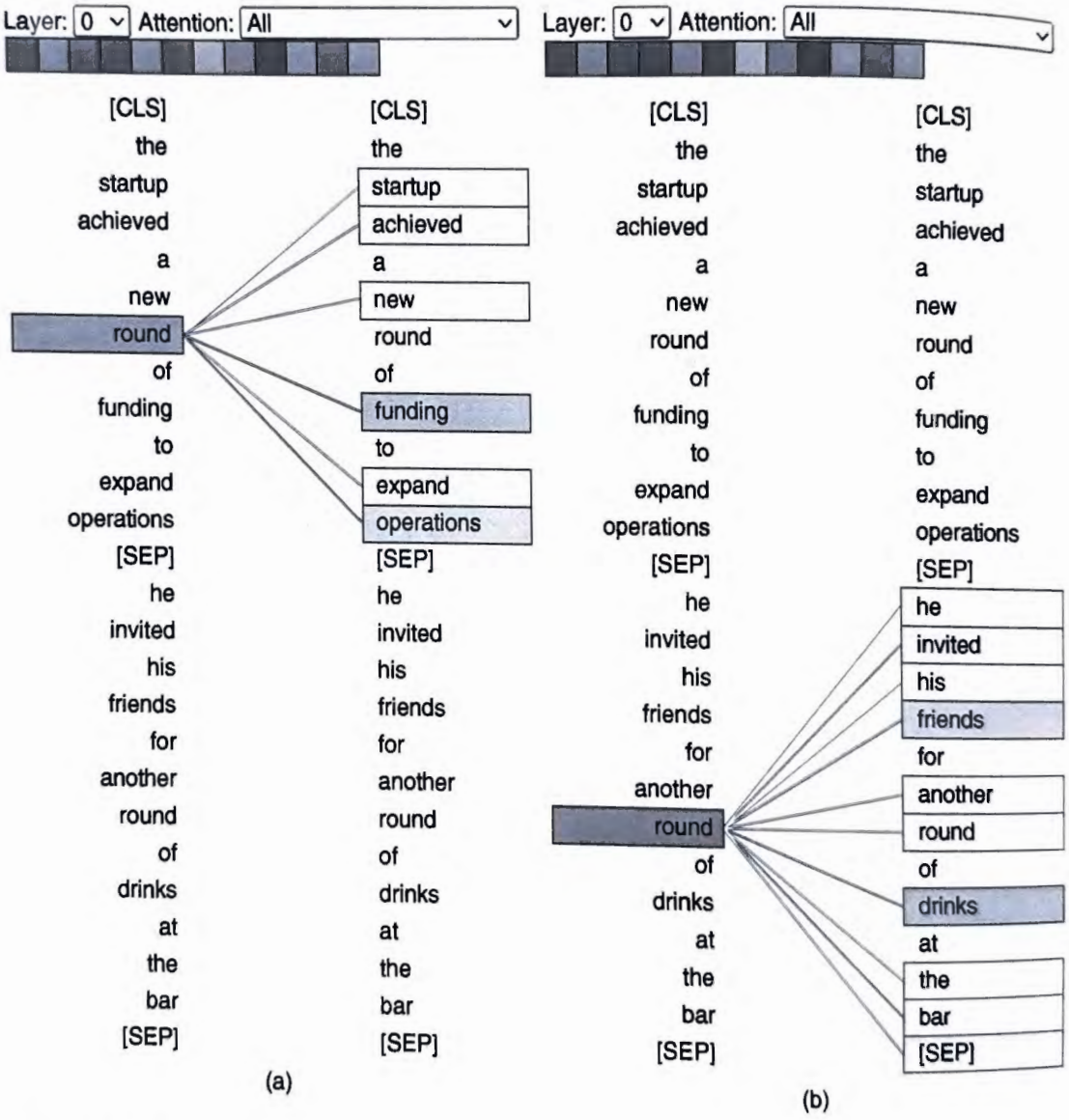

Figure 5.6 Visualizing attention.

attention scores between words. By selecting any word in the left column, highlighted with a gray background, we can see lines connecting this word to all words in the right column. The intensity of each line represents the magnitude of the attention score, ranging from 0 to 1, indicating how much influence each word in the right column has on the selected word in the left column.

Let's look at Figure 5.6a, where we focus on the word **round** in the first sentence, *"The startup achieved a new **round** of funding to expand operations."* The visualization shows that the words "funding," "operations," "startup," and "achieved" have the highest attention scores, contributing the most to refining the representation of "round." Notably, none of the words from the second sentence are considered important in updating this specific representation.

On the other hand, in Figure 5.6b, we select the word **round** from the second sentence, *"He invited his friends for another **round** of drinks at the bar."* In this context, the words "drinks," "friends," and "invited" are most significant in updating the representation of "round." Similarly, no significant attention is assigned to words from the first sentence.

These visualizations illustrate how attention helps the model focus on relevant parts of the context when refining word representations. Now, let's see how this attention update is computed.

If we are dealing with a stock's returns series instead, we might imagine that a return at time t may depend significantly on some past return $t' = t - \Delta$ that was particularly significant, perhaps because there was an earnings announcement or economic news at t' (Guijarro-Ordonez et al., 2021).

Now, let's see how this attention update is already computed. Let's say we have some embedding representation for each token in the input sequence, denoted as $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_N$, each of dimension $\mathbb{R}^{d_v}$. (In the case of a returns series, each v_t might be an embedding representation of the stock returns at time t in a portfolio.) As mentioned, most attention mechanisms create refined embeddings by computing a linear combination of all input embeddings, such as the $\mathbf{v}_s$ in our example, which, as we will see next, are linear combinations of the input embeddings $\mathbf{X}$. To find a refined representation for each input vector, we compute a weighted sum. For example, the refined representation, or hidden representation, $\mathbf{h}_1$ for $\mathbf{v}_1$ is computed as:

$$
\begin{aligned}
\mathbf{h}_1 &= a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + ... + a_N\mathbf{v}_N \\
&= \sum_n a_n\mathbf{v}_n \\
&= \begin{bmatrix} | & \cdots & | \\ \mathbf{v}_1 & \cdots & \mathbf{v}_N \\ | & \cdots & | \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \\
&= \begin{bmatrix} | & \cdots & | \\ \mathbf{v}_1 & \cdots & \mathbf{v}_N \\ | & \cdots & | \end{bmatrix} \begin{bmatrix} | \\ \mathbf{a}_1 \\ | \end{bmatrix}
\end{aligned}
\tag{5.8}
$$

where $\mathbf{a} = [a_1, a_2, ..., a_N]^T$ is the attention vector, also known as attention scores or attention weights, used to compute $\mathbf{h}_1$. Similarly, for the refined representation or hidden representation of the third vector, we have:

$$\mathbf{h}_3 = \begin{bmatrix} | & \cdots & | \\ \mathbf{v}_1 & \cdots & \mathbf{v}_N \\ | & \cdots & | \end{bmatrix} \begin{bmatrix} | \\ \mathbf{a}_3 \\ | \end{bmatrix} \tag{5.9}$$

In general, the refined representation for the i-th input is:

$$\mathbf{h}_i = \begin{bmatrix} | & \cdots & | \\ \mathbf{v}_1 & \cdots & \mathbf{v}_N \\ | & \cdots & | \end{bmatrix} \begin{bmatrix} | \\ \mathbf{a}_i \\ | \end{bmatrix} \tag{5.10}$$

This can be represented in matrix form as:

$$\begin{bmatrix} --\mathbf{h}_1^T-- \\ --\mathbf{h}_2^T-- \\ \vdots \\ --\mathbf{h}_N^T-- \end{bmatrix} = \begin{bmatrix} --\mathbf{a}_1^T-- \\ --\mathbf{a}_2^T-- \\ \vdots \\ --\mathbf{a}_N^T-- \end{bmatrix} \begin{bmatrix} --\mathbf{v}_1^T-- \\ --\mathbf{v}_2^T-- \\ \vdots \\ --\mathbf{v}_N^T-- \end{bmatrix} \tag{5.11}$$

$$\mathbf{H} = \mathbf{AV}$$

where $\mathbf{A}$ is the matrix of attention weights. In short, we update each token through a linear combination of all other tokens. In a financial time-series application for forecasting returns, to avoid look-ahead bias, we should only include past returns when calculating the current return. This approach is known as causal masked attention in NLP terminology. The question now is, how do we determine this matrix $\mathbf{A}$? Well, these weights are learned from data during training.

Before explaining how the attention matrix is computed, let's clarify the difference between *hard attention* and *soft attention*.

- **Hard Attention:** Each attention vector $\mathbf{a}_n$ satisfies the property $\|\mathbf{a}_n\|_0 = 1$, meaning it is a one-hot vector where only one component is 1 and the rest are 0. This means the new representation will be exactly one of the input vectors.
- **Soft Attention:** Each attention vector $\mathbf{a}_n$ satisfies $\|\mathbf{a}_n\|_1 = 1$, meaning the new representation is a weighted combination of the input vectors.

Although there are several methods to compute the attention matrix and the matrix of embedding vectors $\mathbf{V}$, in this book, we will focus on the most well-known approach, Scaled Dot-Product Attention, as introduced in the influential paper by Vaswani et al. (2023).

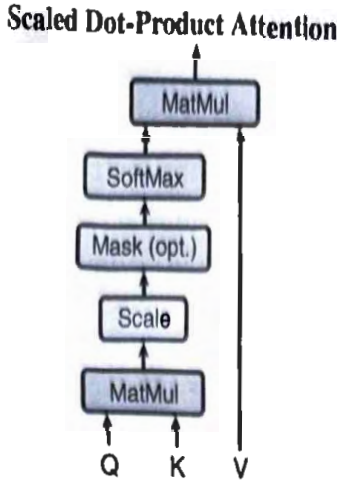

Figure 5.7 Scaled Dot-Product Attention. Figure 2 (left) from Vaswani et al. (2023).

5.3.5.2 Scaled Dot-Product Attention. In Scaled Dot-Product Attention, the attention scores $\mathbf{A}$ are computed as:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \tag{5.12}$$

and so, the hidden representation is computed as:

$$\begin{aligned} \mathbf{H} &= \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \\ \mathbf{H} &= \mathbf{A}\mathbf{V} \end{aligned} \tag{5.13}$$

The operations of the Scaled Dot-Product Attention are illustrated in Figure 5.7, serving as a useful reference for understanding the process.

Let's break down the equation for Scaled Dot-Product Attention.

- $\mathbf{Q}$ (Query Matrix): This matrix is computed from the input embeddings $\mathbf{X}$ as $\mathbf{Q} = \mathbf{X}\mathbf{W}_q$. For now, consider the input embeddings $\mathbf{X}$ as those generated from the basic NLP pipeline explained earlier, where they are assigned based on token ID. In Section 5.3.5.4, we will discuss how additional sequence information is incorporated into the token embeddings to fully create $\mathbf{X}$. The query matrix $\mathbf{Q}$ contains N row vectors, each representing a query vector $\mathbf{q}_n^T$ of dimension d_q. The matrix $\mathbf{W}_q$ is a learnable parameter of size $d \times d_q$.
- $\mathbf{K}$ (Key matrix): Similarly, the Key matrix is computed as $\mathbf{K} = \mathbf{X}\mathbf{W}_k$. Each row of $\mathbf{K}$ corresponds to a key vector $\mathbf{k}_n^T$ of dimension d_k. The matrix $\mathbf{W}_k$ is also learnable parameter of size $d \times d_k$.
- $\mathbf{V}$ (Value matrix). The Value matrix is computed as $\mathbf{V} = \mathbf{X}\mathbf{W}_v$. It contains N row vectors, each representing a value vector $\mathbf{v}_n^T$ of dimension d_v. The matrix $\mathbf{W}_v$ is a learnable parameter of size $d \times d_v$.

For Scaled Dot-Product Attention to work, the dimensions d_q and d_k must match, and a common choice is $d_q = d_k = d$.

The attention mechanism used here is a soft-attention mechanism, enforced by applying the softmax function to ensure that the rows of $\mathbf{A}$ sum to 1. The scaling factor $1/\sqrt{d_k}$ was introduced by the authors of the *Attention Is All You Need* paper to prevent the argument of the softmax from becoming too large, thereby avoiding regions of the function with very small gradients (Vaswani et al., 2023).

To better understand Scaled Dot-Product Attention, let's focus on the computation of the unnormalized attention scores:

$$\tilde{\mathbf{A}} = \mathbf{Q}\mathbf{K}^T \tag{5.14}$$

We can call this matrix $\tilde{\mathbf{A}}$ the unnormalized attention scores. For every input vector in the sequence $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N$, we compute its query representation $\mathbf{q}_1, \mathbf{q}_2, ..., \mathbf{q}_N$, its key representation $\mathbf{k}_1, \mathbf{k}_2, ..., \mathbf{k}_N$, and its value representation $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_N$, and using this information, we compute the attention weights.

For example, to compute the attention weights $\tilde{\mathbf{a}}_1$ needed to the hidden representation $\mathbf{h}_1$ of $\mathbf{x}_1$, we take the dot product of its query vector $\mathbf{q}_1$ with the key vectors in the sequence:

$$\begin{bmatrix} -- \tilde{\mathbf{a}}_1^T -- \end{bmatrix} = \begin{bmatrix} -- \mathbf{q}_1^T -- \end{bmatrix} \begin{bmatrix} | & | & | & | \\ \mathbf{k}_1 & \mathbf{k}_2 & \dots & \mathbf{k}_N \\ | & | & | & | \end{bmatrix} \tag{5.15}$$

The terminology and concepts used in the *Attention Is All You Need* paper by Vaswani et al. (2023) draw from information retrieval, where we have a database or key-value store indexed by keys with corresponding values. To retrieve a value, we compute the similarity between a query and all the keys in the database. The retrieved value can be the one with the maximum score or a weighted sum of values based on these scores, which is the essence of self-attention.

After computing the unnormalized attention scores $\tilde{\mathbf{a}}_1$, we normalize them using the softmax function

$$\mathbf{a}_1 = \text{softmax}\left(\frac{\tilde{\mathbf{a}_1}}{\sqrt{d_k}}\right) \tag{5.16}$$

Now, we can compute the hidden representation $\mathbf{h}_1$ by taking a linear combination of all the values in the key-value store, following the same key-store analogy:

$$\begin{bmatrix} -- \mathbf{h}_1^T -- \end{bmatrix} = \begin{bmatrix} -- \mathbf{a}_1^T -- \end{bmatrix} \begin{bmatrix} -- \mathbf{v}_1^T -- \\ -- \mathbf{v}_2^T -- \\ \vdots \\ -- \mathbf{v}_N^T -- \end{bmatrix} \tag{5.17}$$

This produces a refined representation $\mathbf{h}_i$ for each input $\mathbf{x}_i$, incorporating information about the whole sequence.

The combination of Scaled Dot-Product Attention with the three linear transformations for the query, key, and value vectors is what's known as an Attention Head

in Transformer models. (Multiple heads can be used to capture different relationships between words, as will be explained later.) Let's implement this concept in PyTorch to gain a better understanding. The following implementation is adapted from the book *Natural Language Processing with Transformers*, by Tunstall et al. (2022).

To implement Scaled Dot-Product Attention, we need to pass in the query, key, and value matrices. Here is a simple implementation of the function in PyTorch:

```
import torch
from math import sqrt
import torch.nn.functional as F

def scaled_dot_product_attention(query, key, value):
    dim_k = query.size(-1)
    unnormalised_scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_k)
    scores = F.softmax(unnormalised_scores, dim=-1)
    return torch.bmm(scores, value)
```

In this function:

- We use `torch.bmm` for batch matrix multiplication, which efficiently handles the batch dimension during computation.
- The scores are scaled by dividing by the square root of the dimension of the key vectors (`sqrt(dim_k)`) to stabilize the dot products and prevent them from becoming too large.
- We apply the `softmax` function to the scores to obtain attention weights and then use these weights to compute a weighted sum of the value vectors.

Next, we'll implement the full attention head. This requires creating the matrices $\mathbf{W}_k$, $\mathbf{W}_q$, and $\mathbf{W}_v$ for the key, query, and value transformations, respectively. Each of these matrices transforms the input embeddings into the required dimensions query, key, and value dimensional spaces.

We define these matrices in the following AttentionHead class. For simplicity, we assume $d_k = d_q = d_v$ = head dimension, which we refer to as `head_dim` in the code:

```
from torch import nn

class AttentionHead(nn.Module):
    def __init__(self, embed_dim, head_dim):
        super().__init__()
        self.q = nn.Linear(embed_dim, head_dim)
        self.k = nn.Linear(embed_dim, head_dim)
        self.v = nn.Linear(embed_dim, head_dim)

    def forward(self, hidden_state):
        attn_outputs = scaled_dot_product_attention(
            self.q(hidden_state),
            self.k(hidden_state),
            self.v(hidden_state)
            )

        return attn_outputs
```

In this implementation:

- The `__init__` method initializes three linear transformations for the query, key, and value matrices. These transformations project the input embeddings into the head_dim dimensional space.
- In the `forward` method, the hidden states are passed through the linear transformations to obtain the query, key, and value matrices.
- The `scaled_dot_product_attention` function is called with these matrices to compute the attention outputs.

5.3.5.3 From Self-attention to Transformers. In this section, we will explain how to transition from the self-attention mechanism discussed earlier to the complete Transformer architecture introduced in the influential paper by Vaswani et al. (2023). The architecture of the Transformer model can be visualized in Figure 5.8.

The Transformer model was originally developed for machine translation tasks, which traditionally follow an encoder-decoder structure. The role of the encoder is to map the input sequence—such as the sentence in the source language—into a sequence of hidden representations. These hidden representations are then used by the decoder to generate the translated output sequence, one token at a time. This process is autoregressive, meaning the decoder generates each token by considering the previously generated tokens as additional inputs. In the Transformer architecture, the encoder is depicted on the left side of Figure 5.8, while the decoder is shown on the right.

As seen in the figure, we need several components for building the complete Transformer model, including Input Embeddings, Multi-Head Attention, and Feed-Forward blocks. These components are fundamental to both the encoder and decoder sections. To build a comprehensive understanding of the Transformer model, we will explore each of these components in details by providing a code implementation of the encoder section.

Additionally, it's important to note that self-attention allows the model to attend to all tokens simultaneously, which is not ideal for tasks such as sequence generation, where the model needs to distinguish between past and future tokens, such as in financial time series. To address this, the self-attention mechanism can be modified to create what is known as causal masked self-attention, which ensures that each token only attends to previous tokens in the sequence. This modification preserves the causal structure needed for generating text in an autoregressive manner.

5.3.5.4 Positional Encodings. Positional encoding enables self-attention to consider the order of tokens in the sequence, addressing a limitation of the self-attention mechanism, which does not account for positional information in the sequence.

Positional encodings are combined to token embeddings so that the self-attention mechanism can incorporate the order of tokens in the sequence. These encodings can be created using a fixed mathematical formula or learned from data. Common methods for integrating token embeddings with positional encodings include concatenation and addition. The most widely used approach is to add the positional encodings to the token

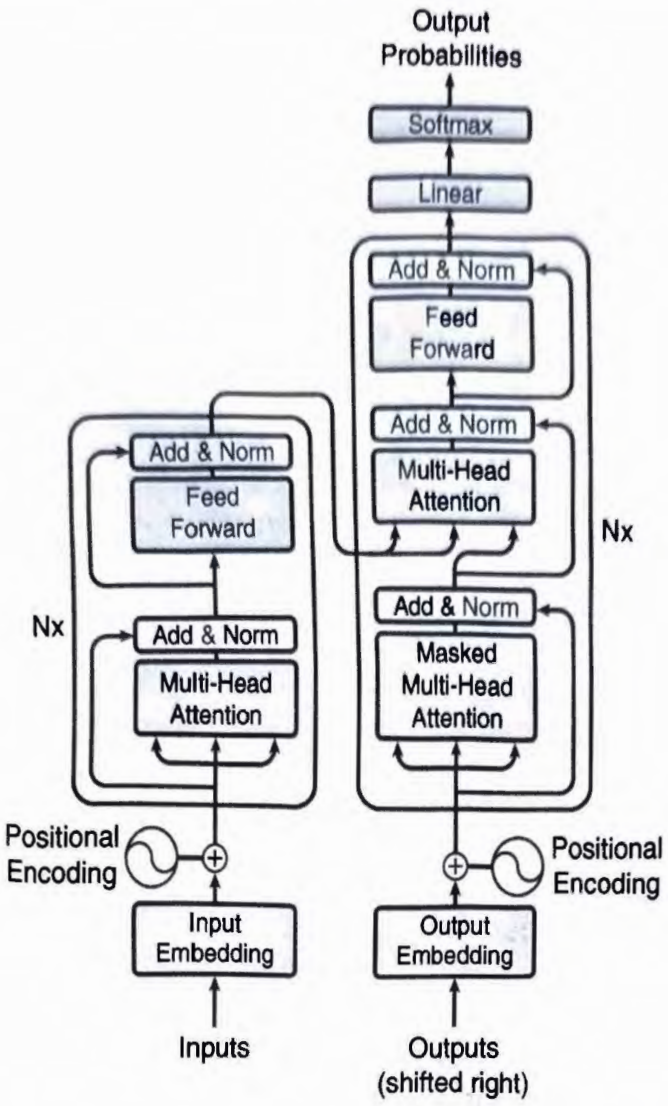

Figure 5.8 The Transformer encoder-decoder architecture, developed for machine translation, processes an input sequence—such as a sentence in the source language—into continuous representations via the encoder. The decoder then generates the translated output sequence, producing one token at a time. Input embeddings correspond to the representation of the source language, while output embeddings correspond to the representation of the target language. The output embeddings are offset by one position to ensure that the prediction for position i depends only on the known outputs at positions less than i. Figure 1 from Vaswani et al. (2023).

embeddings directly. It's important to note that positional encodings depend only on the position within the sequence.

With positional encodings, the input to the `AttentionHead` is no longer just token embeddings, but the addition of token embeddings and positional information.

The following code implements a class that creates both token and positional embeddings and adds them together.

```
class Embeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embeddings = nn.Embedding(config.vocab_size,
                                             config.hidden_size)
```

```
        self.position_embeddings=
nn.Embedding(config.max_position_embeddings,
                                                config.hidden_size)
        self.layer_norm = nn.LayerNorm(config.hidden_size, eps=1e-12)
        self.dropout = nn.Dropout()

    def forward(self, input_ids):
        # Create position IDs for input sequence
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length,
dtype=torch.long).unsqueeze(0)
        # Create token and position embeddings
        token_embeddings = self.token_embeddings(input_ids)
        position_embeddings = self.position_embeddings(position_ids)
        # Combine token and position embeddings
        embeddings = token_embeddings + position_embeddings
        embeddings = self.layer_norm(embeddings)
        embeddings = self.dropout(embeddings)
        return embeddings
```

In this class:

- The `__init__` method creates both token and position embeddings, as well as a layer normalization and dropout layer.
- The `forward` method combines token and position embeddings by addition, normalizes them, and applies dropout.

5.3.5.5 Multi-headed Attention. The self-attention mechanism involves a single set of projections for generating the query, key, and value representations. In practice, the authors in Vaswani et al. (2023) found it beneficial to use multiple sets of these transformations, known as attention heads. These heads are computed in parallel and then combined by concatenation, which joins the vectors end-to-end, to create a richer representation of

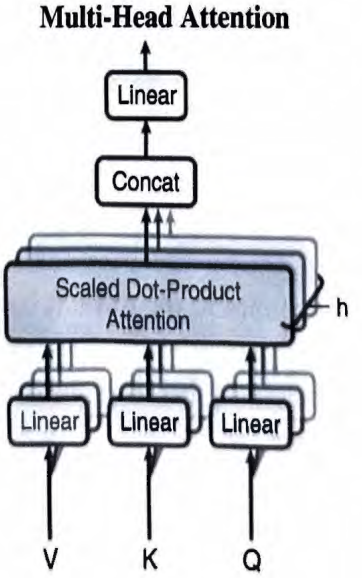

Figure 5.9 Multi-head Attention. Figure 2 (right) from Vaswani et al. (2023).

the input, see Figure 5.9 for a schematic representation of this process. Multi-head attention enables the model to focus simultaneously on different aspects of the input.

Multi-head attention allows for different dimensions for the query, key, and value projections. However, in practice, each attention head typically has the same dimension. In addition to the dimensions of these projection matrices, we must specify the number of attention heads. For example, in the original Transformer architecture, the input embedding dimension is 512, and there are 8 attention heads. This configuration sets the dimension for each head as $d_v = d_k = d_q = d_{\text{head}} = 512/8 = 64$.

An implementation of the following `MultiHeadAttention` class demonstrates this concept:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        embed_dim = config.hidden_size
        num_heads = config.num_attention_heads
        head_dim = embed_dim // num_heads
        self.heads = nn.ModuleList(
            [AttentionHead(embed_dim, head_dim) for _ in range(num_heads)]
        )
        self.output_linear = nn.Linear(embed_dim, embed_dim)

    def forward(self, hidden_state):
        x = torch.cat([h(hidden_state) for h in self.heads], dim=-1)
        x = self.output_linear(x)
        return x
```

In this implementation:

- The `__init__` method creates multiple attention heads using the `AttentionHead class`, based on the specified number of heads.
- The `forward` computes attention heads outputs and concatenate them, which is them passed through a final linear layer.

5.3.5.6 The Feed-forward Layer. The feed-forward layer adds nonlinearity to the model and is applied to the output of the multi-head attention layer. It consists of two linear transformations with a nonlinear activation function in between, as shown here:

```
class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.linear_1 = nn.Linear(config.hidden_size, config.intermediate_size)
        self.linear_2 = nn.Linear(config.intermediate_size, config.hidden_size)
        self.gelu = nn.GELU()
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, x):
        x = self.linear_1(x)
        x = self.gelu(x)
        x = self.linear_2(x)
```

```
        x = self.dropout(x)
        return x
```

This class performs the following steps:

- The `__init__` method defines two linear layers, a GELU activation function, and dropout for regularization.
- The `forward` method applies the first linear transformation, the GELU activation, the second linear transformation, and finally dropout before returning the output.

In the original paper, the dimensionality of the input and output is 512, referred in the code as `hidden_size`, while the inner layer has a dimensionality of 2048, referred in the code as `intermediate_size`.

The multi-head attention, feed-forward layers, and positional encodings together form the building blocks of the Transformer model. They enable the model to capture complex dependencies and learn rich representations from the input data.

5.3.5.7 Add and Norm Blocks. As depicted in Figure 5.8, there are Add & Norm blocks integrated into the Transformer architecture.

These blocks perform two key operations: **Add**, which applies a residual connection (i.e., adding a layer's input to its output to improve gradient flow and facilitate training), and **Norm**, which involves layer normalization (see Goodfellow et al., 2016, for more details). As shown in the figure, the authors use a residual connection around both Multi-head Attention and feed-forward blocks.

5.3.5.8 The Transformer Encoder Layer. Now that we've covered the core components of the Transformer architecture, we are ready to implement the Transformer encoder layer, as shown in the left of Figure 5.8. The Encoder layer, also known as Encoder block, consists of an embedding layer, a multi-head attention mechanism, and a feed-forward network. As described, batch normalization and residual connections are also included.

Following is the implementation of the `TransformerEncoderLayer`, incorporating layer normalization and skip connections for better gradient flow:

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.layer_norm_1 = nn.LayerNorm(config.hidden_size)
        self.layer_norm_2 = nn.LayerNorm(config.hidden_size)
        self.attention = MultiHeadAttention(config)
        self.feed_forward = FeedForward(config)

    def forward(self, x):
        # Apply layer normalization and then copy input into query, key, value
        hidden_state = self.layer_norm_1(x)
        # Apply attention with a skip connection
        x = x + self.attention(hidden_state)
        # Apply feed-forward layer with a skip connection
        x = x + self.feed_forward(self.layer_norm_2(x))
        return x
```

A note on regularization: As described by the authors of the Transformers paper, and as you may have noticed in the code implementation, dropout (see Chapter 3) is used for regularization. Dropout is applied around the Add & Norm blocks, as well as to the sum of the embeddings and positional encodings in both the encoder and decoder stacks (Vaswani et al., 2023).

5.3.5.9 The Complete Transformer Encoder. The Transformer model is constructed by stacking multiple `TransformerEncoderLayer` modules. Each layer processes the input embeddings and refines them further, allowing the model to learn complex relationships within the data.

Here's how we can implement the complete Transformer encoder:

```
class TransformerEncoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.embeddings = Embeddings(config)
        self.layers = nn.ModuleList([TransformerEncoderLayer(config)
                                     for _ in
range(config.num_hidden_layers)])

    def forward(self, x):
        x = self.embeddings(x)
        for layer in self.layers:
            x = layer(x)
        return x
```

5.3.5.10 Model Objective. Now that we have defined the Transformer encoder architecture, the next question is: What tasks can we solve with it? how do we structure the model's output for solving these tasks and train the model?

The typical approach is to use the Transformer encoder model as the core *body* of our model and attach either a decoder, as in language translation tasks, or a specific *head* that is suited to the task at hand. For instance, for a classification task, we can attach a specific head, a network layer that takes the output of the Transformer and generates a probability distribution over the possible classes. This allows us to use an optimization algorithm to find the model parameters that best explain the relationship between inputs and targets in our dataset.

Let's consider a few examples to understand this concept better:

- **Sentiment Analysis:** For this task, we would use a classification head that processes the output of the Transformer model and predicts the sentiment (for example, positive, negative, or neutral) of a given input text. The model learns to associate specific patterns in the text with specific sentiments.
- **Language Modeling:** The BERT model, introduced by (Devlin et al. 2019), is a good example of this approach. BERT stands for Bidirectional Encoder Representations from Transformers and uses the Transformer encoder architecture. One of BERT's primary training objectives is the Masked Language Model (MLM) task, where certain words in a sentence are randomly masked, and the model is trained to predict

these missing words. Specifically, 15% of the words in a sentence are masked. Of these, 80% are replaced with the [MASK] token, 10% remain unchanged, and the final 10% are substituted with a random word. To solve this task, we would use a classification head that processes the output of the Transformer and generates a probability distribution over the input's vocabulary.

To adapt the Transformer encoder for a classification task like sentiment analysis, we add a classification head on top of the encoder. As an example, the head implemented here consists of a dropout layer followed by a fully connected layer that maps the encoded representation to the desired number of labels:

```
class TransformerForSequenceClassification(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.encoder = TransformerEncoder(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

    def forward(self, x):
        x = self.encoder(x)[:, 0, :] # select hidden state of [CLS] token
        x = self.dropout(x)
        x = self.classifier(x)
        return x
```

- `self.encoder`: This is the Transformer encoder that processes the input sequence.
- `self.dropout`: This layer is used to prevent overfitting during training.
- `self.classifier`: This fully connected layer maps the encoder's output to the desired number of classes.

As just shown, the classification model leverages the Transformer encoder to create powerful representations of the input text. By attaching a classification head, we can train this model for various tasks, such as sentiment analysis, language modeling, question answering, and more. However, the typical way these models are trained to solve downstream tasks, such as sentiment analysis, involves a two-step training process. First, in the pre-training phase, the Transformer encoder is trained on a language modeling task. Then, in the second step, known as fine-tuning, the model is further trained on a specific dataset related to the task, and a classification or regression head is added to solve specific problems. More details on this process, along with practical examples, can be found in Chapters 9 and 10.

In fact, the structure described here is essentially an implementation of the BERT model, which has become a foundational tool for many NLP applications due to its versatility and powerful performance across a wide range of tasks.

For complementary purposes, the accompanying notebook [*DecoderOnly Transformers—Notebook*] showcases an implementation of a decoder-only transformer model, specifically GPT, to complement the discussion of decoder-only transformers presented here.

5.3.6 *From NLP Transformer to the Time-series Transformers*

Given the great success of Transformer models in capturing long-range dependencies in sequential data—such as text, audio, and video—it is natural to wonder if they can also be applied to time-series applications, as time-series data, like text, is inherently sequential.

Transformers are known for several advantages, which we will discuss in more detail in Chapter 10. To mention a few, these models improve their performance in NLP tasks as model size, training data volume, and training time increase (see Kaplan et al., 2020). They are also highly parallelizable and adaptable for multiple downstream tasks, as demonstrated by Devlin et al. (2019). These qualities are motivating the industry to explore whether Transformers can achieve a similar level of success in modeling time-series problems.

For language modeling tasks, Transformers are trained with the objective of predicting the next token in a sequence based on the previous tokens. This is conceptually similar to the time-series forecasting task, where we want to predict the next element of the series based on previous values, or perform what is called a "one-step-ahead forecast." In time-series terminology, we could say that the language modeling task is akin to forecasting a discrete time series where the elements in the sequence take values from a finite vocabulary, with a time horizon of one. Given this similarity in objectives, we might wonder what modifications are needed to apply Transformers to time series.

As hinted, one key difference is that most of the time, the values in a time series are unbounded, continuous quantities, unlike in a text sequence, where elements (called tokens) take values from a finite vocabulary. A straightforward idea for adapting Transformers to time-series data is to convert the continuous values of the series into a finite set of values, or tokens, drawing an analogy to text applications, similar to how words are represented in language tasks. This approach is taken in *Chronos: Learning the Language of Time Series* (Ansari et al., 2024).

5.3.6.1 Discretizing Time-series Data: The Chronos Approach. In the paper *Chronos: Learning the Language of Time Series* (Ansari et al., 2024), the authors propose an approach for using Transformers on time-series data by **tokenizing** time-series values. This is achieved by first normalizing and then quantizing the time series, transforming continuous values into discrete "tokens" that can be processed by standard Transformer architectures.

1. Normalization: Each time series is normalized to mitigate scaling differences between series and improve model optimization. The authors use mean scaling, in which each series is normalized by the mean absolute historical value within the "historical context"—the number of inputs to the Transformer.
2. Quantization: The normalized values are then quantized into a fixed number of bins, converting continuous values into a discrete set of values, or tokens. These discrete tokens represent the time series in a way similar to how text is represented by a sequence of tokens.

Once each value has been tokenized, **token embeddings** and **positional embeddings** are created in the same way as for language modeling tasks and combined by addition to form the input embeddings for the Transformer. At inference time, since the model outputs tokens, a process of **dequantization** and **denormalization** is applied to convert discrete token predictions back into continuous values.

This method allows the use of standard Transformer architectures, such as T5 (Raffel et al., 2020; encoder-decoder) and GPT-2 (Radford et al., 2019; decoder-only), without architectural modifications. The authors of Chronos demonstrated that this approach is effective across a wide range of time-series data, achieving strong results on out-of-sample data. To train their models, the authors compiled time-series data from various domains, including retail, energy, finance, healthcare, and climate science, enhancing generalization by incorporating synthetic data generated through data augmentation techniques.

In the accompanying notebook [*Chronos Notebook*], we explore the use of Chronos on the Exchange Rate dataset, which contains daily exchange rates for currencies of eight countries (Australia, United Kingdom, Canada, Switzerland, China, Japan, New Zealand, and Singapore) from 1990 to 2016 (Exchange Rate data). Part of this data was used to train Chronos, and according to the references provided in their repository and paper, we evaluate Chronos's performance on the unseen part of the data, highlight some of its limitations, and compare it with some simple baselines.

5.3.6.2 Continuous Input for Transformers: The Lag-Llama Approach. An alternative approach to discretizing the time series is to treat the data as it is, using continuous values so that the network takes continuous inputs and outputs continuous values. This approach not only requires preprocessing the input to create "token embeddings," but also adjustments to the output, as we are now solving a regression task rather than a classification task, as in language modeling. This is the approach taken in the paper *Lag-Llama: Towards Foundation Models for Probabilistic Time Series Forecasting* (Rasul et al., 2024).

In this work, the authors adapt the Llama model (Touvron et al., 2023), a decoder-only Transformer architecture, to output continuous values rather than discrete tokens by introducing a final layer that predicts the parameters of continuous distributions, specifically a t-distribution, rather than the parameters of a categorical distribution typical in multiclass classification problems.

The modifications in Lag-Llama include the following:

1. Probabilistic continuous output layer: Instead of predicting the parameters of a categorical distribution that allocates probability mass over a finite vocabulary of tokens, Lag-Llama outputs the parameters of a t-distribution, allowing the model to make probabilistic forecasts over continuous values.
2. Lagged and date-time features: In addition to using the current time step, Lag-Llama incorporates lagged features—previous values at various intervals (e.g., quarterly, monthly, weekly)—to enrich the input and capture seasonality and periodic patterns. The model also includes features such as the hour, day, and month, etc., derived

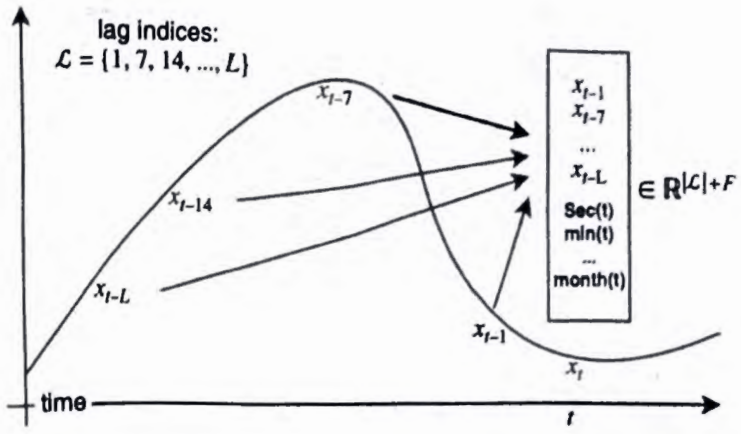

Figure 5.10 An illustration showing how the current and lagged values of the series, combined with engineered date-time features, are concatenated into a vector. Each component of the vector represents $|\mathcal{L}|$ past value of x_t (shown in blue), along with F temporal covariates (date-time features) derived from the timestamp (shown in red). Figure 1 from Rasul et al. (2024).

from the timestamp, which provide additional contextual information about each time point. The authors also include summary statistics of the series to provide additional context.

3. Normalization and outlier control: Lag-Llama normalizes each time series and mitigates outliers using the interquartile range (IQR).

By enriching the current input with lagged and date-time features, the authors provide the Llama model with more contextual information to capture long-term and short-term dependencies effectively. In our experience with forecasting tasks involving large datasets of multivariate time series, this type of feature engineering helps models capture seasonal characteristics and improves forecasting performance.

In the Lag-Llama approach, the current and lagged values of the series, along with engineered date-time features, are concatenated into a vector. See Figure 5.10 for an illustration of how these features are constructed. This vector is then projected into the model's "token embedding" space, creating an analogous "token representation" as in NLP applications, which serves as the input to the Transformer model, as shown in Figure 5.11.

In the accompanying notebook [Lag-Llama Notebook], we explore the use of Lag-Llama on the same Exchange Rate dataset. For Lag-Llama, since this dataset has not been used for training the model according to the authors, we will evaluate its "zero-shot" performance. As an additional exercise, we will fine-tune the model on this dataset and observe how its performance is affected. Similar to Chronos, we will highlight some of its limitations and compare its performance with simple baselines.

5.3.6.2.1 Innovations and Approaches. The examples of Chronos and Lag-Llama illustrate some of the most common techniques for adapting Transformer architectures for time-series applications, which in these cases required few to no modifications to existing Transformer architectures. In these cases, much of the work revolves around

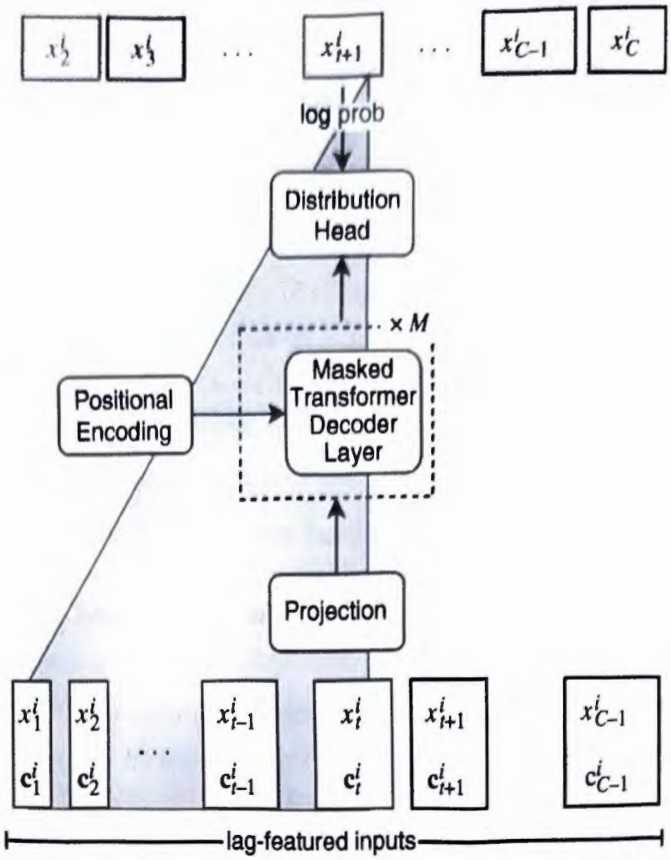

Figure 5.11 An illustration showing how the model input, comprising the time-series value at timestep t with $|\mathcal{L}|$ lags, F date-time features, and summary statistics, is projected into the transformer's "token embedding" space, creating a "token representation" similar to NLP applications. Figure 2 from Rasul et al. (2024).

techniques for preprocessing time-series data and enriching contextual information, using methods such as normalization, quantization, and the addition of lagged values and datetime features. We also saw how different preprocessing choices, such as applying quantization or not, affect how the problem is framed: as a classification problem (as in Chronos, which requires no changes to standard Transformer architectures and treats the problem similarly to text tasks) or as a regression problem (as in Lag-Llama, where the authors modified the last layer of the architecture to model a probability distribution over continuous values).

While these examples showcase a few interesting adaptations, they are by no means representative of the full range of approaches that can be applied to improve model performance. In particular, there are approaches that focus on improving the space and time complexity of Transformers, which we have not yet discussed.

Over the past few years, numerous Transformer architectures for time series have been proposed. To the best of our knowledge, the main areas of work are around the following:

1. Input Normalization and Contextual Information: This area includes scaling time-series data to help model optimization, applying various normalization techniques, handling outliers, and quantizing the series. Context can be enriched by

incorporating lagged values or creating datetime features to capture important seasonal and calendar effects, or incorporating macroeconomic indicators, as well as indicators from markets or indices that are not part of the target time series—some of which were outlined in Chapter 3, among others. This information is typically projected to create "token embeddings," which serve as inputs to the Transformer. For example, some models use one-dimensional convolutional layers as projection layers, as seen in the Informer architecture (Zhou et al., 2021).

2. Positional Encoding: Transformers are permutation-invariant by nature, posing a challenge when handling sequential data like time series. Positional encoding is crucial to convey order information in NLP tasks and becomes even more vital for time-series applications, where the sequence order is essential. Solutions range from fixed positional encodings, as used in the original Transformer (Vaswani et al., 2023), to learnable positional embeddings like the one discussed in Section 5.3.5.4.
3. Attention Layer efficiency: Improving the efficiency of the attention mechanism is an active area of research, especially for time-series data, which often involves long historical sequences and forecasting over extended horizons. Improving computational complexity of this layer can help reduce inference latency and help us to meet the system requirements of specific applications. Recall from our earlier discussion in Section 5.3.5.2 that the vanilla Transformer's attention mechanism (Vaswani et al., 2023), computes a dot product between each query and all keys, resulting in a time complexity of $\mathcal{O}(N^2)$ for N inputs. To reduce this complexity, several methods have been proposed. For example, the Informer model (Zhou et al., 2021) reduces the computational cost of the attention layer by computing attention for a subset of $\log N$ "active queries," rather than for all N queries, thus, achieving a time complexity $\mathcal{O}(N \log N)$.
4. New transformers architecture variants: Researchers have developed architecture variants to account for the multi-resolution nature of time series. For example, the Informer architecture down-samples the series by inserting max-pooling layers between attention blocks (Zhou et al., 2021). The Pyraformer model (Liu et al., 2021) introduces a pyramidal attention module that summarizes features at different resolutions, enabling it to model temporal dependencies over varying time scales effectively.

For a more detailed overview of the most well-known Transformer architectures for time series at the time of writing, we recommend the excellent resource *Transformers in Time Series: A Survey* (Wen et al., 2023). Additionally, for an application of transformers as building blocks in portfolio construction, see Cong et al. (2021).

5.4 Model Fitting

Autoregressive models are often referred to as Maximum Likelihood models, since they primarily rely on Maximum Likelihood Estimation (MLE) for parameter estimation. All the models discussed in this chapter use MLE as the main method for parameter

learning. Once we define a model class parametrized by θ, denoted as p_θ, it becomes essential to establish a metric that evaluates the model's ability to approximate the data probability distribution, denoted as p_{data}. Of course, we discussed MLE fitting already in Chapter 3, but here is a new way to think about it.

Considering that our goals include computing probability densities, determining hidden structures, and generating new samples, determining a single model scoring metric that fulfills all these objectives can be challenging. Intuitively, if a model can closely approximate the distribution p_{data}, it should theoretically perform well in the other tasks. Therefore, our objective is to align the model distribution p_θ as closely as possible with p_{data}.

To quantify the closeness between two distributions, we use the Kullback-Leibler divergence, $\text{KL}(P\|Q)$, defined as:

$$\text{KL}(P\|Q) = \mathbb{E}_{\mathbf{x}\sim P}\left[\log\frac{P(x)}{Q(x)}\right] \tag{5.18}$$

Our goal is to minimize the Kullback-Leibler divergence between the empirical data distribution and our model's distribution:

$$\begin{aligned}\text{KL}(p_{\text{data}}\|p_\theta) &= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\log\frac{p_{\text{data}}(\mathbf{x})}{p_\theta(\mathbf{x})}\right] \\ &= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\log p_{\text{data}}(\mathbf{x}) - \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\log p_\theta(\mathbf{x})\end{aligned} \tag{5.19}$$

The term $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\log p_{\text{data}}(\mathbf{x})$ does not depend on θ and is thus irrelevant to the optimization process. Hence, the objective function, cost function, or score to minimize is given by:

$$\underset{\theta}{\text{argmin}}\ \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} - \log p_\theta(\mathbf{x}) \tag{5.20}$$

Given that we only have access to samples from p_{data}, the empirical estimation is:

$$\underset{\theta}{\text{argmin}}\ \mathcal{L}(\theta) = \frac{1}{N}\sum_{\mathbf{x}_n\in\mathcal{D}} - \log\ p_\theta(\mathbf{x}_n) \tag{5.21}$$

This formulation corresponds to Maximum Likelihood Estimation, or minimizing the Negative Log Likelihood, which aims to increase the probability mass at points present in our training dataset while assigning lower probabilities to those that are absent.

To minimize this objective function, the typical optimization algorithm used is Stochastic Gradient Descent (SGD):

$$\theta \leftarrow \theta - \eta\nabla_\theta\tilde{\mathcal{L}}(\theta) \tag{5.22}$$

Here, $\tilde{\mathcal{L}}(\theta)$ represents the loss computed using a batch or subset of the data, facilitating efficient gradient computation and model updates.

5.5 Conclusions

As we have explored in this chapter, autoregressive models are powerful tools for modeling high-dimensional sequences. We began with classical approaches such as linear AR models, progressed to RNNs, and concluded with the groundbreaking Transformers that have reshaped the fields of NLP, speech, computer vision, and more. However, at the time of writing, Transformers for time-series applications have not yet reached the level of performance observed in other domains such as text and vision tasks.

Adapting Transformers for time series may require further optimization and empirical work to achieve the same level of success. Throughout this chapter, we have highlighted points that could provide pathways for improvement. By addressing the unique challenges posed by time-series data, future research and development can help bridge the current performance gap. Based on findings in the literature, simple customized solutions, such as solutions based on linear AR models, can still outperform them, particularly when forecasting financial time series. However, Transformer models offer many advantages, and their use depends on your main objective—whether it is performance, scalability for handling multiple heterogeneous time series, custom versus general solutions, or other factors.

When it comes to applications of Transformers for developing trading strategies, they are powerful tools for preprocessing alternative data, providing valuable insights for systematic strategies or discretionary investment managers. As a use case, in Chapter 9, we will demonstrate how to leverage this powerful model for processing alternative data and develop systematic trading strategies.