HIS PROJECT - TSA

Final Report

Author
Aneeq Ahmad
Alberto Mejia
Jay Asodariya
Jemish Moradiya

Contents

1 Chapter 01 Summary

1.1 Introduction

This book serves as an advanced guide for data scientists and ML engineers to deepen their skills in time series analysis, a crucial and often overlooked aspect of ML. The book aims to progress beyond traditional methods such as ARIMA to the latest ML techniques, addressing the growing complexity and volume of business-related time series data.

1.2 What is a Time Series?

A time series is a sequence of observations recorded over time. Examples include the number of chocolate bars consumed over a month or monthly weight measurements. Time series can reveal relationships and trends, such as the correlation between chocolate consumption and weight changes, and can extend to other domains like stock prices and climate measurements.

1.3 Types of Time Series

The book distinguishes between regular time series with fixed time intervals and irregular ones without such consistency. Regular time series examples include hourly or monthly data points, while irregular ones could be patient lab test readings that occur sporadically.

1.4 Applications of Time Series Analysis

Three main applications are discussed: forecasting future values, classifying time series data (e.g., normal vs. abnormal EKG readings), and deriving causal inferences from time series to understand dynamics and relationships.

1.5 Data-Generating Process (DGP)

The DGP refers to the underlying mechanism that produces a time series, which is often stochastic and complex. Accurate forecasting hinges on closely approximating this DGP with a mathematical model.

1.6 Synthetic Time Series Generation

The book delves into generating synthetic time series by combining fundamental components such as noise and signals to simulate real-world data complexity. Techniques for creating white and red noise processes, cyclical or seasonal signals, and autoregressive signals are explained.

1.7 Stationary vs. Non-Stationary Time Series

Stationary time series have constant mean and variance over time, while non-stationary series display changes in these statistical properties due to trends or varying variance (heteroscedasticity), which complicates their analysis.

1.8 Forecastability of Time Series

Forecastability is dependent on understanding the DGP, the amount of historical data, and the presence of repeating patterns. The book categorizes time series predictability and discusses factors that influence it, such as the high predictability of tides compared to the randomness of lottery numbers or the volatility of stock prices.

1.9 Forecasting Terminology

The terminology section introduces concepts like multivariate forecasting, explanatory forecasting, backtesting, and forecast combination. It explains the role of in-sample and out-sample data, as well as exogenous and endogenous variables in the context of time series forecasting.

1.10 Predicatbility

The predictability of a time series is influenced by several factors:

- **Understanding the Data-Generating Process (DGP):** A thorough understanding of the mechanisms that generate the time series data leads to better forecasting because the models can approximate the actual process more closely.

- **Amount of Data:** Access to a large volume of historical data improves predictability by revealing underlying trends, cycles, and variabilities.

- **Repeating Patterns:** Predictability is higher when the time series exhibits clear and consistent patterns that repeat over time.

- **Type of Data:** The inherent nature of the time series affects its predictability. Series influenced by well-understood phenomena (like tides) are more predictable than those with high randomness (like lottery numbers).

- **Statistical Characteristics:** The stationarity of a time series—constant mean, variance, and covariance—facilitates predictability. Non-stationary series are more challenging due to trends, seasonality, and other instabilities.

Each of these factors contributes to the overall predictability of a time series, with some series being inherently more predictable than others based on these characteristics.

1.11 Forecasting

2 Chapter 01 Code Analysis

```
ar_value = [self.previous_value[i] * self.ar_param[i] for i in
    range(len(self.ar_param))]
noise = np.random.normal(loc=0.0, scale=self.sigma, size=1)
ar_value = np.sum(ar_value) + noise
self.previous_value = self.previous_value[1:]+[ar_value[0]]
return ar_value
```

3 Chapter 02 Summary

Steps for Acquiring and Processing Time Series Data:

1. Understanding the time series dataset

2. Prepairing the data model

3. Handling missing data: Missing data can sometimes be informative, therefore it is important to take this into account in the data generation process

   (a) Last Observation Carried Forward or Forward Fill

   (b) Next Observation Carried Backward of Backward Fill

   (c) Mean Value Fill

   (d) Linear Interpolation

   (e) Nearest Interpolation

   (f) Spline, Polynomial, and Other Interpolations

4. Converting data into tiime series data

   (a) Time Series Identifiers

   (b) Metadata or Static Features

   (c) Time-Varying Features

   (d) Find the Global End Date

   (e) Basic Preprocessing

   (f) Mapping additional information

5. Compact, expanded, and wide forms of data

   (a) Wide (useless): We have the date as an index or as one of the columns and the different time series as different columns of the DataFrame. As the number of time series increases, they become wider and wider, hence the name.

   (b) Expanded: is when the time series is expanded along the rows of a DataFrame. If there are n steps in the time series, it occupies n rows in the DataFrame.

   (c) Compact: is when any particular time series occupies only a single row in the pandas DataFrame – that is, the time dimension is managed as an array within a DataFrame row.

6. Enforcing regular intervals in time series

7.

3.1 Data Preprocessing

3.2 Missing Data

4 Chapter 02 Code Analysis

1. Week of Year: df.date.dt.isocalendar().week.iloc[0] //error in reference to weekofyear

2. Why add 4 day to the date range in line 22?

3. Resampling is useful for aggregations and degragations

4. Difference between y1=df.ISE and y2=df['ISE']?

5.

This line should be skipped for the code to work

```
df = df.loc["2022-07-07␣7:00":"2022-07-07␣09:00", "pm2_5_1_hr"]
fig = px.line(df, x=df.index, y="pm2_5_1_hr", title="Missing␣
    Values␣in␣PM2.5")
fig = format_plot(fig, ["Original"])
fig.write_image("imgs/chapter_2/missing_values.png")
fig.show()
```

There is a code error referencing the parent directory

```
os.makedirs("imgs/chapter_2", exist_ok=True)
source_data = Path("scripts/data/london_smart_meters/") #
    scripts is added here
block_data_path = source_data/"hhblock_dataset/hhblock_dataset"
fig.show()
```

5 Chapter 03 Summary

Important topics discussed in chapter

1. Components of a time series

2. Visualizing time series data

3. Decomposing a time series

4. Detecting and treating outliers

5.1 Components of a Time Series

The following terms can be mixed in different ways, but two very commonly assumed ways are additive and multiplicative

1. Trend: is a long-term change in the mean of a time series. It is the smooth and steady movement of a time series in a particular direction.

2. Seasonal: When a time series exhibits regular, repetitive, up-and-down fluctuations.

3. Cyclical: is often confused with seasonality, but it stands apart due to a very subtle difference. Like seasonality, the cyclical component also exhibits a similar up-and-down pattern around the trend line, but instead of repeating the pattern every period, the cyclical component is irregular.

4. Irregular: is left after removing the trends, seasonality, and cyclicity from a time series. Traditionally, this component is considered unpredictable and is also called the residual or error term. (Not completely useless. Can sometimes be explained by an exogenous varibale to a certain extent)

5.2 Visualizing Time Series Data

1. Line Charts

2. Seasonal (box)Plots

3. Calendar Heatmaps

4. Autocorrelation plot

5.3 Decomposing a Time Series

5.3.1 Detrending

Here we estimate the trend component (which is the smooth change in the time series) and remove it from the time series, giving us a detrended time series.

Types:

1. Moving averages: It can be seen as a window that is moved along the time series in steps, and at each step, the average of all the values in the window is recorded.

2. Locally Estimated Scatterplot Smoothing (LOESS) Regression: We use an ordinal variable that moves between the time series as the independent variable and the time series signal as the dependent variable. For each value in the ordinal variable, the algorithm uses a fraction of the closest points and estimates a smoothed trend using only those points in a weighted regression. The weights in the weighted regression are the closest points to the point in question. This is given the highest weight and it decays as we move farther away from it. This gives us a very effective tool for modeling the smooth changes in the time series (trend).

5.3.2 Deseasonalizing

Here, we estimate the seasonality component from the detrended time series. After removing the seasonal component, what is left is the residual.

1. Moving averages: It can be seen as a window that is moved along the time series in steps, and at each step, the average of all the values in the window is recorded.

2. Fourier series: Any periodic function, no matter the shape, curve, or absence of it, or how wildly it oscillates around the axis, can be broken down into a series of sine and cosine waves.

5.4 Detecting and Treating Outliers

An outlier, as its name suggests, is an observation that lies at an abnormal distance from the rest of the observations. This can be for many reasons, including faulty measurement equipment, incorrect data entry, and black-swan events, to name a few. Being able to detect such outliers and treat them may help your forecasting model understand the data better.

Techniques:

1. Standard Deviation

2. Interquartile range

3. Isolation Forest

4. Extreme studentized deviate (ESD) and seasonal ESD (S-ESD)

5.5 Decompositon

5.6 Outliers

6 Chapter 03 Code Analysis

7 Chapter 04

7.1 Setting up a test harness

7.1.1 Creating holdout (test) and validation dataset

As a standard practice, in machine learning, we set aside two parts of the dataset, name them validation data and test data, and don't use them at all to train the model.

- **validation set:** is used in the modeling process to assess the quality of the model. To select between different model classes, tune the hyperparameters, perform feature selection, and so on, we need a dataset.
- **holdout (test) set:** is like the final test of your chosen model. It tells you how well your model is doing in unseen data.

The best practice is to use the most recent part of the dataset as the test data. Additionally, it is advisable to have validation and test datasets of equal size to ensure that the decisions made during the modeling process, based on the validation data, are as applicable as possible to the test data.

7.1.2 Chossing an evluation metric

In time series forecasting realm, there are scores of metrics with no real consensus on which ones to use. One of the reasons for this overwhelming number of metrics is that no one metric measures every characteristic of a forecast.

- **Mean Absolute Error (MAE):**
- **Mean Squared Error (MSE):**
- **Mean Absolute Scaled Error (MASE):**
- **Forecast Bias (FB):**

7.2 Generating strong baseline forecasts

Time series forecasting has been around since the early 1920s, and through the years, many brilliant people have come up with different models, some statistical and some heuristic-based.

Refered to as:

- Naïve forecast

- Moving average forecast
- Seasonal naive forecast
- Exponential smoothing (ETS)
- Simple exponential smoothing (SES)
- Double exponential smoothing (DES)
- Triple exponential smoothing or Holt -Winters (HW)
- The Autoregressive Integrated Moving Average (ARIMA)
- Theta Forecast
- Fast Fourier Transform forecast

After performing the aformationed forecasting techniques it is important to remember that a comparison of their performance should be made. Not only with respect to the forecast bias of each technique but also the time elapsed to perform the techniques. When the 2-3 top candidates have finally been chosen, the forecasting algorithm can now be used on the validation and test data to assess which is the most adequate.

7.3 Assessing the forecastability of a time series

7.4 Setting a Strong Baseline Forecast

8 Chapter 04 Code Analysis

9 Chapter 05 Summary

9.1 Understanding the basics of machine learning

9.1.1 Supervised machine learning tasks

Machine learning can be used to solve a wide variety of tasks such as regression, classification, and recommendation. But, since classification and regression are the most popular classes of problems, we will spend just a little bit of time reviewing what they are.

9.1.2 Overfitting and underfitting

to machine learning models as well when they don't learn enough patterns, and this is called underfitting. Overfitting is an undesirable machine learning behavior that occurs when the machine learning model gives accurate predictions for training data but not for new data.

9.1.3 Hyperparameters and validation sets

Hyperparameters are parameters of the model that are not learned from data but rather are set before the start of training. For instance, the weight of the regularization is a hyperparameter.

9.2 Time series forecasting as regression

Time series forecasting, by definition, is an extrapolation problem, whereas regression, most of the time, is an interpolation one. Extrapolation is typically harder to solve using data-driven methods. Another key assumption in regression problems is that the samples used for training are **independent and identically distributed (IID).**. Thankfully, there are ways to convert a time series into a regression and get over the IID assumption by introducing some memory to the machine learning model through some features.

9.2.1 Time delay embedding

In time delay embedding, we assume a window of arbitrary length M ¡ L and extract fixed-length subsequences from the time series by sliding the window over the length of the time series.

9.2.2 Temporal embedding

There are many ways to do this, from simply aligning a monotonically and uniformly increasing numerical column that captures the passage of time to sophisticated Fourier terms to capture the periodic components in time.

9.3 Local versus global models

We can consider that all the time series in a related time series come from separate data generating processes (DGPs), and thereby model them all separately. We call these the local models of forecasting. An alternative to this approach is to assume that all the time series are coming from a single DGP. Instead of fitting a separate forecast function for each time series individually, we fit a single forecast function to all the related time series. This approach has been called global or cross- learning in literature.

10 Chapter 05 Code Analysis

11 Chapter 06 Summary

11.1 Feature engineering

Feature engineering, as the name suggests, is the process of engineering features from the data, mostly using domain knowledge, to make the learning process smoother and more efficient. Two main ideas to encode time into the regression framework: time delay embedding and temporal embedding.

11.2 Avoiding data leakage

Data leakage occurs when the model is trained with some information that would not be available at the time of prediction. Typically, this leads to high performance in the training set, but very poor performance in unseen data. There are two types of data leakage:

- **Target leakage** is when the information about the target (that we are trying to predict) leaks into some of the features in the model
- **Train-test contamination** is when there is some information leaking between the train and test datasets.

11.3 Setting a forecast horizon

Forecast horizon is the number of time steps into the future we want to forecast at any point in time.

11.4 Time delay embedding

Time delay embedding is to embed time in terms of recent observations.

- **Lags or backshift**
- **Rolling window aggregations**
- **Seasonal rolling window aggregations**
- **Exponentially weighted moving averages (EWMA)**

11.5 Temporal embedding

Temporal embedding as a process where we try to embed time into features that the ML model can leverage.

- **Calendar features**
- **Time elapsed**
- **Fourier terms**

12 Chapter 06 Code Analysis

13 Chapter 07 Summary

13.1 Handling non-stationarity in time series

Stationarity in econometrics assumes constant statistical properties over time. In time series and regression, this matters as we estimate a single function. For example, the number of park visitors pre- and post-pandemic illustrates concept drift, a phenomenon recognized in machine learning when a model's relevance shifts over time.

There are four main questions we can ask ourselves to check whether our time series is stationary or not:

- Does the mean change over time? Or in other words, is there a trend in the time series?

- Does the variance change over time? Or in other words, is the time series heteroscedastic?

- Does the time series exhibit periodic changes in mean? Or in other words, is there seasonality in the time series?

-

- Does the time series have a unit root?

13.2 Detecting and correcting for unit roots

Time series analysis has its roots in econometrics and statistics and unit root is a concept derived directly from those fields.

- **Unit roots**

- **The Augmented Dickey-Fuller (ADF) test**

- **Differencing transform**

13.3 Detecting and correcting for trends

In Chapter 5, we discussed time series forecasting as a challenging extrapolation problem, particularly due to trends. ARIMA and exponential smoothing address this by autoregression and explicit trend modeling. Standard regression may struggle with extrapolation, but with suitable features like lags, it becomes more adept. Detrending simplifies regression application by focusing on trend-removed data.

- **Deterministic and stochastic trends**

- **Kendall's Tau** is a measure of correlation but carried out on the ranks of the data. Similar to Spearman's correlation, which also calculates correlation on ranked data, Kendall's Tau is a non-parametric test and therefore does not make assumptions about the data.
- **Mann-Kendall test (M-K test)** is used to check for the presence of a monotonic upward or downward trend. And since the M-K test is a non-parametric test, like Kendall's Tau
- **Detrending transform**

13.4 Detecting and correcting for seasonality

- **Detecting seasonality** autocorrelation function (ACF)
- **Deseasonalizing transform**

13.5 Detecting and correcting for heteroscedasticity

- **Detecting heteroscedasticity**
- **Log transform**
- **Box-Cox transform**

13.6 AutoML approach to target transformation

14 Chapter 07 Code Analysis

Steps:

- Preprocessed this file from chapter 6 is must 01-Feature Engineering.ipynb

Errors:

- need to install joblib library
- need to install darts library for transformation models
- File Path error on each file path line

15 Matlab for Time Series Analysis

Example I have tried to run and understand on matlab

- Find Events in Timetable Using Event Table
- Time Series Objects and Collections

16 Chapter 09 Summary

16.1 Combining forecasts

16.2 Best fit

This strategy of choosing the best forecast is by far the most popular and is as simple as choosing the best forecast for each time series based on the validation metrics. This strategy has been made popular by many automated forecasting software, which calls this the "best fit" forecast. The algorithm is very simple:

- Find the best-performing forecast for each time series using a validation dataset.
- For each time series, select the forecast from the same model for the test dataset.

16.3 Measures of central tendency

16.4 Simple hill climbing

It is the simplest way to implement a hill climbing algorithm. It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state.

Few limitations of hill climbing

- Runtime considerations
- Short-sightedness
- Forward-only

16.5 Stochastic hill climbing

The key difference between simple hill climbing and stochastic hill climbing is in the evaluation of candidates. In a simple hill-climb, we evaluate all possible options and pick the best among them. However, in a stochastic hill-climb, we randomly pick a candidate and add it to the solution if it is better than the current solution. This addition of stochasticity helps the optimization not get the local maxima/ minima.

16.6 Simulated annealing

It is inspired by a physical phenomenon – annealing solids. Annealing is the process of heating a solid to a predetermined temperature (usually above its melting point), holding it for a while, and then slowly cooling it. This is done to ensure that the atoms assume a new globally minimum energy state, which induces desirable properties to some metals, such as iron.

16.7 Optimal weighted ensemble

16.8 Stacking or blending

We train another learning algorithm on the predictions of some base learners to combine these predictions. This second-level model is often called a stacked model or a meta model. also known as **stacked generalization**

Stacking is when the meta model is trained on the entire training dataset, but with out-of-sample predictions.

Blending is similar to this but slightly different in the way we generate out-of-sample predictions.

17 Chapter 09 Code Analysis

18 Chapter 10 Summary

18.1 Why Global Forecasting Models (GFMs)?

The need for Global Forecasting Models (GFMs) in forecasting related time series datasets, highlighting shared attributes. Unlike Local Forecasting Models (LFMs), which treat each series independently, GFMs consider all series as originating from a single data generating process, offering more accurate and efficient forecasting. GFMs address drawbacks of LFMs, providing a holistic approach for improved modeling.

- **Sample Size:**Stresses the importance of a large dataset for effective GFM training, ensuring a diverse and representative range of information.
- **Cross-learning:**Utilizes insights from one time series to improve forecasting for another, leveraging interdependencies between related series.
- **Multi-task learning:**Trains the GFM to handle multiple forecasting tasks simultaneously, capturing common patterns across different time series.
- **Engineering complexity:**Integrates complex features and relationships into the GFM for more nuanced and accurate forecasts.

18.2 Creating GFMs

Training Global Forecasting Models (GFMs) is straightforward by consolidating related time series into a single dataframe and training a unified model. It's crucial to ensure that all time series in the dataset have the same frequency to maintain performance. The standard framework from Chapter 8, used for Local Forecasting Models, can be adapted for GFMs. Specific adjustments, such as defining FeatureConfig and MissingValueConfig, are made in the notebook to accommodate the global modeling approach for all households in the London Smart Meters dataset.

18.3 Strategies to improve GFMs

- **Increasing memory:**Enhances GFM performance by increasing memory capacity, allowing the model to capture longer-term dependencies in the time series
  - **Adding more lag features:**Improves forecasting accuracy by incorporating lagged values of the target variable, capturing historical patterns.
  - **Adding rolling features:**Boosts GFM's capability by introducing rolling statistics (e.g., rolling mean) to capture trends and changes in the time series over time.
  - **Adding EWMA(Exponentially Weighted Moving Average) features:**Introduces Exponentially Weighted Moving Average features, emphasizing recent observations in the time series for improved sensitivity to changes.
- **Using time series meta-features:**Incorporates meta-features that provide additional information about the time series, enhancing the model's understanding of underlying patterns.
  - **Ordinal encoding and one-hot encoding:**Transforms categorical variables into numerical representations (ordinal encoding) or binary vectors (one-hot encoding) for better model compatibility.
  - **Frequency encoding:**Encodes categorical variables based on their frequency of occurrence, providing the model with information about the distribution of categories.
  - **Target mean encoding:**Utilizes the mean of the target variable for different categories, enriching the model with information about the relationship between categorical features and the target.
- **Tuning hyperparameters:**Optimizes GFM performance by fine-tuning hyperparameters through methods such as grid search, random search, or Bayesian optimization.
  - **Grid search**
  - **Random search**
  - **Bayesian Optimization**
- **Partitioning:**Divides the dataset into subsets for training and validation purposes, enhancing model robustness.
  - **Random partitioning:**Splits the data randomly into training and validation sets.
  - **Judgemental partitioning:**Divides the data based on domain knowledge or expert judgment.
  - **Algorithmic partitioning:**Uses algorithmic methods to partition the data based on characteristics or patterns within the time series.

18.4 Bonus – interpretability

Interpretability in the context of machine learning and artificial intelligence, defining it as the degree to which a human can understand the cause of a decision. Two approaches to interpretability are transparency, where the model is inherently simple and understandable, and post hoc interpretation, which involves techniques to understand model predictions. Post hoc techniques like permutation feature importance, Shapley values, and LIME are discussed as methods applicable to any machine learning model, including Global Forecasting Models (GFMs).

19 Chapter 10 Code Analysis

20 Chapter 11 Summary

20.1 What is deep learning and why now?

- **Increase in compute availability**
- **Increase in data availability**

20.2 Perceptron - The first neural network

The Perceptron, developed by Frank Rosenblatt in the 1950s, is one of the first neural network models. It functions by taking weighted inputs, summing them, and then outputting a binary result based on a threshold. This model mimics the basic function of a biological neuron but is much simpler. Despite its simplicity and limitations (such as handling only linearly separable problems), the Perceptron laid the groundwork for more complex neural networks.

- **Inputs:** These are the real-valued inputs that are fed to a Perceptron. This is like the dendrites in neurons that collect the input.
- **Weighted sum:** Each input is multiplied by a corresponding weight and summed up. The weights determine the importance of each input in determining the outcome.
- **Non-Linearity:** The weighted sum goes through a non-linear function.Real-world data is often complex and not linearly separable. A nonlinear function allows the neural network to capture and model these nonlinear relationships in the data.

20.3 Components of a deep learning system

Deep learning is presented as a highly modular system, characterized by two key properties of its parametrized modules:

- **Output production:** These modules can generate an output from a given input through a series of computational steps.
- **Feedback adjustment:** When provided with the desired output, the modules can communicate back to their inputs, indicating how they should change to more closely align with this desired outcome. This feedback mechanism helps in iteratively adjusting the inputs to reduce the difference between the actual and desired outputs.

The core of this concept is the use of differentiable functions, which is crucial for gradient-based optimization methods commonly employed in deep learning. This differentiability allows for efficient computation of gradients, which are used to update the parameters of the model in a direction that minimizes the error or loss. This makes deep learning models adaptable and effective for a wide range of complex tasks.

20.4 Linear layers and activation functions

Learns the best features by which we can make the problem linearly separable. Linearly separable means when we can separate the different classes (in a classification problem) with a straight line.

20.4.1 Linear Transformations (In hidden layers)

- **Purpose:** Linear transformations in neural networks are typically accomplished through operations like weighted sums. They are used to project input data into a different space.
- **Function:** A linear transformation combines input features in a linear way (e.g., summing up inputs each multiplied by their respective weights). This process is essential for feature combination and transformation.
- **Implementation and Impact:** Linear transformations alone cannot capture complex patterns in data, especially if the data is not linearly separable. They are limited to linear relationships.

20.4.2 Intermediate Activation Functions (In hidden layers)

- **Purpose:** Activation functions introduce nonlinearity into the network. They help the network learn and represent more complex relationships in the data.
- **Function:** After the linear transformation, the activation function processes the result to produce a nonlinear output. Common activation functions include ReLU (Rectified Linear Unit), sigmoid, and tanh.
- **Implementation and Impact:** Without activation functions, a neural network, regardless of how many layers it has, would essentially be a linear

regression model, unable to solve complex problems like image recognition or natural language processing.

Linear transformations handle the combination and re-scaling of input features, while activation functions introduce the necessary nonlinearity that allows the network to model complex patterns. A typical layer in a neural network sequentially applies a linear transformation followed by a nonlinear activation function. This combination enables the network to learn from and make predictions on complex, real-world data.

20.4.3 Output Activation Functions (In output layer)

- **Purpose:** The output activation function is critical in adapting the final layer's output of a neural network to the format required for a specific task, such as regression, binary classification, or multiclass classification. It ensures that the network's output is interpretable and suitable for the problem at hand.

- **Function:** This function processes the output of the network's final linear transformation to produce a result that aligns with the nature of the target variable. For instance, in a binary classification task, it converts the final layer's output into a probability (between 0 and 1), while in a regression task, it might leave the output as is (linear activation) or apply a transformation to ensure the output falls within a certain range.

- **Implementation and Impact:** The selection and application of an output activation function have profound implications on the training dynamics, model accuracy, and result interpretability. It needs to be compatible with the loss function used, directly influencing how the model's performance is evaluated and optimized during training. This choice also determines how the final output is presented, making it crucial for ensuring the model's outputs are relevant and comprehensible for the given task.

20.4.4 Loss function

- **Function:** A loss function, also known as a cost function, quantitatively measures how far the predictions of a neural network are from the actual target values. In mathematical terms, it calculates the error or difference between the predicted output and the true output. The specific form of a loss function can vary depending on the type of task. For example, Mean Squared Error (MSE) is common in regression tasks, while Cross-Entropy loss is often used in classification tasks.

- **Purpose:** The primary purpose of a loss function is to guide the training process of the neural network. By providing a quantifiable metric of how well the model is performing, the loss function becomes the objective that the training process aims to minimize. It serves as a feedback mechanism for the model, indicating how far off its predictions are, and thus how it needs to adjust its weights during training.

- **Implementation and Impact:** During training, the neural network uses an optimization algorithm, typically some form of gradient descent, to adjust its weights in a way that minimizes the loss function. The loss function must be compatible with the output activation function of the neural network. For instance, using a softmax activation in the output layer typically pairs with a cross-entropy loss function.

20.5 Gradient descent

20.5.1 Forward Computation (Forward Propagation)

- This involves passing input data through the network, layer by layer, using the computations defined in each layer, to produce an output.
- The output is then compared to the desired output using a loss function to determine how close the model's prediction is to the actual target.

20.5.2 Backward Computation (Backward Propagation)

- Once the output is obtained and the loss is calculated, this step involves computing the gradient of the loss function with respect to all the parameters in the network.
- The gradient, a generalization of derivatives for multivariate functions, indicates the rate of change of the loss with respect to changes in the network's parameters.

20.5.3 Gradient and its Importance

- Gradients, akin to slopes in high school mathematics, inform us about the local slope of a function.
- By understanding the gradient of the loss function, gradient descent, a mathematical optimization technique, can be employed to adjust the network's parameters in a way that minimizes the loss, thus optimizing the model's performance.

21 Chapter 11 Code Analysis

22 Chapter 12 Summary

22.1 Understanding the encoder-decoder paradigm

The encoder-decoder algorithm is a fundamental architecture in deep learning, often used for sequence-to-sequence tasks. Its main purpose is to take a variable-length input sequence, encode it into a fixed-length representation (the encoder part), and then decode this representation into a variable-length output sequence (the decoder part). In other words, it's used for tasks where you have an input sequence and want to generate a corresponding output sequence. The encoder-decoder architecture allows deep learning models to handle input and output sequences of different lengths, making it powerful for tasks involving sequential data. It's particularly useful for tasks where the relationship between input and output is complex and not one-to-one.

- **Input/Feature Space:** The input space, also known as the feature space, is where your original data resides. In the context of the encoder-decoder algorithm, the input space typically represents the data you're trying to process or transform.
- **Encoder:** The encoder takes the data from the input space and processes it layer by layer through neural networks.Each layer extracts increasingly abstract and meaningful features from the input data.
- **Latent Space:** The output of the encoder is the latent representation in the latent space. This latent representation is a compressed, abstract, and semantically meaningful representation of the input data. It captures the essential information required for generating an appropriate output.
- **Decoder:** Takes this latent representation from the encoder and transforms it back into a form that corresponds to the output space. The decoder leverages the information encoded in the latent space to generate an appropriate output.
- **Loss:** Is a measure of how far the prediction is from the desired output. The goal in training the model is to minimize this loss. This training process ensures that the decoder learns to extract relevant information from the latent space and use it to generate accurate outputs.

22.2 Feed-forward networks

A Feed-forward network (FFN) or fully connected network is one of the simplest neural network architectures. It takes a fixed-size input vector and processes it through a series of layers to produce the desired output. The term "feed-forward" signifies that data flows in one direction through the network, and "fully connected" means that each unit in a layer is connected to every unit in the previous and subsequent layers.

- **Input and Output Layers:** The first layer is called the input layer, and its size matches the dimension of the input data. The final layer is the output layer, which is determined by the desired output format. For example, if you need one output, you have one unit in the output layer; if you need ten outputs, you have ten units.

- **Hidden Layers:** All the layers between the input and output layers are known as hidden layers. These layers contain units that perform computations and contribute to the network's ability to learn complex patterns.
- **Network Structure:** The structure of the FFN is defined by two key hyperparameters: the number of hidden layers and the number of units in each layer. These hyperparameters determine the network's capacity to represent and learn from the data.

In the context of time series forecasting, an FFN can serve as both an encoder and a decoder. As an encoder, it can transform time series data, making it suitable for regression tasks. As a decoder, it operates on the latent vector, generated by the encoder, to produce the desired output. This decoder role is the most common use of FFNs in time series forecasting.

22.3 Recurrent neural networks

RNNs are a class of neural networks specifically designed for processing sequential data. They are particularly well-suited for tasks where the order and context of the data matter. Unlike feed-forward networks (FFNs), RNNs have connections that loop back on themselves, allowing them to maintain a hidden state, or memory, that captures information from previous timesteps.

Key Components of an RNN:

- **Hidden State (Memory):** The core component of an RNN is the hidden state, often referred to as the "memory." It's a vector that stores information about the data seen in previous timesteps. This hidden state is updated at each timestep based on the current input and the previous hidden state.
- **Input:** At each timestep, the RNN receives an input, which could be a single element of a sequence or a vector representing the entire input at that timestep. The input is used to update the hidden state.

In essence, Recurrent Neural Networks (RNNs) can be conceptualized as comprising two concurrent feed-forward networks (FFNs). The first FFN operates on the input data, mapping it to a specific output. Simultaneously, the second FFN is applied to the initial hidden state, aiming to transform it into a specific hidden state output. As a result, RNN blocks can be organized to accommodate a diverse range of input and output scenarios, including many-to-one setups, as well as many-to-many configurations. That being said, the main drawback of RNNs in sequence modeling is the potential for vanishing or exploding gradients during backpropagation, which occurs as the network struggles with lengthy sequences, leading to either halted learning or unstable training. We can analogize this phenomenon to the process of repeatedly multiplying a scalar number by itself: If the number is less than one, it gradually diminishes with each iteration, approaching zero; conversely, if the number exceeds one, it rapidly grows exponentially.

22.4 Long short-term memory (LSTM) networks

LSTM was designed to address the challenges of vanishing and exploding gradients encountered in vanilla RNNs. The core concept behind LSTM is inspired by computer logic gates, and it introduces a crucial element known as a "memory cell" to facilitate long-term memory retention alongside the traditional hidden state memory of RNNs.

- **Input Gate:** This gate determines how much information to read from the current input and the previous hidden state.
- **Forget Gate:** The forget gate's role is to decide how much information should be discarded from the long-term memory.
- **Output Gate:** The output gate determines the extent to which the current cell state contributes to creating the current hidden state, which serves as the output of the cell.
- **Memory Cell:** It can be thought of as a container that holds information over multiple timesteps. It has the capability to store and update information for long-term dependencies, and its content is modified by the input and output gates.

So, while both RNNs and LSTMs have elements that can be seen as forms of memory, the memory cell in LSTMs is a more specialized and flexible mechanism explicitly designed to address the limitations of traditional RNNs in capturing and retaining long-term information in sequential data.

22.5 Gated recurrent unit (GRU)

The idea behind GRU is akin to using gates to control the flow of information within the network. However, GRU distinguishes itself by eliminating the long-term memory cell component found in LSTMs. Instead, it relies solely on the hidden state to convey and manage information. In essence, the hidden state itself acts as the "gradient highway," enabling information to pass through the network efficiently. Unlike LSTMs, GRUs use two key gates: the reset gate, which determines how much of the previous hidden state is considered when computing the candidate's hidden state for the current timestep, and the update gate, which governs how much of the previous hidden state is carried forward and how much of the current candidate's hidden state contributes to the new hidden state.

In relation to the previously mentioned concepts, GRUs are another variation of recurrent neural networks designed to address the challenges faced by traditional RNNs. Similar to LSTMs, GRUs introduce gating mechanisms to control information flow, but they simplify the architecture by eliminating the dedicated memory cell. Instead, GRUs rely on the hidden state as the primary carrier of information, making them computationally more efficient and suitable for capturing dependencies in sequential data.

22.6 Convolution networks

At the core of CNNs lies the mathematical operation of convolution, which plays a central role in data processing. In the context of image data, CNNs employ kernels, which are 2D matrices with numerical values. These kernels are smaller than the images they operate on, and they are ”slid” across the image, performing element-wise multiplications and summations at each position. This process allows CNNs to extract features from the data effectively.

For time series data, CNNs transition to 1D convolutions, where a 1D kernel is similarly moved along the sequence to produce an output. Unlike manually engineered features, CNNs learn the kernel weights from the data, making them adaptable and capable of generating various features. By adjusting kernel weights, CNNs can capture different characteristics in the data, enhancing their capacity as feature generators.

1D Convolution for Time Series Data:

- **Kernel Movement:** Similar to image data, 1D kernels are moved along the sequence of data points. At each position, the kernel interacts with a segment of the time series data, performing element-wise operations and producing an output value.

- **Learnable Weights:** Just like with image data, the weights within the 1D kernels are learned from the time series data during training. This adaptability enables CNNs to discover meaningful patterns or features in the sequence.

- **Feature Generation:** By adjusting the weights within the kernels, CNNs can capture different characteristics within the time series. For example, specific weights may emphasize short-term trends, while others may focus on long-term patterns. This flexibility makes CNNs powerful feature generators for time series analysis.

- **Adaptive Learning:** It’s crucial to note that the values within these kernels are not fixed but are learned by the CNN during training. This adaptability allows CNNs to adjust their feature extraction behavior based on the specific dataset and task.

- **Stride:** refers to the step size at which a convolutional kernel moves across the input data during the convolution operation. It determines how much the kernel ”jumps” or shifts between each operation.

- **Dilation:** Adjusts the gaps or ”holes” between the kernel values during convolution, expanding the receptive field without changing the kernel size, enabling the layer to capture more input information.

- **Receptive Field:** The receptive field of a convolutional layer is the area in the input data that influences the feature generated by that layer. It’s essentially the spatial extent or size of the input window over which the convolution operation is performed.

It it important to note that in multi-layered Convolutional Neural Networks (CNNs), calculating the receptive field becomes more complex due to the hierarchical structure of the network. In multi-layered CNNs, each layer processes features extracted by previous layers. As a result, the receptive field of deeper layers encompasses larger portions of the input data, allowing them to capture more context and complex patterns.

23 Chapter 12 Code Analysis

24 Chapter 13 Summary

24.1 Tabular regression

Tabular regression in time series forecasting involves using structured data arranged in rows and columns to predict future values. This approach applies regression models, such as linear regression or decision trees, to learn patterns from historical data, considering features like past values and external factors. The process includes data preparation, feature engineering, model training, evaluation, and making predictions. The effectiveness depends on the chosen algorithm and the characteristics of the time series data.

24.2 Single-step-ahead recurrent neural networks

Single-step-ahead recurrent neural networks (RNNs) are designed for time series forecasting, predicting the next value in a sequence based on previous values. They use recurrent connections to maintain a memory of past inputs. During training, the model learns patterns in historical data, and after training, it can predict the next value given a sequence of past values. Advanced variants like LSTM and GRU address some limitations of basic RNNs.

24.3 Sequence-to-sequence models

Sequence-to-sequence (Seq2Seq) models are neural networks designed for tasks like language translation and summarization. They consist of an encoder to process input sequences and a decoder to generate output sequences. Attention mechanisms enhance their ability to handle varying-length sequences, making them valuable for tasks with diverse structures.

24.3.1 RNN-to-fully connected network

Combining an RNN with a fully connected network involves using an RNN to capture sequential patterns and connecting its output to a fully connected layer for further processing or classification. This hybrid architecture is useful for tasks where both sequential and global information are relevant.

24.3.2 RNN-to-RNN

Connecting one RNN to another, often termed ”stacking” or ”chaining” RNNs, allows for capturing more complex dependencies in sequential data. This is common in tasks where hierarchical or long-range dependencies are crucial. Each RNN layer processes information sequentially before passing it to the next layer.

24.4 Summary

In the previous chapter, we applied basic deep learning concepts using PyTorch for common modeling patterns. We explored RNN, LSTM, and GRU for time series prediction, then delved into Seq2Seq models, combining encoders and decoders. Encoders and decoders can be complex, even mixing convolution and LSTM blocks. We introduced ”teacher forcing” for faster training and improved performance. In the next chapter, we'll dive into the popular and attention-grabbing topic of attention and transformers.

25 Chapter 13 Code Analysis

26 Chapter 14 Summary

26.1 Convolutional Neural Network (CNN)

CNNs play a crucial role in various applications, particularly in computer vision, due to their ability to automatically learn hierarchical features from input data.

- **Feature Extraction** automatically extracting hierarchical features from input data. Through a series of convolutional and pooling layers, they learn to recognize low-level features like edges and textures, which progressively combine to form more complex and abstract features[?].

- **Spatial Hierarchies** helps in building a robust representation of the input.

- **Translation Invariance** meaning they can recognize patterns regardless of their position in the input.

- **Transfer Learning** involves taking a pre-trained CNN and fine-tuning it on a specific dataset, often resulting in improved performance with less training data.

- **Texture and Pattern Recognition** recognizing textures and patterns in data, making them valuable in tasks where local patterns and structures are essential, such as medical imaging and satellite image analysis[?].

- **Reduced Parameter Sharing** significantly reduces the number of parameters compared to fully connected networks. This parameter efficiency makes CNNs computationally feasible for large-scale image datasets.

26.2 Combinations

Main ways to combine a convolutional neural network (CNN) and a long short-term memory (LSTM) network:

- Use the output of the CNN as the input to the LSTM. This allows the LSTM to learn features from the input data that have been learned by the CNN.
- Use the output of the LSTM as the input to the CNN. This allows the CNN to learn features from the output of the LSTM.
- Use a parallel architecture, where the CNN and LSTM operate on the input data independently, and their outputs are concatenated and passed to the fully connected layer.

26.3 Classification and forecasting solutions

26.3.1 Classification

- **Image Classification** learn hierarchical features from images and can automatically extract patterns and shapes. Architectures like AlexNet, VGGNet, and ResNet have been successful in various image classification challenges[**?**].
- **Transfer Learning for Classification** often leads to improved performance, especially when labeled data is limited.
- **Spatial Hierarchies in Time Series Data** can be adapted for classification. Each time step becomes a pixel in the "image," and the CNN learns spatial hierarchies of features over time.

26.3.2 Forecasting

- **1D CNNs for Time Series** can be used for time series data. The operation is applied along the temporal axis, capturing local patterns and dependencies.
- **Temporal Convolution** learns filters that can identify relevant features over different time scales, making them suitable for forecasting tasks.
- **Hybrid Models** combining CNNs with recurrent neural networks (RNNs) or long short-term memory networks (LSTMs) are increasingly used for time series forecasting. CNNs can capture local patterns, while RNNs or LSTMs can model temporal dependencies.[**?**]
- **Multivariate Time Series** network can learn patterns and relationships between different variables, enhancing forecasting capabilities.
- **Feature Learning** powerful feature extractors, automatically learn relevant features from the input data, reducing the need for manual feature engineering.

26.4 What is attention?

In deep learning, attention is a mechanism inspired by human cognitive function. It allows neural networks to dynamically focus on specific parts of input sequences, emphasizing relevant information while de-emphasizing less important parts. This attention mechanism is particularly useful in tasks involving sequences, such as natural language processing and time series analysis. It enables models to weigh different parts of the input sequence differently, enhancing their ability to capture complex relationships and dependencies.

26.5 Generalized attention model

A generalized attention model is a flexible extension of the traditional attention mechanism in deep learning. It allows for customization in how attention is computed and applied, providing adaptability for different tasks and data characteristics. It enables neural networks to dynamically focus on specific parts of input sequences, enhancing their ability to capture complex relationships and dependencies.

26.5.1 Alignment functions

There are many variations of the alignment function that have come up over the years. Let's review a few popular ones that are used today.

- **Dot product:** Simplest alignment function. Measures similarity by taking the dot product of query and key vectors. Requires the same dimensions for query and key vectors.
- **Scaled dot product attention:** Variation of the dot product with scaling to control the magnitude of the gradients. Mitigates issues with vanishing or exploding gradients.
- **General attention:** Allows for different weight matrices for query and key vectors. Adds flexibility compared to dot product attention.
- **Additive/concat attention:** Combines query and key vectors through concatenation followed by a trainable matrix. Introduces non-linearity, enhancing expressive power. Offers another level of flexibility in modeling attention.

26.5.2 Distribution function

The distribution function's main purpose is to convert learned scores from the alignment function into weights that sum up to 1. The widely used softmax function accomplishes this, interpreting weights as probabilities. However, softmax has drawbacks, as it assigns some weight to every element, lacking sparsity. Alternatives like sparsemax and entmax address this by allowing probability mass on select relevant elements and assigning zero to others. These functions are useful when knowledge encoding involves only a few timesteps, enhancing interpretability and avoiding implausible options.

26.6 Forecasting with sequence-to-sequence models and attention

Forecasting with Seq2Seq models and attention involves using neural networks designed for sequential data. The encoder processes historical data, and attention helps the decoder focus on specific information during sequence generation. This approach improves accuracy by capturing complex patterns and dependencies, making it valuable for various forecasting tasks like stock prices or weather predictions.

26.7 Transformers – Attention is all you need

"Attention Is All You Need" (2017) introduced Transformers, a revolutionary architecture using scaled dot product attention, discarding recurrent networks for more efficiency in processing long sequences. Transformers, highlighted by BERT in 2018, dominate NLP, time series forecasting, reinforcement learning, and computer vision tasks. The original 2017 Transformer architecture remains foundational amid numerous adaptations.

26.7.1 Key component of Transformer model

- **Self-attention:** Allows each word in a sequence to focus on other words, capturing relationships and dependencies efficiently.
- **Multi-headed attention:** Utilizes multiple self-attention mechanisms in parallel, providing the model with diverse perspectives on relationships within the sequence.
- **Positional encoding:** Incorporates position information into the input embedding, addressing the model's lack of inherent understanding of sequence order.
- **Position-wise feed-forward layer:** Applies a feed-forward network independently to each position in the sequence, enhancing the model's ability to capture different features.

26.7.2 How the component put together

- **Encoder:** The vanilla Transformer model is an encoder-decoder model. There are N blocks of encoders, and each block contains an MHA layer and a position-wise feed-forward layer with residual connections in between.

  Residual connections: Introduces shortcut connections that help mitigate the vanishing gradient problem during training.

Layer normalization: Normalizes input to each layer, stabilizing training by reducing internal covariate shift.

- **Decoder:** The decoder block is also very similar to the encoder block, but with one key addition. Instead of a single self-attention layer, the decoder block has a self-attention layer, which operates on the decoder input, and an encoder-decoder attention layer.

  Masked self-attention: Allows each position to attend to all positions in the decoder up to and including that position, preventing information leakage from future positions.

26.8 Forecasting with Transformers

Forecasting with Transformers involves using the Transformer architecture for time series prediction. The self-attention mechanism enables the model to capture complex dependencies, and positional encoding addresses sequence order. Trained on historical data, Transformers excel in handling long-range dependencies, making them effective for various forecasting tasks, including stock prices and weather predictions.

26.9 Understanding the encoder-decoder paradigm

Deep learning, including the encoder-decoder paradigm, is employed in time series analysis to learn patterns from historical data for future predictions. In 1997, Ramon Neco and Mikel Forcada proposed an architecture for machine translation that had ideas reminiscent of the encoder-decoder paradigm. In 2013, Nal Kalchbrenner and Phil Blunsom proposed an encoder-decoder model for machine translation, although they did not call it that. The encoder-decoder architecture, pioneered around 2014 by Cho et al. and Sutskever et al., enables modeling variable-length inputs and outputs, facilitating end-to-end learning for tasks like machine translation.

The feature space, or the input space, is the vector space where your data resides. If the data has 10 dimensions, then the input space is the 10-dimensional vector space. Latent space is an abstract vector space that encodes a meaningful internal representation of the feature space. To understand this, we can think about how we, as humans, recognize a tiger. We do not remember every minute detail of a tiger; we just have a general idea of what a tiger looks like and its prominent features, such as its stripes. It is a compressed understanding of this concept that helps our brain process and recognize a tiger faster.

Now that we have an idea about latent spaces, let's see what an encoder-decoder architecture does. An encoder-decoder architecture has two main parts: an encoder and a decoder.

- **Encoder:** The encoder takes in the input vector, x, and encodes it into a latent space. This encoded representation is called the latent vector, z.

- **Decoder:** The decoder takes in the latent vector, z, and decodes it into the kind of output we need $(y\hat{y})$.

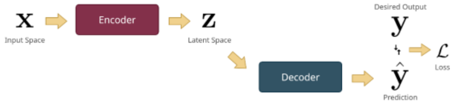


Figure 12.1 – The encoder-decoder architecture

In the context of time series forecasting, the encoder consumes the history and retains the information that is required for the decoder to generate the forecast. As we learned previously, time series forecasting can be written as follows:

$$y_t = h(y_{t-1}, y_{t-2}, \cdots, y_{t-N})$$

Now, using the encoder-decoder paradigm, we can rewrite it as follows:

$$z_t = h(y_{t-1}, y_{t-2}, \cdots, y_{t-N})$$

$$y_t = g(z_t)$$

Here, h is the encoder and g is the decoder. Each encoder and decoder can be some special architecture suited for time series forecasting. Let's look at a few common components that are used in the encoder-decoder paradigm.

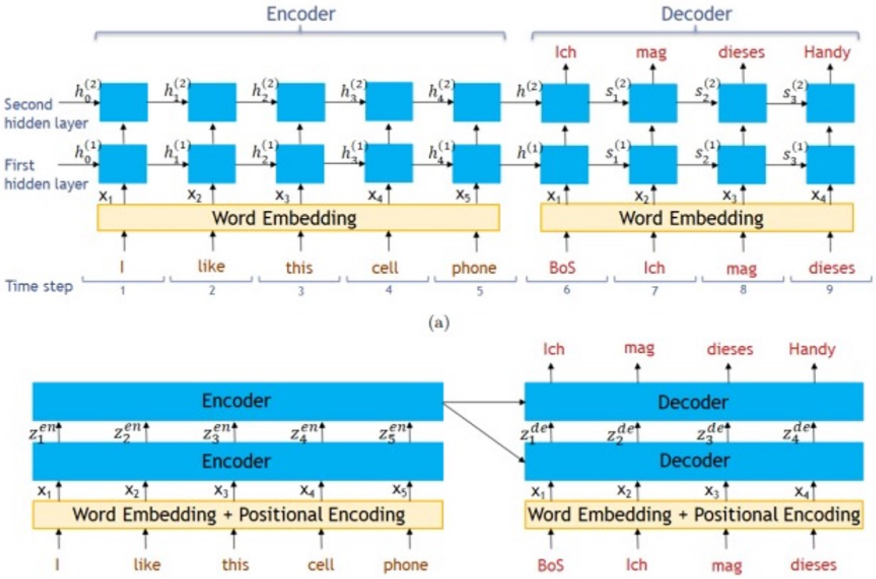
26.9.1 Feed-forward networks

Feed-forward networks (FFNs) or fully connected networks are the most basic architecture a neural network can take. We discussed perceptrons in Chapter 11, Introduction to Deep Learning. If we stack multiple perceptrons (both linear units and non-linear activations) and create a network of such units, we get what we call an FFN.

A Feedforward Neural Network (FFN) processes a fixed-size input vector through computational layers, culminating in the desired output. It's termed "feed-forward" because information moves sequentially through the network. Also known as a fully connected network, each unit in a layer connects to every unit in the previous and next layers. The input layer matches the input dimension, and the output layer is defined by the desired output. Hidden layers, positioned in between, contribute to the network's capacity. The network's structure is defined by two hyperparameters: the number of hidden layers and the units in each layer. For example, the illustrated network has two hidden layers, each with eight units. A Feedforward Neural Network (FFN) serves dual roles as both an encoder and a decoder. As an encoder, it transforms the time series problem into a regression task by embedding time. In this role, it's used similarly to machine learning models discussed in Chapter 5. As a decoder, the FFN operates on the latent vector (output from the encoder) to produce the final forecasted output. This is a common and effective usage of FFN in the context of time series forecasting.

26.10 Self-Attention Operation

The Transformer architecture is based on finding associations or correlations between various input segments (after adding the position information in these segments) using the dot product. Let xin i=1, x Rd be a set of n words (or data points) in a single sequence. The subscript i represents the position of the vector xi, which is equivalently the position of the word in the original sentence or the word sequence. The self-attention operation is the weighted dot product of these input vectors xi with each other.



An example of a simple language translation task performed using (a) a classical model, such as an RNN, LSTM, or GRU, and (b) a Transformer.

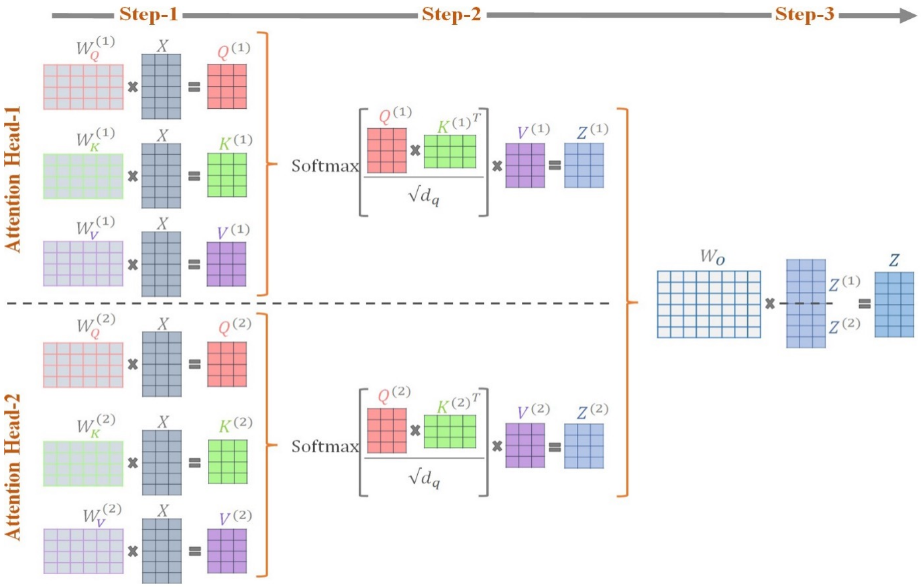
26.10.1 Multi-Head Self-Attention

The input data X may contain several levels of correlation information and the learning process may benefit from processing the input data in multiple different ways. Multiple self-attention heads are introduced that operate on the same input in parallel and use distinct weight matrices Wq, Wk, and Wv, to extract various levels of correlation between the input data. For example, consider the sentence “Do we have to turn left from here or have we left the street behind?”. There are two occurrences of the word “left” in the sentence. Each occurrence has a different meaning, and consequently a different relationship with the rest of the words in the sentence. Transformers can capture such information by using multiple heads. Each head is built using a separate set of query, key, and value weight matrices and computes self-attention over the input sequence in parallel with other heads. The use of multiple heads in a Transformer is analogous to using multiple kernels at each layer in convolutional neural network, where each kernel is responsible for learning distinct features or representations.

Step 1: Generation of Multiple Sets of Distinct Query, Key, and Value Vectors Step 2: Scaled Dot Product Operations in Parallel This step consists of implementing the following relationship, as shown in the figure:

$$Z = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

Step-3 - Concatenating and Linearly Combining Outputs It is important to note that the input and output of the multi-head self-attention are of the same dimension, that is, dimension (X) = dimension (Z).

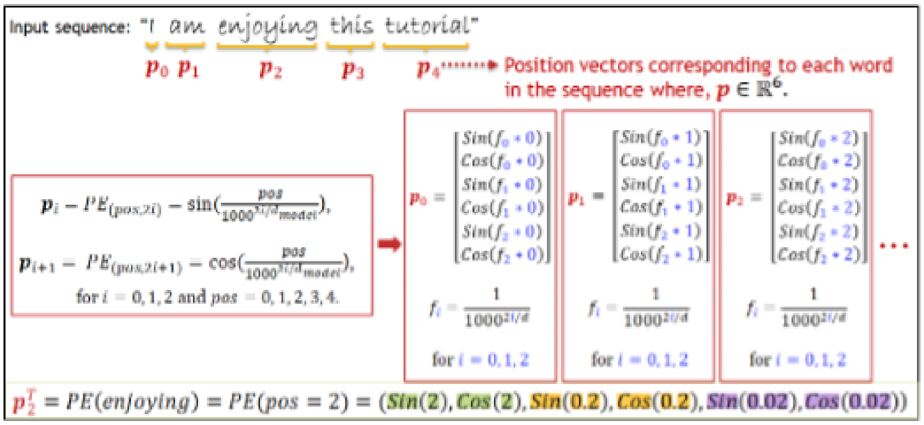


26.10.2 Positional Encoding

Transformers successfully avoided the recurrence and unlocked a performance bottleneck of sequential operations. But now there is a problem. By processing all the positions in a sequence in parallel, the model also loses the ability to understand the relevance of the position. For the Transformer, each position is independent of the other, and hence one key aspect we would seek from a model that processes sequences is missing. The original authors did propose a way to make sure we do not lose this information—positional encoding

$$\begin{aligned} \mathrm{PE}_{(pos,2i)} = \mathbf{p}_i \quad &= \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right), \\ \mathrm{PE}_{(pos,2i+1)} = \mathbf{p}_{i+1} &= \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right), \end{aligned}$$

Where *pos* is the position (time-step) in the sequence of input words. If the input, $\mathbf{X}$, is a d_{model}-dimensional embedding for n tokens in a sequence, positional embedding, $\mathbf{P}$, is a matrix of the same size ($n \times d_{\text{model}}$). The element on the posth row and $2i^{th}$ or $(2i+1)^{th}$ column is defined as follows.



Input sequence: "I am enjoying this tutorial"
p_0 p_1 p_2 p_3 p_4
Position vectors corresponding to each word in the sequence where, $p \in \mathbb{R}^6$.
$p_i = PE_{(pos,2i)} = \sin(\frac{pos}{1000^{2i/d_{model}}})$,
$p_{i+1} = PE_{(pos,2i+1)} = \cos(\frac{pos}{1000^{2i/d_{model}}})$,
for $i = 0, 1, 2$ and $pos = 0, 1, 2, 3, 4$.
$p_0 = [Sin(f_0 * 0), Cos(f_0 * 0), Sin(f_1 * 0), Cos(f_1 * 0), Sin(f_2 * 0), Cos(f_2 * 0)]$
$p_1 = [Sin(f_0 * 1), Cos(f_0 * 1), Sin(f_1 * 1), Cos(f_1 * 1), Sin(f_2 * 1), Cos(f_2 * 1)]$
$p_2 = [Sin(f_0 * 2), Cos(f_0 * 2), Sin(f_1 * 2), Cos(f_1 * 2), Sin(f_2 * 2), Cos(f_2 * 2)]$
$f_i = \frac{1}{1000^{2i/d}}$
for $i = 0, 1, 2$
...
$p_2^T = PE(enjoying) = PE(pos = 2) = (Sin(2), Cos(2), Sin(0.2), Cos(0.2), Sin(0.02), Cos(0.02))$

26.11 Summary

In recent chapters, we covered fundamental concepts and building blocks of deep learning for time series forecasting, applying them practically using PyTorch. We intentionally postponed attention mechanisms and Transformers for a dedicated chapter. We introduced the generalized attention model, explored specific attention schemes, and seamlessly integrated attention into Seq2Seq models. Transitioning to the Transformer, we adapted its architecture for time-series applications, concluding with training a Transformer model for forecasting on a household sample. This chapter lays the groundwork for using deep learning in time series forecasting. The next chapter will explore the global forecasting model paradigm.

27 Chapter 14 Code Analysis

27.1 Code

GitHub reference link

27.2 Conclusion

Each strategy has advantages and disadvantages of its own, and the optimal option will rely on the particular issue at hand. All things considered, combining CNNs with LSTMs can offer a potent tool for handling a variety of sequence learning problems.

27.3 Research Summary

27.4 Encoder-Decoder Transformer Architecture

The encoder-decoder transformer architecture can be represented mathematically as follows
E = Encoder(X)
D = Decoder(E)

27.4.1 Encoder Mechanism

The encoder mechanism can
Z = [Q(X), K(X), V(X)]
H = [Attention(Z), PositionalEncoding(X)]
E = Feedforward(H)

where:

- Q, K, and V are matrices representing the query, key, and value vectors, respectively.
- Attention is a function that computes the attention scores between the query vector and the key vectors.
- PositionalEncoding is a function that adds positional information to the input text.
- Feedforward is a neural network that performs a series of non-linear transformations.

27.4.2 Decoder Mechanism

The decoder mechanism can be represented mathematically as follows:
Y = [Q(Yt), K(Yt), V(Yt)]
M = [MaskedAttention(Z, Yt), DecoderSelfAttention(Yt)]
N = [EncoderDecoderAttention(Z, Yt)]
H = [M, N]
D = Feedforward(H)
Y = Yt + D

where:

- Yt is the output sequence of tokens at time step t.
- MaskedAttention is a function that computes the masked attention scores between the query vector and the key vectors, where the attention is masked to prevent the decoder from attending to future tokens in the output sequence.
- DecoderSelfAttention is a function that computes the decoder self-attention scores between the query vector and the key vectors, which allows the decoder to attend to different parts of its output text.
- EncoderDecoderAttention is a function that computes the encoder-decoder attention scores between the query vector and the key vectors, which allows the decoder to attend to the input text.
- Feedforward is a neural network that performs a series of non-linear transformations.

27.5 Attention Mechanism

The attention mechanism can be represented mathematically as follows

$$S = Q * K^T, A = Softmax(S), V = A * V$$

where:

- Y is the set of all possible outputs.
- P(Y) is the probability distribution over the possible outputs.
- D is the output of the decoder.

27.6 Probabilistic Forecasting

Transformer models can be used to generate probabilistic forecasts by using a softmax function to output a probability distribution over the possible outputs. This can be written mathematically as follows

$$P(Y) = Softmax(D)$$

where:

- Y is the set of all possible outputs.
- P(Y) is the probability distribution over the possible outputs.
- D is the output of the decoder.

27.7 Softmax function

The softmax function is a function that is used to convert a vector of real numbers into a vector of probabilities. It is often used in neural networks to normalize the output of a layer before it is passed to the next layer. The softmax function can be represented mathematically as follows

$$S = Q \cdot K^T$$

where:

- Z is the vector of real numbers.
- S is the vector of probabilities.

The softmax function works by taking a vector of real numbers and calculating the exponential of each number. It then divides each number by the sum of the exponentials. This ensures that the sum of the probabilities is equal to 1. The softmax function is a very important function in neural networks because it allows the network to learn complex relationships between the inputs and the outputs. It is also used to normalize the output of the network, which helps to prevent the network from overfitting to the training data.

Here is an example of how the softmax function can be used to normalize a vector of real numbers:

$$Z = [1, 2, 3]$$

$$S = e^(Z)/Sum(e^(Z))$$

$$S = [2.7183, 7.3891, 20.0855]/30.9505$$

$$S = [0.0930, 0.2397, 0.6673]$$

As you can see, the softmax function has normalized the vector of real numbers to a vector of probabilities. The sum of the probabilities is equal to 1.

27.8 Positional encoding

27.8.1 Positional Encoding with Sinusoids

One common method for positional encoding involves using sinusoidal functions to represent the relative positions of tokens. This approach is widely used in the transformer architecture due to its effectiveness in capturing long-range dependencies between tokens. The basic idea behind positional encoding with sinusoidal functions is to add a set of additional vectors to the input representation, where each vector represents a specific position in the sequence. These vectors are typically created using sine and cosine functions, which vary with the position index. The specific implementation of positional encoding with sinusoidal functions involves multiplying the input representation by a matrix of positional encoding vectors. This matrix is constructed by generating a set of sine and cosine functions for each position in the sequence. The functions have different frequency and offset values for different positions, allowing the model to capture information about the relative distances between tokens. The resulting positional encoding vector is then added to the input representation, effectively embedding the positional information into the model's internal representation of the input sequence. The mathematical formulation for positional encoding with sinusoidal functions can be expressed as follows [**?**]

$$PE(pos) = sin(pos/(10000^2)) + cos(pos/(10000^2))$$

where:

- PE(pos) is the positional encoding vector for position pos.
- 10000 is the embedding dimension.

This formulation generates positional encoding vectors with a period of 10,000, which is sufficient for capturing long-range dependencies in natural language sequences.

27.8.2 Positional Encoding with Learned Parameters

While using pre-trained positional encoding vectors is a common approach, there have been studies that explore the use of learned positional encoding parameters. This approach involves introducing additional parameters to the model that are specifically responsible for learning positional information. The parameters are typically learned during the training process, allowing the model to adapt the positional encoding to the specific characteristics of the training data. The mathematical formulation for learned positional encoding can be expressed as follows

$$PE(pos) = W_pos * pos + b_pos$$

where:

- PE(pos) is the positional encoding vector for position pos.
- Wpos is the weight matrix for positional encoding.
- bpos is the bias vector for positional encoding.

This formulation allows the model to learn a more adaptive and task-specific representation of positional information.

27.9 Token

The input sequence of tokens in the transformer model is generated by splitting the input text into individual words or subwords. This process is called tokenization. Tokenization is necessary because the transformer model cannot work with raw text. It needs to process the text as a sequence of tokens, which are then represented as vectors of numbers.

Two main types of tokenization:

- byte pair encoding : BPE is a more sophisticated approach that can split words into smaller subwords, which can be helpful for languages with complex morphology.
- Wordpiece is a simpler approach that only splits words into subwords if necessary.

Once the input text is tokenized, the transformer model can then process it as a sequence of vectors. These vectors are then passed through the encoder and decoder layers, where they are transformed and combined to generate the output sequence of tokens.

27.10 Data Flow

The transformer model takes an input sequence of tokens and passes it through the encoder, which transforms it into a latent representation. The decoder then takes the latent representation and generates an output sequence of tokens. The decoder is able to generate the output sequence by attending to different parts of the input sequence and its own output sequence.

The data flow through the transformer model can be mathematically represented as follows:

$$X = InputSequence$$
$$E = Encoder(X)$$
$$Z = LatentRepresentation(E)$$
$$Y = Decoder(Z)$$
$$OutputSequence = Y$$

The encoder transforms the input sequence into a latent representation using the encoder layers. The decoder generates the output sequence using the latent representation and the decoder layers.

27.11 References

28 Chapter 15 Summary

29 Chapter 15 Code Analysis

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[3] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[4] Zhiguang Wang, Weizhong Yan, and Tim Oates. Time series classification from scratch with deep neural networks: A strong baseline, 2016.

[5] Zhen Zeng, Rachneet Kaur, Suchetha Siddagangappa, Saba Rahimi, Tucker Balch, and Manuela Veloso. Financial time series forecasting using cnn and transformer, 2023.

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., Polosukhin, I. (2017). Attention is All You Need. arXiv preprint arXiv:1706.03762.
- Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep Learning. MIT Press.
- Bishop, C. M. (2006). Pattern recognition and machine learning. Springer.