Machine Learning for Time Series: A State-Space Framework

Victor M. Preciado

December 5, 2024

Contents

Chapter 1

Linear State-Space Models

State-Space Models (SSMs) provide a cohesive framework for modeling time series data by capturing both the underlying dynamics of the system and the uncertainty present in the observations. This framework builds on the idea of **hidden latent states**, which evolve over time and represent the internal—often unobservable—conditions of the system. The evolution of these states follows a recursive process, where each new state is determined by the previous state, any external inputs, and a noise term that accounts for uncertainty in the system. Meanwhile, the observations are modeled as noisy functions of the hidden states, effectively linking the unobserved dynamics to the data we can observe directly. By incorporating both state evolution and observation mechanisms, SSMs offer a flexible way to model complex time series behaviors.

The state-space framework serves as a unified theoretical foundation for a broad spectrum of time series models. Classical linear models, such as AR, MA, and their combinations, can be viewed as special cases of state-space models where the state equations are linear and the hidden states directly relate to lagged values of the observed data. Moreover, the state-space formulation extends seamlessly to modern deep learning approaches, such as Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, Gated Recurrent Units (GRUs), and neural state-space models such as Mamba. These advanced models can be viewed as extensions of the classical state-space approach, where neural networks are used to parameterize the state evolution and observation equations, allowing for the modeling of non-linear, non-stationary, and highly complex time series data. By positioning SSMs as a general and adaptable framework, we can clarify how these various models, though often discussed separately, all adhere to the same underlying principles.

1.1 State-Space Models

Mathematically, an SSM consists of two main components: the *state equation* (also known as the process model) and the *observation equation* (also known as the measurement model).

- **State Equation**: This equation governs the time evolution of the hidden state vector $\mathbf{X}_k$, which represents the unobserved internal dynamics of the system at time k. The most general version of the state equation is expressed as a nonlinear and time-variant recursive relation:

$$\boxed{\mathbf{X}_{k+1} = f_{\boldsymbol{\theta}_x}(\mathbf{X}_k, \mathbf{u}_k, \boldsymbol{\xi}_k), \quad \mathbf{X}_0 = \mathbf{x}_0,} \tag{1.1}$$

where $\mathbf{X}_k$ is a random vector[1] called the **hidden state** at time k and $\mathbf{u}_k$ denotes the vector of *deterministic* **external inputs** (such as exogenous variables); $f_{\boldsymbol{\theta}_x}(\cdot)$ is the **state transition function** that describes how the hidden state vector evolves based on the current state, inputs, and parameters with $\boldsymbol{\theta}_x$ being the vector of **model parameters** for the state equation, and $\boldsymbol{\xi}_k$ is the **process noise**, which accounts for uncertainties and unmodeled dynamics in the system's evolution.

- **Observation Equation**: The observation equation maps the random hidden state vector $\mathbf{X}_k$ to the observable output vector $\mathbf{Y}_k$, which represents the measured data at time k. This relationship is formalized as:

$$\boxed{\mathbf{Y}_k = g_{\boldsymbol{\theta}_y}(\mathbf{X}_k, \mathbf{u}_k, \boldsymbol{\eta}_k),} \tag{1.2}$$

where $\mathbf{Y}_k$ denotes the **observable data** at time k, $\mathbf{u}_k$ represents the external inputs, $g_{\boldsymbol{\theta}_y}(\cdot)$ is the observation function with parameters $\boldsymbol{\theta}_y$, and $\boldsymbol{\eta}_k$ is the **observation noise**, accounting for measurement errors or uncertainties in the data collection process. This equation encapsulates how the latent dynamics of the system are reflected in the observable data, while also acknowledging the inherent noise and errors in the measurement process.

Together, these two equations recursively describe the stochastic behavior of the system, providing a clear distinction between the latent system dynamics (through the state equation) and the process of observation (through the observation equation). Importantly, the state-space framework induces a **Markov process**, where all relevant information about the system's evolution is encapsulated in the hidden state vector $\mathbf{X}_k$. This Markov property implies that the future behavior of the system depends solely on the current hidden state and not on the full history of past states.

Example 1: Pendulum with Friction as a Nonlinear State-Space Model

The dynamics of a pendulum with friction can be modeled using a discrete-time nonlinear state-space model. Let us denote the angle of displacement of the pendulum from the vertical position as x_{k1} and the angular velocity as x_{k2}.

- **State Variables:** The state vector $\mathbf{x}_k \in \mathbb{R}^2$ is defined as:

$$\mathbf{x}_k = \begin{bmatrix} x_{k1} \\ x_{k2} \end{bmatrix} = \begin{bmatrix} \text{angle (rad)} \\ \text{angular velocity (rad/s)} \end{bmatrix}.$$

- **State Equations:** Assuming a pendulum of length L, mass m, and a friction coefficient c, the equations of motion can be derived from Newton's second law

[1] Note that this vector is random due to the inclusion of the process noise $\boldsymbol{\xi}_k$. In what follows, we denote random vectors by bold capital letters.

and discretized for a time step Δt using Euler's method:

$$x_{k+1,1} = x_{k1} + \Delta t \cdot x_{k2},$$
$$x_{k+1,2} = x_{k2} + \Delta t \cdot \left(-\frac{g}{L}\sin(x_{k1}) - \frac{c}{m}x_{k2} + u_k\right) + \xi_k$$

where g is the acceleration due to gravity, u_k is an external torque applied to the pendulum at time k, and ξ_k represents process noise, capturing any random perturbations.

- **Observation Equation:** In this model, the observed output y_k at each time step is a noisy measurement of the pendulum's angle x_{k1}:

$$y_k = x_{k1} + \eta_k,$$

where η_k is the observation noise.

Thus, the system can be expressed as a discrete-time state-space model where the nonlinear state equation describes the dynamics of the angle and angular velocity, and the observation equation provides a noisy measurement of the angle. Using this state-space formulation, we can analyze the pendulum's behavior over time and estimate its state under various conditions.

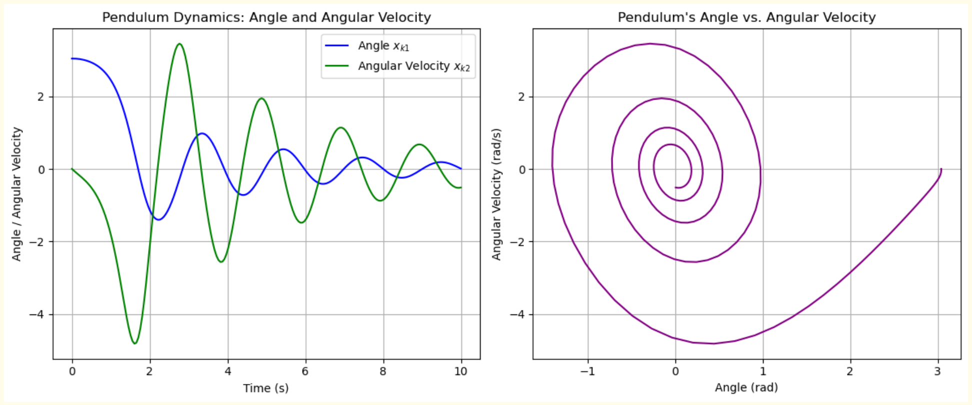


Figure 1.1: (Left) Trajectory of the pendulum's angle and velocity over time. (Right) Pendulum's angle x_{k1} compared with angular velocity x_{k2}.

This model can be expanded to handle multiple pendula or to explore chaotic dynamics in double pendulum systems, with further extensions involving Lagrangian or Hamiltonian dynamics.

55

1.2 Classical Forecasting Models in State-Space Form

Linear State-Space Models (LSSMs) provide a tractable framework for modeling and forecasting time-series data. By leveraging linear relationships between latent (hidden) states and observed outputs, LSSMs strike an effective balance between complexity and analytical tractability, making them a foundational tool in time series analysis. A major strength of LSSMs is their ability to reconstruct many classical time-series models, such as autoregressive, moving average, and other extensions, within a unified framework. This allows for a seamless transition between different modeling paradigms, while also enabling generalizations to more complex systems.

In an LSSM, both the state equation and the observation process are governed by linear transformations. Specifically, the hidden state at each time step is updated as a linear function of the previous state, external inputs, and process noise, while the observed output is modeled as a linear function of the current state, inputs, and measurement noise. Unlike Hidden Markov Models (HMMs), where the hidden state is a discrete random variable, the hidden state in LSSMs is a *random vector of continuous random variables*, reflecting the continuous nature of the underlying system. These linear transformations can always be conveniently expressed in terms of matrix operations[2]. This matrix formulation not only simplifies the mathematical treatment of the model but also allows for efficient computation, especially when dealing with high-dimensional data.

In the following sections, we will formalize the structure of LSSMs, discuss their properties, and explore how they can be applied to time series forecasting and estimation tasks. Let us now present the core equations of the LSSM in matrix form:

- The **state equation** in an LSSM is mathematically described by the following recursion:

$$\boxed{\mathbf{X}_{k+1} = A_k\mathbf{X}_k + B_k\mathbf{u}_k + \boldsymbol{\xi}_k, \quad \mathbf{X}_0 = \mathbf{x}_0,} \tag{1.3}$$

  where $\mathbf{X}_k$ is the **hidden state vector** at time k, which is a vector containing continuous random variables. The matrix A_k, known as the **state transition matrix**, defines how the current state vector $\mathbf{X}_k$ influences the subsequent state $\mathbf{X}_{k+1}$, encapsulating the internal dynamics of the system. The matrix B_k, called the **input matrix**, models the effect of known, deterministic **external inputs $\mathbf{u}_k$** on the state evolution. The random vector $\boldsymbol{\xi}_k$ represents **process noise**, typically modeled as a multivariate white Gaussian noise, i.e., $\boldsymbol{\xi}_k \sim_{\text{iid}} \mathcal{N}(\mathbf{0}, Q)$, where Q is the process noise covariance matrix. This state equation governs the system's hidden state dynamics, describing how the state vector evolves over time as a function of the previous state, external inputs, and random disturbances.

- The **measurement equation** maps the latent states to the observed data. It is mathematically expressed as:

$$\boxed{\mathbf{Y}_k = C_k\mathbf{X}_k + D_k\mathbf{u}_k + \boldsymbol{\eta}_k,} \tag{1.4}$$

  where $\mathbf{Y}_k$ is the **observation vector** at time k. The matrix C_k, referred to as the **observation matrix**, maps the hidden state vector $\mathbf{X}_k$ to the observed data,

[2]Assuming finite-dimensional state and output vectors.

translating the latent dynamics into real-world measurements. The matrix D_k, called the **direct transmission matrix**, captures any direct effects of the inputs on the observations, while the random vector $\boldsymbol{\eta}_k$ represents **measurement noise**, typically modeled as a multivariate white Gaussian noise (independent of the process noise), i.e., $\boldsymbol{\eta}_k \sim_{\text{iid}} \mathcal{N}(\mathbf{0}, R)$, where R is the measurement noise covariance matrix. This equation models how the measurement errors and other external factors may influence the observed data.

In the general formulation of Linear State-Space Models (LSSMs), the matrices A_k, B_k, C_k, and D_k are allowed to be time-varying, meaning their elements can change with each time step k. However, in many practical applications, these matrices are often assumed to be **time-invariant**, in which case the subscript k is dropped from the notation. LSSMs model with time-invariant system matrices are called Linear Time-Invariant (LTI) systems or models. Change LSSM to LTI in the rest of the book (whenever applicable)... Thus, the matrices A, B, C, and D remain constant over time. A diagrammatic representation of a time-invariance LSSM can be found in Fig. 1.2.

Insert diagram of LTI system.

Figure 1.2: Block diagram representation of the linear state-space equations, where the $\mathcal{D}$-block represents a one-time-step delay, i.e., $\mathcal{D}\, \boldsymbol{X}_{k+1} = \boldsymbol{X}_k$.

In the rest of this section, we explore several classical forecasting models through the lens of Linear State-Space Models (LSSMs). Representing forecasting models in LSSM form provides a unified framework that allows for the application of powerful analysis, estimation, and control techniques. By structuring these models within a state-space formulation, we can examine their underlying dynamics, assess stability, and develop efficient algorithms for filtering, smoothing, and forecasting. This approach is particularly useful for time series models such as Autoregressive (AR), Moving Average (MA), and Autoregressive Integrated Moving Average (ARIMA), which are widely used in practical forecasting applications. We will derive the state-space representations of these models, highlighting the transition and measurement equations that capture their essential characteristics. This unifying perspective not only simplifies their implementation but also facilitates a deeper understanding of the relationships among these classical models.

1.2.1 Autoregressive model in State-Space Form

Consider the Autoregressive process of order p, denoted AR(p), defined by the equation:

$$\boxed{Y_{k+1} = \phi_1 Y_k + \phi_2 Y_{k-1} + \cdots + \phi_p Y_{k-p+1} + \epsilon_k,}$$

where $\epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2)$.This AR($p$) model describes the current value Y_{k+1} as a linear combination of the past p values of the process. The random variables ϵ_k are i.i.d. Gaussians. By reformulating this model in a state-space framework, we gain access to a range of analytical tools available for linear dynamical systems.

To represent this AR(p) process as a Linear State-Space Model (LSSM), we introduce a state vector, a state transition equation, and a measurement equation.

1. **State Equation:**

   We define the state vector $\mathbf{X}_k^{\text{AR}} \in \mathbb{R}^p$ to encapsulate the history of the process up to the previous p values, as follows:

$$\mathbf{X}_k^{\text{AR}} = \begin{bmatrix} Y_k & Y_{k-1} & \cdots & Y_{k-p+2} & Y_{k-p+1} \end{bmatrix}^\intercal.$$

   This state vector allows us to describe the evolution of the AR(p) process in a recursive form. The state equation, which dictates how $\mathbf{X}_k^{\text{AR}}$ transitions to $\mathbf{X}_{k+1}^{\text{AR}}$, is then given by:

$$\underbrace{\begin{bmatrix} Y_{k+1} \\ Y_k \\ \vdots \\ Y_{k-p+3} \\ Y_{k-p+2} \end{bmatrix}}_{\mathbf{X}_{k+1}^{\text{AR}} \in \mathbb{R}^p} = \underbrace{\begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{p-1} & \phi_p \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}}_{A^{\text{AR}} \in \mathbb{R}^{p\times p}} \underbrace{\begin{bmatrix} Y_k \\ Y_{k-1} \\ \vdots \\ Y_{k-p+2} \\ Y_{k-p+1} \end{bmatrix}}_{\mathbf{X}_k^{\text{AR}}} + \underbrace{\begin{bmatrix} \epsilon_k \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{\boldsymbol{\xi}_k^{\text{AR}} \in \mathbb{R}^p},$$

   where A^{AR} is the $p \times p$ state transition matrix, structured to incorporate the autoregressive coefficients $\phi_1, \phi_2, \ldots, \phi_p$ in the first row, while the subdiagonal entries facilitate shifting each previous observation down one position in the state vector. The vector $\boldsymbol{\xi}_k^{\text{AR}}$ introduces the white noise term ϵ_k into the system, with zeros in all other entries, maintaining the dimensionality of the state vector.

2. **Measurement Equation:**

   The measurement equation establishes a direct relationship between the state vector $\mathbf{X}_k^{\text{AR}}$ and the observed process value Y_k. Specifically, the observation at each time step is given by the first element of the state vector, allowing us to write:

$$Y_k = \underbrace{\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \end{bmatrix}}_{C^{\text{AR}} \in \mathbb{R}^{1\times p}} \mathbf{X}_k^{\text{AR}},$$

   where C^{AR} is the observation matrix, designed to extract the first component of $\mathbf{X}_k^{\text{AR}}$, corresponding to the current value Y_k.

In summary, this state-space representation of the AR(p) model provides a systematic way to express autoregressive processes within a linear dynamical system framework. The state equation recursively describes the evolution of the process, while the measurement equation ensures that the observed data correspond to the current state. This formalism enables the application of state-space methods such as Kalman filtering, state estimation, and control, which are valuable for both theoretical analysis and practical implementation.

1.2.2 Moving Average in State-Space Form

The Moving Average process of order q, denoted as MA(q), is a time series model in which each observation Y_k is a linear combination of the current and past q noise terms. Formally, the MA(q) process is defined by:

$$\boxed{Y_k = \epsilon_k + \theta_1 \epsilon_{k-1} + \theta_2 \epsilon_{k-2} + \cdots + \theta_q \epsilon_{k-q},}$$

where $\epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2)$. Each observation Y_k is thus constructed as a weighted sum of current and past noise values, with weights $\theta_1, \theta_2, \ldots, \theta_q$ corresponding to the moving average coefficients.

The MA(q) process can be elegantly reformulated as a Linear State-Space Model (LSSM), which provides a systematic framework for analyzing such processes. We accomplish this by introducing a state vector, a state equation that governs its evolution, and a measurement equation that relates the state to the observed data.

1. **State Equation:**

   To represent the MA(q) model in state-space form, we define the state vector $\mathbf{X}_k^{\text{MA}} \in \mathbb{R}^{q+1}$ as a vector containing the current and past q noise terms:

   $$\mathbf{X}_k^{\text{MA}} = \begin{bmatrix} \epsilon_k & \epsilon_{k-1} & \cdots & \epsilon_{k-q+1} & \epsilon_{k-q} \end{bmatrix}^{\mathsf{T}}.$$

   This choice of state vector allows us to track the noise terms involved in the MA(q) process, enabling a recursive description of the model.

   The state equation describes the evolution of $\mathbf{X}_k^{\text{MA}}$ over time by shifting each past noise term down one position at each time step. Mathematically, this evolution is expressed as:

   $$\underbrace{\begin{bmatrix} \epsilon_k \\ \epsilon_{k-1} \\ \vdots \\ \epsilon_{k-q+1} \\ \epsilon_{k-q} \end{bmatrix}}_{\mathbf{X}_k^{\text{MA}} \in \mathbb{R}^{q+1}} = \underbrace{\begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}}_{A^{\text{MA}} \in \mathbb{R}^{(q+1)\times(q+1)}} \underbrace{\begin{bmatrix} \epsilon_{k-1} \\ \epsilon_{k-2} \\ \vdots \\ \epsilon_{k-q} \\ \epsilon_{k-q-1} \end{bmatrix}}_{\mathbf{X}_{k-1}^{\text{MA}}} + \underbrace{\begin{bmatrix} \epsilon_k \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{\boldsymbol{\xi}_k^{\text{MA}} \in \mathbb{R}^{q+1}}.$$

   Here, A^{MA} is the state transition matrix, structured to shift each noise term down by one position while inserting a zero at the last position. The noise vector $\boldsymbol{\xi}_k^{\text{MA}}$ introduces the current noise term ϵ_k into the system, effectively updating the state vector with each new observation.

2. **Measurement Equation:**

   The measurement equation relates the state vector $\mathbf{X}_k^{\text{MA}}$ to the observed value Y_k by applying the moving average coefficients. In the case of the MA(q) process, the observed value Y_k is obtained by linearly combining the elements of $\mathbf{X}_k^{\text{MA}}$ using the moving average coefficients $\theta_1, \theta_2, \ldots, \theta_q$. This relationship is given by:

   $$Y_k = \underbrace{\begin{bmatrix} 1 & \theta_1 & \theta_2 & \cdots & \theta_q \end{bmatrix}}_{C^{\text{MA}} \in \mathbb{R}^{1\times(q+1)}} \mathbf{X}_k^{\text{MA}}.$$

   The matrix C^{MA} serves as the observation matrix, designed to apply the MA coefficients to the respective elements of $\mathbf{X}_k^{\text{MA}}$. This yields Y_k as a weighted sum of the current noise term ϵ_k and its past values, aligning with the original MA(q) process.

In summary, the state-space representation of the MA(q) model provides a convenient framework for analyzing and interpreting moving average processes. By recasting the MA(q) process in this form, we can leverage state-space methods for filtering, estimation, and forecasting. This formulation clarifies the recursive structure of the MA model and offers a systematic approach to handle its components within a unified linear framework.

1.2.3 Autoregressive Moving-Average in State-Space Form

The Autoregressive Moving-Average (ARMA) model is a widely-used time series model that combines the properties of both autoregressive (AR) and moving average (MA) processes. In general, an ARMA(p, q) model uses p autoregressive terms and q moving average terms to describe the dynamics of a time series, capturing dependencies on both past values and past error terms. For simplicity, we consider here the specific case of an ARMA$(2, 2)$ model, where $p = 2$ and $q = 2$. However, the methodology outlined below can be extended to ARMA(p, q) models of any order.

Let us consider the Autoregressive Moving-Average model of order $(2, 2)$, denoted as ARMA$(2, 2)$, which is defined by the following equation:

$$\boxed{Y_k = \phi_1 Y_{k-1} + \phi_2 Y_{k-2} + \epsilon_k + \theta_1 \epsilon_{k-1} + \theta_2 \epsilon_{k-2},}$$

where $\epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2)$. Here, ϕ_1 and ϕ_2 are the autoregressive (AR) coefficients, while θ_1 and θ_2 are the moving average (MA) coefficients. This ARMA$(2, 2)$ process combines autoregressive terms (which depend on past values of Y_k) with moving average terms (which depend on past error terms ϵ_k).

To reformulate the ARMA$(2, 2)$ model as a Linear State-Space Model (LSSM), we construct an augmented state vector that incorporates both the past observations Y_{k-1} and Y_{k-2} and the past error terms ϵ_{k-1} and ϵ_{k-2}. This representation allows us to describe the ARMA process within a structured, recursive framework, facilitating analysis and control.

1. **State Equation:**

   We define the state vector $\mathbf{X}_k \in \mathbb{R}^4$ as:

$$\mathbf{X}_k = \begin{bmatrix} Y_{k-1} & Y_{k-2} & \epsilon_{k-1} & \epsilon_{k-2} \end{bmatrix}^\intercal .$$

   The state equation describes the evolution of $\mathbf{X}_k$ over time, capturing the dynamics of both the autoregressive and moving average components. Specifically, we can express this evolution as:

$$\underbrace{\begin{bmatrix} Y_k \\ Y_{k-1} \\ \epsilon_k \\ \epsilon_{k-1} \end{bmatrix}}_{\mathbf{X}_k} = \underbrace{\begin{bmatrix} \phi_1 & \phi_2 & \theta_1 & \theta_2 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} Y_{k-1} \\ Y_{k-2} \\ \epsilon_{k-1} \\ \epsilon_{k-2} \end{bmatrix}}_{\mathbf{X}_{k-1}} + \underbrace{\begin{bmatrix} \epsilon_k \\ 0 \\ \epsilon_k \\ 0 \end{bmatrix}}_{\boldsymbol{\xi}_k},$$

   where A is the state transition matrix, which encodes the autoregressive coefficients ϕ_1 and ϕ_2 as well as the moving average coefficients θ_1 and θ_2. The noise vector $\boldsymbol{\xi}_k$ introduces the current error term ϵ_k into the state vector, thus ensuring that the model adheres to the structure of an ARMA$(2, 2)$ process.

2. **Measurement Equation:**

   The measurement equation links the state vector $\mathbf{X}_k$ to the observed data Y_k. Since Y_k corresponds directly to the first component of $\mathbf{X}_k$, we express the measurement equation as:

$$Y_k = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}}_{C \in \mathbb{R}^{1\times 4}} \mathbf{X}_k .$$

Here, C is the observation matrix, which extracts the first element of the state vector to provide the current observation Y_k.

In this state-space representation, the ARMA$(2,2)$ process is described by a system of recursive equations. The state equation encodes the evolution of past values and error terms, while the measurement equation captures the observed output. This unified framework enables the application of state-space analysis and filtering techniques, such as the Kalman filter, to the ARMA$(2,2)$ model, thus facilitating estimation and prediction within a structured linear system.

1.2.4 ARIMA Model in State-Space Form

The **Autoregressive Integrated Moving Average** (ARIMA) model extends the Autoregressive Moving Average (ARMA) model by including an *integration* component, which accounts for non-stationary trends in a time series. While ARMA models are suited for stationary processes, ARIMA models apply **differencing** in an attempt to convert non-stationary data with trends in a stationary series, enabling the effective modeling of a wider range of time series. Differencing is a widely-used technique aimed at transforming a non-stationary time series with trends into a stationary one. For a time series $(Y_k)_{k\geq 0}$, differencing involves calculating the change between consecutive observations, which can help to reduce slow-moving patterns or trends in the data.

Given a time series $(Y_k)_{k\geq 0}$, we define the **delay operator** $\mathcal{D}$ such that $\mathcal{D}\,Y_k = Y_{k-1}$. This operator allows us to compactly represent differencing operations by delaying each observation by a specified number of time steps. In general, **differencing of order** d is represented by applying the differencing operator $(1-\mathcal{D})^d$, which removes polynomial trends of degree d. Higher-order differencing can be applied as needed to achieve stationarity, though care must be taken, as excessive differencing may result in an over-differenced series with oscillatory behavior.

Applying differencing of any order can help stabilize a time series by reducing or removing trends, thus aiding in meeting the stationarity requirement for many time series models. Typically, differencing is applied iteratively, with each application removing progressively higher-order patterns or trends in the data. However, while differencing can make a series stationary, it does not always guarantee stationarity, and its effectiveness depends on the underlying characteristics of the data. Furthermore, excessive differencing can introduce new problems, such as amplifying noise or creating an over-differenced series that oscillates around zero without meaningful patterns. This loss of structure can complicate modeling and lead to poorer forecasting performance. Therefore, it is important to identify the minimal degree of differencing necessary to achieve approximate stationarity, balancing trend removal with the preservation of the series' essential characteristics.

To illustrate the state-space representation of an ARIMA model, we consider a specific example of an ARIMA(p,d,q) model with $p=d=q=1$. This example clarifies how the integrated component interacts with the autoregressive and moving average parts within a unified framework. In an ARIMA$(1,1,1)$ model, the differencing order $d=1$ implies that we apply first-order differencing on Y_k to make the series stationary. This differencing process transforms the original series $(Y_k)_{k=0}^{L}$ into a stationary series $(W_k^{(1)})_{k=0}^{L-1}$, where $W_k^{(1)} = Y_k - Y_{k-1}$. Thus, the ARIMA$(1,1,1)$ model becomes equivalent to an ARMA$(1,1)$

model for the first-order differenced series, represented as:

$$W_k^{(1)} = \phi_1 W_{k-1}^{(1)} + \epsilon_k + \theta_1 \epsilon_{k-1}.$$

The ARIMA$(1,1,1)$ model can now be expressed in state-space form by defining a state vector that incorporates both the ARMA structure and the differencing order.

1. **State Vector and State Equation:** Define the state vector $\mathbf{X}_k \in \mathbb{R}^3$ as follows:

$$\mathbf{X}_k = \begin{bmatrix} W_k^{(1)} & \epsilon_k & Y_{k-1} \end{bmatrix}^\intercal.$$

The state equation describes the evolution of $\mathbf{X}_k$:

$$\underbrace{\begin{bmatrix} W_k^{(1)} \\ \epsilon_k \\ Y_{k-1} \end{bmatrix}}_{\mathbf{X}_k} = \underbrace{\begin{bmatrix} \phi_1 & \theta_1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} W_{k-1}^{(1)} \\ \epsilon_{k-1} \\ Y_{k-2} \end{bmatrix}}_{\mathbf{X}_{k-1}} + \underbrace{\begin{bmatrix} \epsilon_k \\ \epsilon_k \\ 0 \end{bmatrix}}_{\boldsymbol{\xi}_k}.$$

Here, the last row of the state matrix A enforces the relation:

$$W_{k-1}^{(1)} = Y_{k-1} - Y_{k-2} \iff Y_{k-1} = W_{k-1}^{(1)} + Y_{k-2}.$$

The transition matrix A incorporates the autoregressive coefficient ϕ_1 and the moving average coefficient θ_1 while also implementing the differencing operation. As such, the transition matrix A applies first-order differencing automatically at each step.

2. **Measurement Equation:**
The measurement equation directly relates the state vector to the observed output Y_k using the relation $W_k^{(1)} = Y_k - Y_{k-1}$, which implies that $Y_k = W_k^{(1)} + Y_{k-1}$. This can be written as a vector product:

$$Y_k = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \mathbf{X}_k.$$

The differencing component in the ARIMA$(1,1,1)$ model captures a non-stationary trend by recursively differencing the original series to obtain a stationary series $(W_k)_k$. This allows us to model the trend component separately from the stationary ARMA structure, thus facilitating effective modeling within the state-space framework. This formulation allows us to represent the ARIMA model's differencing, autoregressive, and moving average components in a unified state-space framework, enabling the application of linear state-space techniques for analysis, forecasting, and control of the ARIMA$(1,1,1)$ process.

1.2.5 SARIMA Model

While the ARIMA model is effective for modeling time series with trends, it does not inherently address seasonal components, which are common in many real-world datasets. The **Seasonal Autoregressive Integrated Moving Average** (SARIMA) model extends ARIMA by introducing seasonal components, denoted by the addition of seasonal terms that repeat over fixed periods. This extension enables SARIMA to capture recurring patterns at multiple periodicities, making it well-suited for time series with both trend and seasonal structures.

To illustrate the modeling of time series with seasonal components, we consider a specific example of electric power consumption data. This dataset exhibits a natural daily periodic component due to the recurring patterns of electricity usage over 24-hour cycles. To model such data effectively, we use the **Seasonal Autoregressive Integrated Moving Average** (SARIMA) model. This model extends the ARIMA framework by including a seasonal component that repeats over a fixed period. For our example, the SARIMA model is denoted by:

$$\text{SARIMA}(p, d, q) \times (P, D, Q)_s,$$

where:

- (p, d, q) are the non-seasonal orders for the autoregressive (AR), integration (I), and moving average (MA) terms.
- $(P, D, Q)_s$ are the seasonal orders for the AR, differencing, and MA terms, with $s = 24$, corresponding to the daily periodicity.

The SARIMA equation can be compactly expressed as:

$$\begin{aligned}\left(1-\sum_{i=1}^{p}\phi_i\mathcal{D}^i\right)\cdot\left(1-\sum_{j=1}^{P}\Phi_j\mathcal{D}^{js}\right)\cdot(1-\mathcal{D})^d\cdot(1-\mathcal{D}^s)^D Y_k \\ =\left(1+\sum_{i=1}^{q}\theta_i\mathcal{D}^i\right)\cdot\left(1+\sum_{j=1}^{Q}\Theta_j\mathcal{D}^{js}\right)\epsilon_k,\end{aligned}$$

where:

- ϕ_i and θ_i (for $i = 1, \dots, p$ and $i = 1, \dots, q$) are the non-seasonal autoregressive (AR) and moving average (MA) coefficients, respectively.
- Φ_j and Θ_j (for $j = 1, \dots, P$ and $j = 1, \dots, Q$) are the seasonal AR and MA coefficients, respectively, corresponding to multiples of the seasonal period s.
- $(1-\mathcal{D})^d$ is the non-seasonal differencing term (where $\mathcal{D}\,Y_k = Y_{k-1}$).
- $(1-\mathcal{D}^s)^D$ is the seasonal differencing term for the periodicity $s = 24$.

This formulation captures the interplay between non-seasonal and seasonal components in the SARIMA model using an additive representation:

- The **non-seasonal AR term**, $1-\sum_{i=1}^{p}\phi_i\mathcal{D}^i$, describes how lagged values of the time series contribute to its current value.
- The **seasonal AR term**, $1-\sum_{j=1}^{P}\Phi_j\mathcal{D}^{js}$, captures the influence of lagged values at seasonal lags (multiples of the seasonal period s).
- The **non-seasonal MA term**, $1+\sum_{i=1}^{q}\theta_i\mathcal{D}^i$, relates the current value to past forecast errors.

- The **seasonal MA term**, $1 + \sum_{j=1}^{Q} \Theta_j \mathcal{D}^{js}$, relates the current value to past forecast errors at seasonal lags.

- The **differencing terms**, $(1 - \mathcal{D})^d$ and $(1 - \mathcal{D}^s)^D$, account for non-seasonal and seasonal trends, ensuring stationarity of the series.

This additive formulation is widely used in time-series analysis because it clearly reflects the influence of each component, making it intuitive and interpretable. It is particularly well-suited for analyzing time series with daily periodicity, such as electric power consumption, as it succinctly incorporates both non-seasonal and seasonal dynamics.

Python Lab: SARIMA for Power Load Prediction

This lab demonstrates how to preprocess a time series dataset, fit a SARIMA model, and evaluate its forecasting performance. We will use the Python `statsmodels` library to implement the Seasonal ARIMA (SARIMA) model. The example dataset contains hourly power load data.

1. **Import Libraries**: First, we import the necessary libraries for data handling, modeling, and visualization. Suppressing warnings ensures cleaner output during the modeling process.

```python
# Import necessary libraries
import pandas as pd
from statsmodels.tsa.statespace.sarimax import SARIMAX
import warnings
import matplotlib.pyplot as plt

# Suppress warnings for cleaner output
warnings.filterwarnings("ignore")
```

2. **Load and Preprocess the Dataset**: The dataset is loaded into a pandas DataFrame, and the columns are renamed for clarity. The `Date` column is converted into a datetime format, then set as the index for easier time series manipulation.

```python
# Load the dataset
data_path = '2011_2019_hourly_power_load_MidAtl.csv'
df = pd.read_csv(data_path)

# Rename columns and set the datetime column
df.rename(columns={'forecast_hour_beginning_ept': 'Date', '
    Electric Load (MW)': 'Value'}, inplace=True)
df['Date'] = pd.to_datetime(df['Date'], format='%m/%d/%y %H:%M')
df.set_index('Date', inplace=True)
```

3. **Define and Split the Time Series**: The `Value` column, which contains the hourly power load, is extracted as the time series. The data is then split into training and validation sets, with 80% allocated for training and 20% for validation.

```python
# Define the time series
time_series = df['Value']

# Split into training and validation sets
```

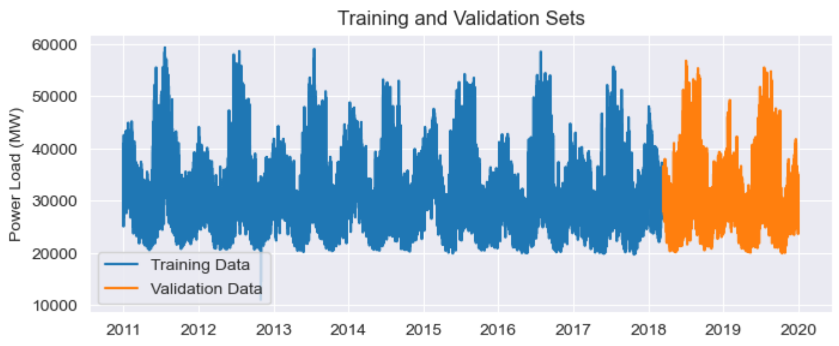


Figure 1.3: Training data (80%) and validation data (20%).

```
train_size = int(len(time_series) * 0.8)
train, validation = time_series[:train_size], time_series[
    train_size:]
```

4. **Define the SARIMA Model** A SARIMA model is defined with specified non-seasonal parameters ($p = 3, d = 1, q = 3$) and seasonal parameters ($P = 2, D = 0, Q = 2, m = 24$), reflecting a seasonal period of 24 hours.

```
# Define the SARIMA model with the specified parameters
sarima_model = SARIMAX(
    train,
    order=(3, 1, 3),                # ARIMA(3,1,3)
    seasonal_order=(2, 0, 2, 24),  # Seasonal component
        (2,0,2,24)
    enforce_stationarity=False,
    enforce_invertibility=False
)
```

5. **Fit the Model** The SARIMA model is fit to the training data using the maximum likelihood estimation (MLE) method. The summary of the fitted model provides useful diagnostics, such as AIC and parameter significance.

```
# Fit the model to the training data
fitted_model = sarima_model.fit(disp=False)

# Print the model summary
print(fitted_model.summary())
```

6. **Forecast Future Values** The fitted model is used to generate forecasts for the validation period. Forecasted values and their confidence intervals are extracted for analysis.

```
# Forecast on the validation set
forecast = fitted_model.get_forecast(steps=len(validation))
forecast_mean = forecast.predicted_mean
forecast_ci = forecast.conf_int()
```

7. **Visualize the Results** The training data, validation data, forecasted values, and confidence intervals are plotted for comparison. The plot helps visually assess the model's performance.

```python
# Plot the actual vs forecasted values
plt.figure(figsize=(10, 6))
plt.plot(train, label='Training Data')
plt.plot(validation, label='Validation Data')
plt.plot(forecast_mean, label='Forecast', color='red')
plt.fill_between(forecast_ci.index, forecast_ci.iloc[:, 0],
    forecast_ci.iloc[:, 1], color='pink', alpha=0.3)
plt.xlabel('Date')
plt.ylabel('Values')
plt.title('SARIMA Model Forecast')
plt.legend()
plt.show()
```

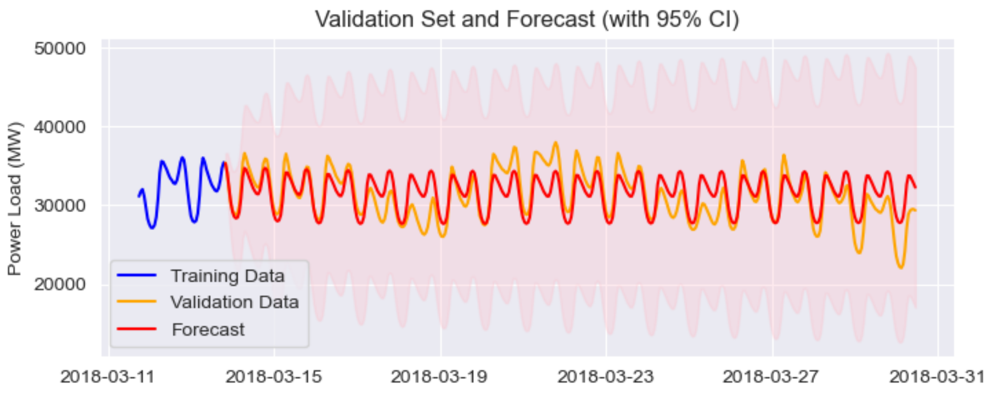


Figure 1.4: Training and validation sets with forecasts. The plot highlights the last 50 samples of the training data and the first 400 samples of the validation data, along with the SARIMA model's forecast and confidence intervals.

1.2.6 Model Diagnostics

The SARIMA model summary includes important statistics, which are shown in the following table. The SARIMAX(3, 1, 3)(2, 0, 2, 24) model, summarized in Table 1.1, was fitted to a dataset comprising $63,092$ observations of power load. The model achieves a log-likelihood of $-475,483.611$, indicating the overall quality of the fit. The p-values of the parameter estimates shown in the table reveal statistically significant contributions to the model (some parameters, not shown for brevity, are not statistically significant). For example, the autoregressive term `ar.L1` has a coefficient of 1.8459, with a standard error of 0.487, yielding a z-score of 3.787 and a p-value of 0.000. Similarly, the seasonal moving average term `ma.S.L24` has a coefficient of -0.5025 and is highly significant (p-value 0.000), confirming the model's ability to capture daily seasonal patterns at lag 24 hours. The residual variance, σ^2, is estimated at 4.286×10^5, reflecting the scale of the data. Model diagnostics, including the Ljung-Box test at lag 1 ($Q = 0.02$, p-value 0.90), suggest no significant

Table 1.1: SARIMAX(3, 1, 3)(2, 0, 2, 24) Summary

Model Details				
Variable	Load			
Observations	63,092			
Log Likelihood	-475,483.611			
Parameters	**Coeff.**	**Std Err**	**z**	**p-value**
ar.L1	1.8459	0.487	3.787	0.000
⋮	⋮	⋮	⋮	⋮
ma.S.L24	-0.5025	0.023	-21.474	0.000
sigma2	4.286×10^5	1657.255	258.600	0.000
Diagnostics	**Value**	**p-value**		
Ljung-Box	0.02	0.90		
Heteroskedasis	0.73	0.00		

autocorrelation in the residuals. However, the heteroskedasticity test ($H = 0.73$, p-value 0.00) indicates potential heteroskedasticity, which may require further consideration. Overall, the SARIMAX model effectively captures both short-term dependencies and seasonal structures, making it well-suited for forecasting tasks.

1.2.7 SARIMAX Model

The **Seasonal Autoregressive Integrated Moving Average with Exogenous Variables (SARIMAX)** model extends the SARIMA framework by incorporating additional predictors, known as **exogenous variables**. These variables, which are external to the time series itself, can significantly enhance forecasting accuracy by accounting for factors influencing the series. The SARIMAX model is particularly useful when external predictors—such as temperature, holiday indicators, or promotional activity—play a crucial role in driving the time series behavior. This flexibility makes SARIMAX well-suited for complex applications, including energy forecasting, sales prediction, and economic analysis.

The SARIMAX model builds on the SARIMA formulation (see previous section) by adding a term to incorporate exogenous variables. It is expressed as:

$$\left(1-\sum_{i=1}^{p}\phi_i\mathcal{D}^i\right)\left(1-\sum_{j=1}^{P}\Phi_j\mathcal{D}^{js}\right)(1-\mathcal{D})^d(1-\mathcal{D}^s)^D Y_k = \left(1+\sum_{i=1}^{q}\theta_i\mathcal{D}^i\right)\left(1+\sum_{j=1}^{Q}\Theta_j\mathcal{D}^{js}\right)\epsilon_k + \boxed{\boldsymbol{\beta}^\intercal \mathbf{u}_k},$$

where $\mathbf{u}_k$ is a vector of exogenous variables (predictors) at time k and $\boldsymbol{\beta}$ is a vector of coefficients representing the influence of these exogenous variables on the dependent variable Y_k. The additional term $\boldsymbol{\beta}^\intercal \mathbf{u}_k$ allows SARIMAX to incorporate external factors that are not captured by the time series' intrinsic patterns, such as trends or seasonality.

Python Lab: Power Load Forecasting Using SARIMAX

This section demonstrates the application of the SARIMAX model to forecast hourly power load data while incorporating temperature as an exogenous variable. The SARIMAX model extends the traditional SARIMA framework by accounting for external factors influencing the target time series. The workflow is divided into several logical steps:

1. **Data Preparation**: In this step, we load, preprocess, and merge two datasets: one containing hourly power load data and another with hourly temperature data. These datasets are indexed by their date-time column to enable seamless integration.

```python
# Load the electric load data
load_data = pd.read_csv('2011_2019_hourly_power_load_MidAtl.csv')
date_column = 'forecast_hour_beginning_ept'
load_column = 'forecast_load_mw'
load_data['Date time'] = pd.to_datetime(load_data[date_column], format='%m/%d/%y %H:%M', errors='coerce')
load_data.set_index('Date time', inplace=True)
load_data.rename(columns={load_column: 'Electric Load (MW)'}, inplace=True)

# Load the weather data
weather_data = pd.read_csv('2011_2019_hourly_weather_Philly.csv')
weather_data['Date time'] = pd.to_datetime(weather_data['Date time'], format='%m/%d/%y %H:%M')
weather_data.set_index('Date time', inplace=True)
weather_data = weather_data[['Temperature']]

# Merge the load and weather data
combined_data = load_data.join(weather_data, how='inner')
first_720_samples = combined_data.head(24 * 30)
```

2. **Visualizing Power Load and Temperature**: The relationship between power load and temperature over the first 30 days is visualized in Figure 1.5. This visualization helps in understanding the correlation between these two variables.

```python
# Plot power load and temperature
fig, ax1 = plt.subplots(figsize=(9, 3.5))
ax1.plot(first_720_samples.index, first_720_samples['Electric Load (MW)'], label='Electric Load (MW)', color='red')
ax1.set_ylabel('Electric Load (MW)', color='red')
ax1.tick_params(axis='y', labelcolor='red')

ax2 = ax1.twinx()
ax2.plot(first_720_samples.index, first_720_samples['Temperature'], label='Temperature (Celsius)', color='blue')
ax2.set_ylabel('Temperature (Celsius)', color='blue')
ax2.tick_params(axis='y', labelcolor='blue')

ax1.xaxis.set_major_locator(mdates.DayLocator(interval=7))
ax1.xaxis.set_major_formatter(mdates.DateFormatter('%b %d'))
plt.title('Electric Load and Temperature Evolution (30 Days)')
plt.tight_layout()
plt.show()
```

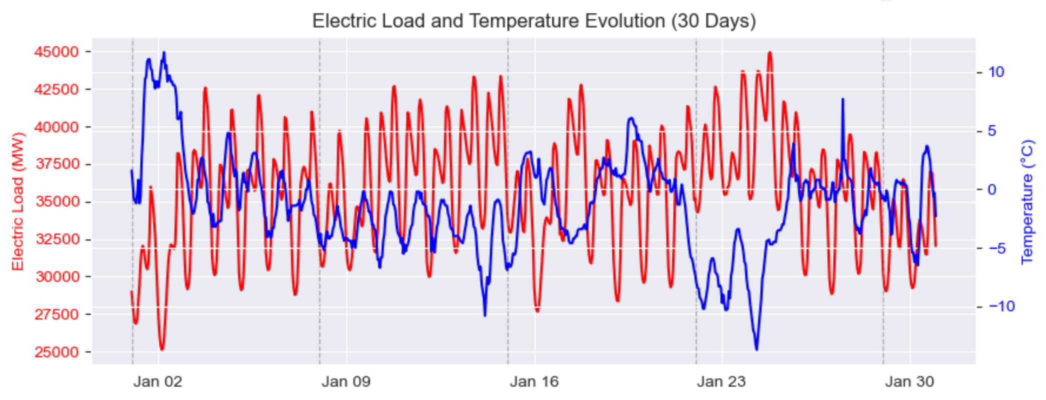


Figure 1.5: Electric Load and Temperature Evolution Over 30 Days

3. **Training the SARIMAX Model**

   The dataset is split into training (80%) and validation (20%) subsets. A SARIMAX model is defined with the following parameters: non-seasonal order `(p=2, d=1, q=1)`, seasonal order `(P=2, D=0, Q=2, s=24)`, and exogenous variable `Temperature`. The model is then fitted to the training data.

```
# Split data into training and validation
train_size = int(len(combined_data) * 0.8)
y_train = combined_data['Electric Load (MW)'][:train_size]
y_val = combined_data['Electric Load (MW)'][train_size:]
X_train = combined_data['Temperature'][:train_size].values.
    reshape(-1, 1)
X_val = combined_data['Temperature'][train_size:].values.
    reshape(-1, 1)

# Define and fit the SARIMAX model
from statsmodels.tsa.statespace.sarimax import SARIMAX
sarimax_model = SARIMAX(
    y_train,
    exog=X_train,
    order=(2, 1, 1),
    seasonal_order=(2, 0, 2, 24),
    enforce_stationarity=False,
    enforce_invertibility=False
)
sarimax_results = sarimax_model.fit(disp=False)
print(sarimax_results.summary())
```

Table 1.2 provides a detailed summary of the SARIMAX model used to forecast hourly electric load, incorporating temperature as an exogenous variable. In the table, we observe that the `Temperature` coefficient is 364.6956, indicating a strong positive relationship between temperature and electric load. A one-unit increase in temperature is associated with an increase of approximately 365 MW in electric load. We also observe a significant coefficients for AR, MA, and seasonal components suggest the importance of lagged dependencies and daily seasonal effects in the time series. Notice that the p-values for most parameters are below 0.05, indicating statistical significance. Furthermore, the residual variance (σ^2) is estimated at 5.721×10^5, representing the

Table 1.2: SARIMAX(2, 1, 1)(2, 0,2, 24) Summary

Model Details				
Variable	Electric Load (MW)			
Observations	63,092			
Log Likelihood	-486,485.008			
Parameters	**Coeff.**	**Std Err**	**z**	**p-value**
Temperature	364.6956	3.601	101.274	0.000
AR(1)	1.5359	0.003	481.885	0.000
AR(2)	-0.5556	0.003	-179.633	0.000
MA(1)	-0.9999	0.000	-2761.697	0.000
Seasonal AR(24)	0.9563	0.039	24.393	0.000
Seasonal MA(24)	-0.6334	0.038	-16.461	0.000
⋮	⋮	⋮	⋮	⋮
sigma2	5.721×10^5	2765.997	206.848	0.000
Diagnostics	**Value**	**p-value**		
Ljung-Box (L1) (Q)	0.01	0.91		
Heteroskedasticity (H)	0.82	0.00		

variability not explained by the model. The `Ljung-Box Test` indicates no significant autocorrelation in the residuals, with a p-value of 0.91 and the `Heteroskedasticity Test` indicates non-constant variance in residuals, with a p-value of 0.00 confirming significance.

4. **Evaluating the Model's Forecasting Performance**

Forecasts are generated for the validation set, and confidence intervals are computed to assess uncertainty. The results are visualized in Figure 1.6.

```python
# Forecast on validation data
forecast = sarimax_results.get_forecast(steps=len(y_val),
    exog=X_val)
forecast_mean = forecast.predicted_mean
forecast_ci = forecast.conf_int()

# Plot forecasts and validation data
plt.figure(figsize=(8, 3))
plt.plot(y_train[-50:], label='Training Data', color='blue')
plt.plot(y_val[:400], label='Validation Data', color='orange'
    )
plt.plot(forecast_mean[:400], label='Forecast', color='red')
plt.fill_between(
    forecast_ci.index[:400],
    forecast_ci.iloc[:400, 0],
    forecast_ci.iloc[:400, 1],
    color='pink',
    alpha=0.3,
    label='95% CI'
)
plt.ylabel('Power Load (MW)')
plt.title('Validation Set and Forecast (with 95% CI)')
plt.legend(loc='lower left')
plt.tight_layout()
```

```
    plt.show()
```

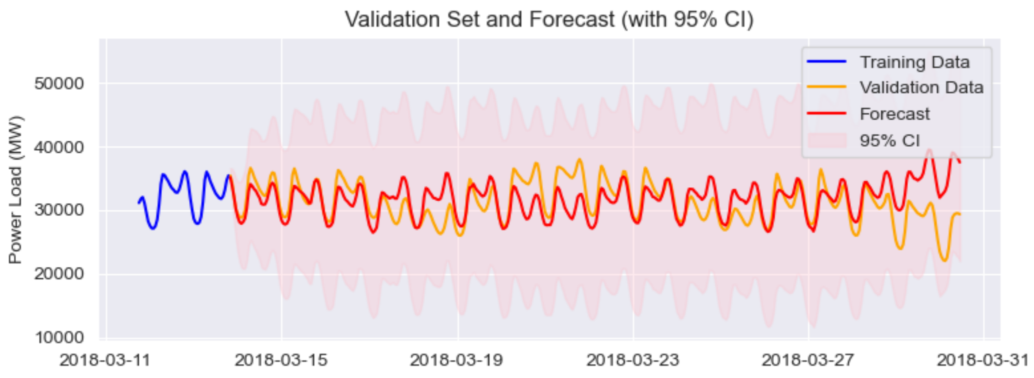


Figure 1.6: Final 50 samples of the validation set and the first 400 samples of the forecast, including 95% confidence intervals. The plot illustrates the model's accuracy and uncertainty in the early forecast period.

This section illustrates the practical application of SARIMAX, highlighting its ability to integrate external predictors such as temperature. The methodology enables more accurate forecasts while providing uncertainty quantification through confidence intervals. The integration of data preparation, visualization, modeling, and evaluation makes this a comprehensive introduction to SARIMAX-based forecasting.

1.3 Temporal Evolution of LTI Models

In this subsection, we derive the solution to the linear state-space model. For simplicity, we consider the case of the noiseless state-space model, in which both the process and measurement noises are zero. Notice that, in the noiseless case, the state and output vectors are deterministic, assuming that the initial condition is also deterministic. The evolution of the noiseless LSSM is governed by the following equations:

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k, \tag{1.5}$$
$$\mathbf{y}_k = C\mathbf{x}_k + D\mathbf{u}_k, \tag{1.6}$$

with given initial condition $\mathbf{x}_0$. The state equation (1.5) can be recursively expanded to express the state vector $\mathbf{x}_k$ as a function of the initial state $\mathbf{x}_0$ and the sequence of inputs and noise terms. Starting from the initial condition $\mathbf{x}_0$, we obtain:

$$\begin{aligned}
\mathbf{x}_1 &= A\mathbf{x}_0 + B\mathbf{u}_0,\\
\mathbf{x}_2 &= A\mathbf{x}_1 + B\mathbf{u}_1 = A^2\mathbf{x}_0 + AB\mathbf{u}_0 + B\mathbf{u}_1,\\
\mathbf{x}_3 &= A\mathbf{x}_2 + B\mathbf{u}_2 = A^3\mathbf{x}_0 + A^2B\mathbf{u}_0 + AB\mathbf{u}_1 + B\mathbf{u}_2.
\end{aligned}$$

In general, the state vector at time k is given by:

$$\mathbf{x}_k = A^k\mathbf{x}_0 + \sum_{i=1}^{k} A^{i-1}B\mathbf{u}_{k-i}. \tag{1.7}$$

This expression illustrates how the current state depends on the initial state $\mathbf{x}_0$ and the history of inputs $\mathbf{u}_i$.

The output equation (1.6) can now be used to compute the output $\mathbf{y}_k$ as a function of the initial state vector $\mathbf{x}_0$, and the sequence of inputs and noise terms. Substituting the expression (1.7) into the output equation, we have:

$$\boxed{\mathbf{y}_k = CA^k\mathbf{x}_0 + \sum_{i=1}^{k} CA^{i-1}B\mathbf{u}_{k-i} + D\mathbf{u}_k.} \tag{1.8}$$

This solution can be written in terms of the so-called **Markov parameters** of the LSSM. The k-th Markov parameter is defined as the matrix:

$$H_i = CA^{i-1}B, \text{ for } i \geq 1; \quad H_0 = D.$$

Using the Markov parameters, the summation terms in (1.8) can be written in terms of **discrete convolutions**.

A discrete convolution is an operation that combines two discrete temporal sequences and outputs another temporal sequence, such that each element of the output sequence is a weighted sum of properly shifted versions of the input sequences (see Fig. ?). Mathematically, the convolution of two sequences $\mathbf{a} = (a_i)_{i\geq 0}$ and $\mathbf{b} = (b_i)_{i\geq 0}$ is defined as:

$$\boxed{(\mathbf{a} * \mathbf{b})_k = \sum_{i=0}^{k} a_i b_{k-i}.}$$

In the context of state-space models, the summation term in (1.8) can be written in terms of the matrix-valued sequence of Markov parameters $\mathbf{H} = (H_i)_{i\geq 0}$ and the vector-valued sequence of inputs $\mathbf{u} = (\mathbf{u}_i)_{i\geq 0}$ as follows:

$$\boxed{\mathbf{y}_k = CA^k\mathbf{x}_0 + (\mathbf{H} * \mathbf{u})_k.} \tag{1.9}$$

Thus, the output $\mathbf{y}_k$ at time k depends on:

- The term $CA^k\mathbf{x}_0$ represents the contribution of the initial state $\mathbf{x}_0$, propagated over time.
- The convolution term $(H * \mathbf{u})_k$ captures the cumulative effect of input sequence over time.

This expression elegantly captures the contributions of the initial state and the inputs, process noise, highlighting the key role of the system matrices A, B, C, and D in shaping the output evolution.

The choice of hidden state variables in a Linear State-Space Model (LSSM) is not unique. This means that different representations of the state variables can yield the same input-output behavior. This flexibility arises because the state variables are internal constructs

Insert diagram.

Figure 1.7: Pictorial representation of the convolution of two temporal sequences.

that describe the system's evolution and are not directly observable. Consequently, it is possible to transform the state variables and the associated system matrices (A, B, C, D) into an equivalent representation while preserving the same input-output relationship. This property, known as *state-space equivalence*, underscores the inherent freedom in modeling dynamic systems and the role of state transformations in simplifying or adapting the representation for specific purposes. Further details and an exploration of this property are provided in the accompanying exercise.

1.3.1 Uncertainty Quantification in LTI Models

Confidence intervals (CIs) are a fundamental tool in statistical modeling, offering a quantitative measure of the uncertainty associated with predictions or estimates. In the context of LSSMs, confidence intervals provide probabilistic bounds around forecasts by incorporating the effects of system dynamics, process noise, and observation noise. These intervals enable practitioners to assess the reliability of model outputs and make informed decisions under uncertainty.

The construction of confidence intervals in LSSMs is grounded in the Gaussian nature of the state and observation distributions, which arise from the linearity of the model and the assumption of normally distributed noise. Given a trained LSSM, predictions for future observations can be expressed as a multivariate Gaussian distribution, characterized by a mean vector and a covariance matrix. Confidence intervals are derived from this distribution, using the covariance structure to quantify the spread of possible outcomes around the mean.

Evolution of the Mean

To quantify the uncertainty in predictions, we consider a stochastic LSSM with *time-invariant* system matrices (A, B, C, D). This model is governed by equations that describe the system's state evolution and observation processes:

$$\begin{aligned} \mathbf{X}_{k+1} &= A\mathbf{X}_k + B\mathbf{u}_k + \boldsymbol{\xi}_k, \quad \boldsymbol{\xi}_k \sim \mathcal{N}(\mathbf{0}, Q), \\ \mathbf{Y}_k &= C\mathbf{X}_k + D\mathbf{u}_k + \boldsymbol{\eta}_k, \quad \boldsymbol{\eta}_k \sim \mathcal{N}(\mathbf{0}, R), \end{aligned}$$

where $\mathbf{X}_k$ and $\mathbf{Y}_k$ represent the random state and output vectors at time k, respectively. The matrices Q and R denote the covariance matrices associated with process noise and observation noise, capturing their inherent variability. These governing equations define the system's state dynamics and measurement process, illustrating the interplay between the model's internal structure and external influences.

To derive confidence intervals for the forecasted outputs of a stochastic LSSM, it is essential to analyze the evolution of both the mean and the covariance of the output variable. The construction of confidence intervals begins with understanding the deterministic evolution of the mean for the state and output variables, which is governed by the following (deterministic) equations:

$$\widetilde{\mathbf{x}}_{k+1} = \mathbb{E}[\mathbf{X}_{k+1}] = \mathbb{E}[A\mathbf{X}_k + B\mathbf{u}_k] = A\mathbb{E}[\mathbf{X}_k] + B\mathbf{u}_k = A\widetilde{\mathbf{x}}_k + B\mathbf{u}_k,$$

$$\widetilde{\mathbf{y}}_k = \mathbb{E}[\mathbf{Y}_k] = \mathbb{E}[C\mathbf{X}_k + D\mathbf{u}_k] = C\mathbb{E}[\mathbf{X}_k] + D\mathbf{u}_k = C\widetilde{\mathbf{x}}_k + D\mathbf{u}_k.$$

These equations are structurally identical to those describing the evolution of a noiseless LSSM. Consequently, they can be solved using (1.9), yielding the following equation describing the evolution of the output mean:

$$\boxed{\widetilde{\mathbf{y}}_k = CA^k\mathbf{x}_0 + (\mathbf{H} * \mathbf{u})_k.}$$

Evolution of the Covariance Matrix

To derive confidence intervals, it is also necessary to analyze the evolution of the covariance matrix of the output vector, defined as

$$\boxed{\mathbf{\Sigma}_k = \mathbb{E}[\mathbf{Y}_k\mathbf{Y}_k^\intercal] - \widetilde{\mathbf{y}}_k\widetilde{\mathbf{y}}_k^\intercal.}$$

The evolution of $\mathbf{\Sigma}_k$ can be expressed as:

$$\begin{aligned}
\mathbf{\Sigma}_k &= \mathbb{E}[\mathbf{Y}_k\mathbf{Y}_k^\intercal] - \widetilde{\mathbf{y}}_k\widetilde{\mathbf{y}}_k^\intercal \\
&= \mathbb{E}[(C\mathbf{X}_k + D\mathbf{u}_k + \boldsymbol{\eta}_k)(C\mathbf{X}_k + D\mathbf{u}_k + \boldsymbol{\eta}_k)^\intercal] - \widetilde{\mathbf{y}}_k\widetilde{\mathbf{y}}_k^\intercal \\
&= C\mathbb{E}[\mathbf{X}_k\mathbf{X}_k^\intercal]C^\intercal + \mathbb{E}[\boldsymbol{\eta}_k\boldsymbol{\eta}_k^\intercal] + D\mathbf{u}_k\mathbf{u}_k^\intercal D^\intercal \\
&\quad + C\widetilde{\mathbf{x}}_k\mathbf{u}_k^\intercal D^\intercal + D\mathbf{u}_k\widetilde{\mathbf{x}}_k^\intercal C^\intercal - \widetilde{\mathbf{y}}_k\widetilde{\mathbf{y}}_k^\intercal.
\end{aligned}$$

Separately, the term $\widetilde{\mathbf{y}}_k\widetilde{\mathbf{y}}_k^\intercal$ can be expanded as:

$$\begin{aligned}
\widetilde{\mathbf{y}}_k\widetilde{\mathbf{y}}_k^\intercal &= (C\widetilde{\mathbf{x}}_k + D\mathbf{u}_k)(C\widetilde{\mathbf{x}}_k + D\mathbf{u}_k)^\intercal \\
&= C\widetilde{\mathbf{x}}_k\widetilde{\mathbf{x}}_k^\intercal C^\intercal + D\mathbf{u}_k\mathbf{u}_k^\intercal D^\intercal + C\widetilde{\mathbf{x}}_k\mathbf{u}_k^\intercal D^\intercal + D\mathbf{u}_k\widetilde{\mathbf{x}}_k^\intercal C^\intercal.
\end{aligned}$$

Substituting this result back into the equation for $\mathbf{\Sigma}_k$ and simplifying terms, we obtain:

$$\begin{aligned}
\mathbf{\Sigma}_k &= C\mathbb{E}[\mathbf{X}_k\mathbf{X}_k^\intercal]C^\intercal - C\widetilde{\mathbf{x}}_k\widetilde{\mathbf{x}}_k^\intercal C^\intercal + \mathbb{E}[\boldsymbol{\eta}_k\boldsymbol{\eta}_k^\intercal] \\
&= C\mathbb{E}[(\mathbf{X}_k - \widetilde{\mathbf{x}}_k)(\mathbf{X}_k - \widetilde{\mathbf{x}}_k)^\intercal]C^\intercal + R \\
&= CP_kC^\intercal + R,
\end{aligned}$$

where R is the covariance matrix of the measurement noise and P is the covariance matrix of the hidden state at time k, i.e.,

$$\boxed{P_k = \mathbb{E}[(\mathbf{X}_k - \widetilde{\mathbf{x}}_k)(\mathbf{X}_k - \widetilde{\mathbf{x}}_k)^\intercal].}$$

Thus, to compute $\mathbf{\Sigma}_k$, it is essential to first derive an expression for P_k, which encapsulates the uncertainty in the state estimate at time k.

To compute P_k, let us consider the state estimation residual, $\boldsymbol{\omega}_k = \mathbf{X}_k - \widetilde{\mathbf{x}}_k$. Hence, the state estimation residual at time $k+1$ can be expressed as:

$$\begin{aligned}
\boldsymbol{\omega}_{k+1} &= \mathbf{X}_{k+1} - \widetilde{\mathbf{x}}_{k+1} \\
&= (A\mathbf{X}_k + B\mathbf{u}_k + \boldsymbol{\xi}_k) - (A\widetilde{\mathbf{x}}_k + B\mathbf{u}_k) \\
&= A(\mathbf{X}_k - \widetilde{\mathbf{x}}_k) + \boldsymbol{\xi}_k \\
&= A\boldsymbol{\omega}_k + \boldsymbol{\xi}_k.
\end{aligned}$$

with the corresponding covariance matrix:

$$\begin{aligned} P_{k+1} &= \mathbb{E}\left[\boldsymbol{\omega}_{k+1}\boldsymbol{\omega}_{k+1}^\intercal\right] \\ &= \mathbb{E}\left[(A\boldsymbol{\omega}_k + \boldsymbol{\xi}_k)(A\boldsymbol{\omega}_k + \boldsymbol{\xi}_k)^\intercal\right] \\ &= A\mathbb{E}\left[\boldsymbol{\omega}_k\boldsymbol{\omega}_k^\intercal\right]A^\intercal + A\mathbb{E}\left[\boldsymbol{\omega}_k\boldsymbol{\xi}_k^\intercal\right] + \mathbb{E}\left[\boldsymbol{\xi}_k\boldsymbol{\omega}_k^\intercal\right]A^\intercal + \mathbb{E}\left[\boldsymbol{\xi}_k\boldsymbol{\xi}_k^\intercal\right]. \end{aligned}$$

Since the process noise $\boldsymbol{\xi}_k$ is uncorrelated with the state estimation residual $\boldsymbol{\omega}_k$, the cross terms vanish:

$$\mathbb{E}\left[\boldsymbol{\omega}_k\boldsymbol{\xi}_k^\intercal\right] = \mathbb{E}\left[\boldsymbol{\xi}_k\boldsymbol{\omega}_k^\intercal\right] = 0.$$

Thus, the covariance update simplifies to:

$$\boxed{P_{k+1} = AP_kA^\intercal + Q.}$$

Starting from an initial covariance matrix full of zeros, i.e., $P_0 = \mathbb{O}_{n\times n}$, this equation allows us to compute P_k recursively. Finally, the covariance matrix of the output vector $\mathbf{Y}_k$ is computed as:

$$\boxed{\boldsymbol{\Sigma}_k = CP_kC^\intercal + R,}$$

where $R = \mathbb{E}\left[\boldsymbol{\eta}_k\boldsymbol{\eta}_k^\intercal\right]$ represents the covariance of the measurement noise $\boldsymbol{\eta}_k$.

These last two equations allow us to compute the matrix $\boldsymbol{\Sigma}_k$, which represents the covariance of the output vector $\mathbf{Y}_k$, from the system matrices and the covariances of the measurement and process noises. This matrix encapsulates the uncertainty in the forecasted output, arising from both process noise and measurement noise. The covariance matrix $\boldsymbol{\Sigma}_k$ plays a central role in constructing confidence intervals for the predicted values, as we will illustrate in the subsequent sections.

Confidence Ellipsoids for the Output Vector

With the covariance matrix $\boldsymbol{\Sigma}_k$ computed, we can construct confidence regions for $\mathbf{Y}_k$. Since the output vector $\mathbf{Y}_k$ is generated by a linear model with Gaussian noise, $\mathbf{Y}_k$ follows a multivariate normal distribution $\mathbf{Y}_k \sim \mathcal{N}(\widetilde{\mathbf{y}}_k, \boldsymbol{\Sigma}_k)$. Consequently, we can **whiten** the signal $\mathbf{Y}_k$, i.e., we can linearly transform it into another random vector $\mathbf{Z}_k$ that follows a standard multivariate normal distribution. This transformation is achieved as follows:

$$\mathbf{Z}_k = \boldsymbol{\Sigma}_k^{-1/2}(\mathbf{Y}_k - \widetilde{\mathbf{y}}_k),$$

where $\boldsymbol{\Sigma}_k^{-1/2}$ is the **matrix square root**[3] of $\boldsymbol{\Sigma}_k^{-1}$, satisfying $\boldsymbol{\Sigma}_k^{-1/2}\boldsymbol{\Sigma}_k(\boldsymbol{\Sigma}_k^{-1/2})^\intercal = \mathbb{I}_p$. Hence, the covariance matrix of $\mathbf{Z}_k$ satisfies:

$$\mathbb{E}\left[\mathbf{Z}_k\mathbf{Z}_k^\intercal\right] = \mathbb{E}[\boldsymbol{\Sigma}_k^{-1/2}(\mathbf{Y}_k - \widetilde{\mathbf{y}}_k)(\mathbf{Y}_k - \widetilde{\mathbf{y}}_k)^\intercal(\boldsymbol{\Sigma}_k^{-1/2})^\intercal] = \mathbb{I}_p.$$

Also, the mean vector of $\mathbf{Z}_k$ satisfies:

$$\mathbb{E}[\mathbf{Z}_k] = \boldsymbol{\Sigma}_k^{-1/2}(\mathbb{E}[\mathbf{Y}_k] - \widetilde{\mathbf{y}}_k) = \boldsymbol{\Sigma}_k^{-1/2}(\widetilde{\mathbf{y}}_k - \widetilde{\mathbf{y}}_k) = \mathbf{0}_p.$$

[3]The square root of a positive definite matrix can be computed using the eigendecomposition $\boldsymbol{\Sigma} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\intercal$, where $\mathbf{U}$ is an orthogonal matrix of eigenvectors and $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues. The square root is then given by $\boldsymbol{\Sigma}^{1/2} = \mathbf{U}\boldsymbol{\Lambda}^{1/2}\mathbf{U}^\intercal$, where $\boldsymbol{\Lambda}^{1/2}$ is obtained by taking the square root of each diagonal entry of $\boldsymbol{\Lambda}$.

Therefore, $\mathbf{Z}_k \sim \mathcal{N}(\mathbf{0}_p, \mathbb{I}_p)$, meaning the components of $\mathbf{Z}_k$ are i.i.d. standard normal variables. Furthermore, the sum of the squares of p standard normal i.i.d. random variables follows a a χ_p^2 distribution with p degrees of freedom[4]; therefore, we have that:

$$\sum_{i=1}^{p} Z_{k,i}^2 = \mathbf{Z}_k^\intercal \mathbf{Z}_k \sim \chi_p^2.$$

When the covariance matrix $\mathbf{\Sigma}_k$ is symmetric and positive definite, it induces a distance metric, known as the **Mahalanobis distance**, defined as:

$$\boxed{d_M^2(\mathbf{y}, \widetilde{\mathbf{y}}_k) = (\mathbf{y} - \widetilde{\mathbf{y}}_k)^\intercal \mathbf{\Sigma}_k^{-1}(\mathbf{y} - \widetilde{\mathbf{y}}_k) \geq 0.}$$

The squared Mahalanobis distance between the random vector $\mathbf{Y}_k$ and its mean $\widetilde{\mathbf{y}}_k$ is a random variable that satisfies:

$$d_M^2(\mathbf{Y}_k, \widetilde{\mathbf{y}}_k) = (\mathbf{Y}_k - \widetilde{\mathbf{y}}_k)^\intercal \mathbf{\Sigma}_k^{-1}(\mathbf{Y}_k - \widetilde{\mathbf{y}}_k) = \mathbf{Z}_k^\intercal \mathbf{Z}_k \sim \chi_p^2.$$

The β-confidence interval for this random variable is defined by $d_M^2(\mathbf{Y}_k, \widetilde{\mathbf{y}}_k) \leq \chi_{p,\beta}^2$, where $\chi_{p,\beta}^2$ represents the β-quantile of the **chi-squared distribution** with p degrees of freedom. This confidence interval induces the following inequality in the space of the output vector:

$$\boxed{(\mathbf{y} - \widetilde{\mathbf{y}}_k)^\intercal \mathbf{\Sigma}_k^{-1}(\mathbf{y} - \widetilde{\mathbf{y}}_k) \leq \chi_{p,\beta}^2.}$$

Since $\mathbf{\Sigma}_k^{-1}$ is positive definite, this inequality defines an ellipsoid centered at $\widetilde{\mathbf{y}}_k$ whose principal axes are aligned with the eigenvectors of $\mathbf{\Sigma}_k$ (see Fig. 1.8-(left)).

Confidence ellipsoids are a natural extension of confidence intervals to multivariate settings. While a confidence interval quantifies uncertainty for a single variable, a confidence ellipsoid captures the joint uncertainty of all components of $\mathbf{Y}_k$, incorporating the relationships between them. The β-confidence ellipsoid is constructed such that the squared Mahalanobis distance remains within the critical value $\chi_{p,\beta}^2$. This ensures that the ellipsoid captures a (β-fraction of the probability mass of the multivariate normal distribution, providing a rigorous probabilistic interpretation of the joint uncertainty in $\mathbf{Y}_k$. The ellipsoid adapts its shape based on $\mathbf{\Sigma}_k$, elongating along directions of higher uncertainty and shrinking along directions of lower uncertainty.

Example 2: Confidence ellipsoid

Consider the case where $p = 2$ and $\mathbf{Y}_k \sim \mathcal{N}(\widetilde{\mathbf{y}}_k, \mathbf{\Sigma}_k)$ with mean $\widetilde{\mathbf{y}}_k = \begin{bmatrix} 0 & 0 \end{bmatrix}^\intercal$ and covariance matrix:

$$\mathbf{\Sigma}_k = \begin{bmatrix} 2.0 & 0.8 \\ 0.8 & 1.0 \end{bmatrix}.$$

[4]The density function of the χ_p^2 distribution is $f_W(w) = \frac{1}{2^{p/2}\Gamma(p/2)} w^{(p/2)-1} e^{-w/2}$ for $w \geq 0$, where $\Gamma(\cdot)$ is the Gamma function.

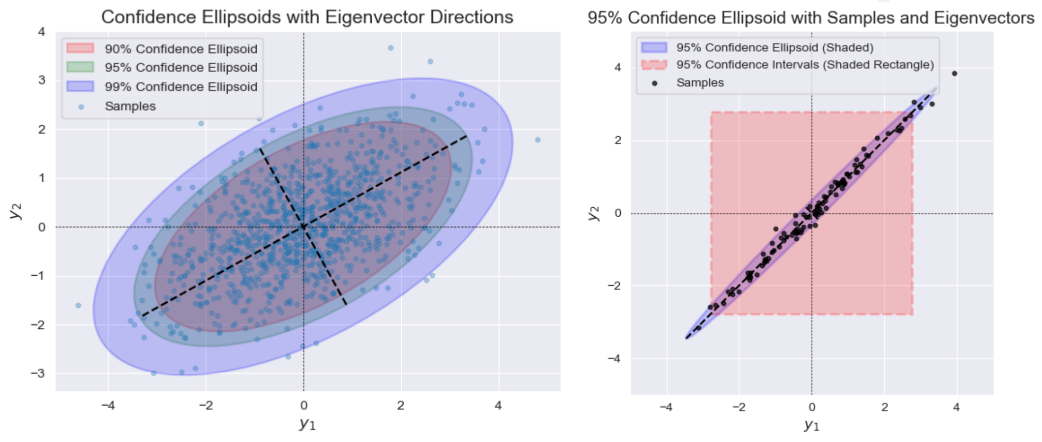


Figure 1.8: (Left) Confidence ellipsoids for 90%, 95%, and 99% levels, superimposed with 1,000 random samples. The dashed black lines indicate the directions of the eigenvectors of the covariance matrix, representing the principal axes of the ellipsoids. (Right) 95% confidence ellipsoid (blue, shaded) and marginal intervals (red, shaded) for two correlated variables, with eigenvectors (black dashed) and 100 random samples (black dots).

The β-confidence ellipsoid is then defined by:

$$(\mathbf{y} - \widetilde{\mathbf{y}}_k)^\intercal \mathbf{\Sigma}_k^{-1} (\mathbf{y} - \widetilde{\mathbf{y}}_k) \leq \chi^2_{p,\beta}.$$

Fig. 1.8-(left) illustrates these confidence ellipsoids for confidence levels of 90%, 95%, and 99%, determined by the critical values of the chi-squared distribution with two degrees of freedom, i.e., $\chi_2^2(1-0.9)$, $\chi_2^2(1-0.95)$, and $\chi_2^2(1-0.99)$, respectively. Geometrically, the principal axes of the ellipses (included in the figure) correspond to the eigenvectors of the covariance matrix $\mathbf{\Sigma}_k$. The length of each axis is proportional to the square root of the corresponding eigenvalue of $\mathbf{\Sigma}_k$. These principal axes represent the directions of greatest and least variability in the output vector $\mathbf{Y}_k$, encapsulating the magnitude and orientation of uncertainty in the two components of the forecasted output. Larger eigenvalues indicate greater uncertainty along the corresponding eigenvector direction, while smaller eigenvalues indicate tighter confidence bounds. As the confidence level increases, the ellipsoids expand to encompass a larger region, reflecting the higher probability mass included within the specified confidence interval.

Confidence ellipsoids provide a probabilistic characterization of the uncertainty in the output vector $\mathbf{Y}_k$. Their construction accounts for both the variances of individual components and the correlations between them. This makes them particularly useful in applications where joint uncertainty needs to be quantified, such as in multivariate time series forecasting.

Projection to Confidence Intervals

The confidence ellipsoid offers a comprehensive depiction of uncertainty in the multivariate setting, capturing both variances and covariances of the components. However, in certain scenarios, it may be more practical to interpret individual confidence intervals for each component of $\mathbf{Y}_k$. These intervals are derived by projecting the ellipsoid onto each axis, effectively isolating the marginal distributions of the components. The confidence ellipsoid provides a comprehensive depiction of uncertainty in the multivariate setting, capturing both variances and covariances of the components. However, in certain scenarios, it may be more practical to interpret individual confidence intervals for each component of $\mathbf{Y}_k$. These intervals are derived by projecting the ellipsoid onto each axis, effectively isolating the marginal distributions of the components. For the i-th component of $\mathbf{Y}_k$, denoted by $Y_{k,i}$, a β-level confidence interval is given by:

$$\boxed{Y_{k,i} \in \left[\widetilde{y}_{k,i} - z_\beta \sigma_{k,i}, \quad \widetilde{y}_{k,i} + z_\beta \sigma_{k,i}\right],}$$

where $\widetilde{y}_{k,i}$ is the i-th element of $\widetilde{\mathbf{y}}_k$ (the mean of the predicted output vector), $\sigma_{k,i} = \sqrt{[\mathbf{\Sigma}_k]_{ii}}$ is the square root of the i-th diagonal element of $\mathbf{\Sigma}_k$ (the standard deviation of $Y_{k,i}$), and z_β is the critical value of the standard normal distribution corresponding to the confidence level β. For example, $z_{0.95} \approx 2$ for a 95% confidence level. These marginal confidence intervals provide a simpler and more interpretable measure of uncertainty for each individual component, complementing the richer but more complex confidence ellipsoid representation.

While these marginal intervals simplify the representation of uncertainty, they introduce significant limitations. Specifically:

- **Loss of Correlation Information:** Confidence intervals depend only on the variances (diagonal elements of $\mathbf{\Sigma}_k$) and neglect the covariances (off-diagonal elements). This omission disregards relationships and dependencies between components.
- **Misleading Uncertainty Representation:** For highly correlated components (see Fig. 1.8-right for an example), individual confidence intervals may suggest substantial uncertainty, even though the confidence ellipsoid reveals that the components are tightly constrained along specific directions due to their correlations.

Although individual confidence intervals are easier to interpret and useful for isolating uncertainties in single variables, they fail to capture the full scope of multivariate uncertainty. In contrast, the confidence ellipsoid accounts for both variances and covariances, offering a more accurate and holistic representation of uncertainty in $\mathbf{Y}_k$. This distinction is particularly crucial for understanding the joint behavior of the system's outputs, especially when correlations between components significantly influence the analysis.

End of Chapter. TBD: Hidden notes...

Exercises

1. (6 points) In this exercise, you will approximate a second-order ODE describing the dynamics of a pendulum using a discrete-time LSSM with two hidden states. The pendulum is governed by the following second-order ODE:

$$\frac{d^2\phi}{dt^2} + c\frac{d\phi}{dt} + \sin\phi = u(t)$$

where $\phi(t)$ is the angle of the pendulum from the vertical at time t, $u(t)$ is an external input torque applied to the pendulum, and c is a friction coefficient. Your goal is to transform this continuous-time system into a discrete-time state-space model using a small discretization step Δt. To do so, follow these steps:

   (a) **Define the State Variables**: To convert this second-order ODE into a first-order system, define the following state variables:

$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} \phi(t) \\ \omega(t) \end{bmatrix},$$

   where $\phi(t)$ is the angle of displacement and $\omega(t) = \frac{d\phi}{dt}$ is the angular velocity. Using this state variable, rewrite the pendulum equation as a system of two first-order ODEs:

$$\begin{cases} \frac{d\phi}{dt} = f_1(\omega, \phi, u(t)) = \omega \\ \frac{d\omega}{dt} = f_2(\omega, \phi, u(t)) = ? \end{cases}$$

   Find the explicit form of the function $f_2(\omega, \phi, u(t))$.

   (b) **Formulate the State Equations**: Now that we have expressed the system as a first-order system of ODEs, write it in the following state-space form:

$$\begin{bmatrix} \frac{d\phi}{dt} \\ \frac{d\omega}{dt} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{dx_1}{dt} \\ \frac{dx_2}{dt} \end{bmatrix}}_{\frac{d\mathbf{x}}{dt}} = \underbrace{\begin{bmatrix} f_1(x_2, x_1, u(t)) \\ f_2(x_2, x_1, u(t)) \end{bmatrix}}_{\mathbf{f}(\mathbf{x}, u(t))}.$$

   Find the explicit formula for the vector on the right-hand side.

   (c) **Discretize the Nonlinear System**: To transform this continuous-time system into a discrete-time model, use Euler's method with a discretization step Δt, i.e.,

$$\mathbf{x}(t + \Delta t) \approx \mathbf{x}(t) + \Delta t \cdot \mathbf{f}(\mathbf{x}(t), u(t)).$$

   Define the discrete-time state as $\mathbf{x}_k = \mathbf{x}(k \cdot \Delta t)$ and input $u_k = u(k \cdot \Delta t)$. The state update equation becomes:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t \cdot f(\mathbf{x}_k, u_k),$$

   which can be written as the following vector equality:

$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \end{bmatrix} = \begin{bmatrix} x_{1,k} \\ x_{2,k} \end{bmatrix} + \Delta t \cdot \begin{bmatrix} ? \\ ? \end{bmatrix},$$

   Fill in the entries of the last vector. Your answer should be a function of $x_{1,k}$, $x_{2,k}$, and u_k.

(d) Implement the resulting LSSM in Python and plot the evolution of x_1 as a function of k. In your simulations, set the initial angle to $x_{0,1} = \pi - 0.1$ (close to the vertical position) and the initial angular velocity to zero, i.e., $x_{0,2} = 0$. Set the inputs to be:

```python
# Parameters
delta_t = 0.01  # Time step
num_steps = 10000  # Number of time steps to simulate
c = 0.1  # Friction coefficient

# Sinusoidal input
amplitude = 0
frequency = 0.5
u = amplitude*np.sin(frequency*np.arange(num_steps))
```

Provide a physical interpretation of your observations.

(e) Using the same parameters, include a plot where the x-axis is $x_{k,1}$ and the y-axis is $x_{k,2}$. This type of plot is called a **phase space** plot. Provide a physical interpretation of your observations.

(f) In a pendulum, you can induce complex behavior by choosing the right set of parameters. For example, plot the phase space with the same initial conditions and the following set of parameters:

```python
# Parameters
delta_t = 0.01  # Time step
num_steps = 100000  # Number of time steps to simulate
c = 0.01  # Friction coefficient

# Sinusoidal input
amplitude = 1
frequency = 0.01
u = amplitude*np.sin(frequency*np.arange(num_steps))
```

What does the pendulum do in physical terms? Does it stabilize to an equilibrium point? Does it oscillate periodically? Does it oscillate irregularly?

2. (5 points) In this exercise, you will build a simple Hidden Markov Model (HMM) to model the operational states of a machine and detect potential faults based on observable signals. Suppose the machine can be in one of three hidden states at any given time: **Normal Operation** (State 1), **Minor Fault** (State 2), or **Severe Fault** (State 3). Although these states are hidden, they influence observable sensor readings. The sensor outputs one of three observable levels of vibration: **Low Vibration** (Observation 1), **Medium Vibration** (Observation 2), or **High Vibration** (Observation 3). Your goal is to analyze an HMM that represents this system, then simulate the model's response to a given observation sequence, and interpret the results. The initial distribution, state transition matrix, and emission matrix of the HMM are as follows:

$$\boldsymbol{\pi} = \begin{bmatrix} 0.8 & 0.15 & 0.05 \end{bmatrix}^\intercal, \; A = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.1 & 0.8 & 0.1 \\ 0.05 & 0.15 & 0.8 \end{bmatrix}, \; B = \begin{bmatrix} 0.9 & 0.08 & 0.02 \\ 0.3 & 0.6 & 0.1 \\ 0.1 & 0.3 & 0.6 \end{bmatrix}.$$

Follow the steps below to construct the model.

(a) **Simulate the Observation Sequence**: Using these HMM parameters, simulate the hidden states and corresponding observations over time. Write a function in `Python` that generates a sequence of hidden states and observations based on the transition and emission probabilities.

(b) **Decode the Observation Sequence**: Build a function in `Python` that, given an observation, identifies the most likely hidden state by (conditioned on the observation) finding the highest emission probability in the emission matrix for each state. For example, if you observe a high vibration level, the most likely hidden state would be **Severe Fault**, since that state has the highest probability of producing this observation.

(c) **Apply the Model to a Given Observation Sequence**: Using the following observation sequence:

```
[1, 1, 2, 3, 3, 2, 1, 1, 2, 3]
```

use your decoding function in `Python` to estimate the hidden state sequence that most likely produced these observations.

(d) **Probability of the Hidden State Sequence**: Using the HMM parameters, calculate in `Python` the probability of observing the hidden state sequence you estimated in the previous step. Use the initial state distribution ($\boldsymbol{\pi}_0$) and transition probabilities (A).

3. (5 points) Consider the following two LSSMs:

(a) The first LSSM is characterized by the state matrices (A, B, C, D) and the initial condition $\mathbf{x}_0$.

(b) The second LSSM is characterized by the state matrices $(\widetilde{A}, \widetilde{B}, \widetilde{C}, \widetilde{D})$ and the initial condition $\widetilde{\mathbf{x}}_0$, where the these matrices and initial state are defined as:

$$\widetilde{\mathbf{x}}_0 = T\mathbf{x}_0, \quad \widetilde{A} = TAT^{-1}, \quad \widetilde{B} = TB, \quad \widetilde{C} = CT^{-1}, \quad \widetilde{D} = D,$$

and T is an invertible matrix.

Prove that the two systems have the same input-output relationship for any sequence of inputs $(\mathbf{u}_k)_{k\geq 0}$. Specifically, show that the output $\mathbf{y}_k$ computed using (1.8) is identical for both systems.

4. (4 points) Consider an LSSM with a single hidden state, governed by the following equations:

$$x_{k+1} = x_k/2 + u_k,$$
$$y_k = x_k.$$

Answer the questions below:

(a) Starting with the initial condition $x_0 = 0$, derive an expression for the output y_k in terms of x_0 and the sequence of inputs $\{u_0, u_1, \ldots, u_{k-1}\}$.

(b) Suppose the input u_k is a step function, defined as $u_k = 1$ for all $k \geq 0$ and 0 otherwise. Using your answer from the previous question, derive an expression for the output sequence y_k under this step input.

(c) Compute the sequence of Markov parameters H_i for this system.

(d) Using the Markov parameters $\{H_0, H_1, H_2, \dots\}$ and the step function as the input sequence, compute the convolution of the Markov parameters with the input. Express the resulting sequence $\{y_0, y_1, y_2, \dots\}$ as a function of k.

Define a new state variable ξ_k by the invertible transformation $\xi_k = \frac{x_k}{2}$.

(e) Derive the transformed system matrices $\widetilde{A}, \widetilde{B}, \widetilde{C}, \widetilde{D}$ and the transformed initial condition ξ_0 for the new state-space model.

(f) Using the step function as an input, compute the output y_k of the transformed LSSM for $k = 0, 1, 2, \dots$

5. (3 points) Consider an LSSM model with system matrices (A, B, C, D). Prove the following identities using the component-wise definition of tensor Jacobians:

$$\frac{\partial \|\mathbf{y}_k - \widehat{\mathbf{y}}_k\|_2^2}{\partial \widehat{\mathbf{y}}_k} = 2(\widehat{\mathbf{y}}_k - \mathbf{y}_k)^\intercal, \ \frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} = C \text{ and } \frac{\partial \mathbf{x}_{j+1}}{\partial \mathbf{x}_j} = A \text{ for all } j \geq 1.$$

Hint: Remember that $\mathbf{y}_k$ are given constant vectors and $\widehat{\mathbf{y}}_k$ is the output of the LSSM; hence, $\widehat{\mathbf{y}}_k$ is a function of A.

6. (3 points) Consider 3 tensors of order 2: U, X and V (of compatible shapes). Prove that:

$$\left[\frac{\partial\,(UXV)}{\partial X}\right]_{i_1 i_2, j_1 j_2} = u_{i_1 j_1} v_{j_2 i_2}.$$

7. Consider an LSSM model with state matrix A. Prove that

$$\frac{\partial \mathbf{x}_{k+1}}{\partial A} = \mathbb{I}^{(2)} \otimes \mathbf{x}_k + A{:}\frac{\partial \mathbf{x}_k}{\partial A},$$

where ":" is the symbol for the contraction product using a single index.

8. Explain in your own words the vanishing and exploding gradient problem. What techniques have we covered in this chapter to alleviate this issue?

Chapter 2

Recurrent Neural Networks

In this section, we describe **Recurrent Neural Networks** (RNNs) as a nonlinear version of State-Space Models for modeling and forecasting time-series data. While LSSMs provide a convenient theoretical framework for modeling and analysis, they often fall short in capturing complex, nonlinear dependencies across time. RNNs build upon this foundation by introducing a nonlinearity into the hidden state evolution, enabling the modeling of a wide range of real-world systems where relationships between inputs, outputs, and states are inherently nonlinear. By incorporating nonlinear activation functions within the state-update equation, RNNs can represent a broader class of dynamical behaviors than their linear counterparts.

Building on the reader's understanding of LSSMs, we will first explore the structure and principles behind the standard RNN model. We will examine how RNNs generalize LSSMs by adapting familiar state-space notation to the RNN setting, thus grounding RNNs within a known framework. We then introduce and analyze the challenges associated with training RNNs. These include the issues of vanishing and exploding gradients, which can hinder the model's ability to learn long-term dependencies. In response to these issues, we present popular RNN variants, such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), both of which are specifically designed to improve training stability over long sequences.

2.0.1 RNNs as a Nonlinear Extension of LSSMs

As introduced in Chapter ?, Linear Time-Invariant (LTI) systems describe the dynamics of linear dynamical systems in terms of hidden states and observable outputs. The evolution of the hidden state is governed by the state-transition matrix A, which encapsulates the system's intrinsic dynamics, and the input matrix B, which defines the influence of external inputs $\mathbf{u}_k$ on the system:

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k + \mathbf{e}_k,$$

where $\mathbf{e}_k$ represents the process noise, capturing the stochastic variability inherent in the system's dynamics. In contrast, RNNs generalize LTIs by a introducing *nonlinear* transformation in the hidden states recursion. This transformation is governed by a nonlinear activation function σ, typically chosen as a *sigmoid* or *hyperbolic tangent* function. The state update equation for an RNN is therefore expressed as:

$$\boxed{\mathbf{x}_{k+1} = \sigma\left(A\mathbf{x}_k + B\mathbf{u}_k + \mathbf{b}\right),} \tag{2.1}$$

where **b** is a **bias term** for the hidden state. This term replaces the process noise present in the standard LTI framework, enabling deterministic updates while allowing the model to learn nonlinear representations of the underlying dynamics.

In an LTI system, the predicted output $\widehat{\mathbf{y}}_k$ is generated as:

$$\widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{n}_k,$$

where $\mathbf{n}_k$ represents the measurement noise. In an RNN, the predicted output $\widehat{\mathbf{y}}_k$ at each time step is modeled as:

$$\boxed{\widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{d},} \tag{2.2}$$

where **d** is an **output bias** term. Unlike the stochastic measurement noise term $\boldsymbol{\epsilon}_k$ in LTIs, the bias term **d** is a deterministic parameter learned during the training process, enabling the RNN to adjust the baseline level of the predicted output. This replacement highlights a key distinction between stochastic LTIs and the deterministic nature of RNNs.

2.0.2 Extensions of Vanilla RNNs

While vanilla Recurrent Neural Networks (RNNs) form the foundation of sequential modeling, several extensions have been developed to address specific challenges and enhance their modeling capabilities. This subsection presents some of the most widely used architectures that build upon the vanilla RNN framework, offering increased flexibility and improved performance in various applications.

Bidirectional RNNs

Vanilla RNNs process sequences in a strictly forward direction, propagating information from the past to the future. While this approach is effective for modeling strictly causal systems—where future states depend solely on past inputs and not vice versa—it has a fundamental limitation: *it cannot incorporate information from future time steps when predicting the current output*. The unidirectional flow of information in vanilla RNNs can lead to suboptimal performance in tasks where understanding the full **context** of a sequence is crucial, such as language processing tasks. Here, **context** refers to the entirety of relevant information within a sequence, including both past and future elements, that contributes to accurately modeling or predicting a given output. For example, in natural language processing, the meaning of a word often depends on both preceding and succeeding words in a sentence. Without access to future time steps, vanilla RNNs cannot fully capture such dependencies, limiting their effectiveness in these scenarios.

Bidirectional RNNs (BiRNNs) overcome this limitation by introducing a second vector of hidden states that processes the sequence in the reverse direction, from the future to the past. This design enables the model to capture information from both preceding and succeeding time steps at each point in the sequence. In a BiRNN, two sets of hidden states are computed for each time step:

- The **forward hidden state**, $\overrightarrow{\mathbf{x}}_k$, processes the input sequence in the forward direction, as typically done in vanilla RNNs:

$$\boxed{\overrightarrow{\mathbf{x}}_{k+1} = \sigma(\overrightarrow{A}\,\overrightarrow{\mathbf{x}}_k + \overrightarrow{B}\mathbf{u}_k + \overrightarrow{\mathbf{b}}),}$$

  where $\overrightarrow{A}$ and $\overrightarrow{B}$ are weight matrices, $\overrightarrow{\mathbf{b}}$ is a bias vector, $\mathbf{u}_k$ is the input at time step k, $\sigma(\cdot)$ is the activation function (e.g., tanh or ReLU), and the initial condition $\overrightarrow{\mathbf{x}}_0$ is a trainable parameter. This hidden state vector is identical to the one defined for the vanilla RNN and captures information from the past up to time k.

- In contrast, the **backward hidden state**, $\overleftarrow{\mathbf{x}}_k$, processes the sequence in reverse order:

$$\boxed{\overleftarrow{\mathbf{x}}_{k-1} = \sigma\left(\overleftarrow{A}\,\overleftarrow{\mathbf{x}}_k + \overleftarrow{B}\mathbf{u}_k + \overleftarrow{\mathbf{b}}\right),}$$

  where $\overleftarrow{\mathbf{x}}_{k-1}$ depends on the current hidden state $\overleftarrow{\mathbf{x}}_k$. In this case, the final condition $\overrightarrow{\mathbf{x}}_L$ is a trainable parameter, where L is the length of the time series. This state captures information from the future down to time k.

By maintaining both forward and backward hidden states, BiRNNs effectively encode bidirectional dependencies within the sequence. The forward and backward hidden states are combined to compute the output at each time step:

$$\boxed{\widehat{\mathbf{y}}_k = C\begin{bmatrix}\overrightarrow{\mathbf{x}}_k\\ \overleftarrow{\mathbf{x}}_k\end{bmatrix} + D\mathbf{u}_k + \mathbf{d},}$$

where C and D are the output weight matrices, and $\mathbf{d}$ is the bias vector. This structure ensures that the output at each time step incorporates information from the entire sequence, providing a richer representation.

The main idea behind BiRNNs is to leverage the full context of a sequence for each prediction. For example, in a language processing task, the meaning of a word may depend on both preceding and succeeding words. Similarly, in time-series forecasting, understanding the current state may require knowledge of both past trends and future events. By combining forward and backward hidden states, BiRNNs effectively capture these dependencies, leading to improved performance in tasks that require holistic sequence understanding.

Stacked RNNs (Deep RNNs)

Stacked RNNs, also referred to as Deep RNNs, extend the standard RNN architecture by layering multiple RNNs on top of each other. This hierarchical structure enables the model to capture temporal patterns at different levels of abstraction. In this architecture, the hidden states of one layer serve as inputs to the subsequent layer, facilitating a richer representation of the input sequence.

For an L-layer Stacked RNN, the hidden states are updated as follows:

1. At the first layer ($l = 1$), the hidden state depends directly on the input sequence:

$$\boxed{\mathbf{x}_k^{(1)} = \sigma\left(A^{(1)}\mathbf{x}_{k-1}^{(1)} + B^{(1)}\mathbf{u}_{k-1} + \mathbf{b}^{(1)}\right),}$$

   where $A^{(1)}$ and $B^{(1)}$ are weight matrices, $\mathbf{b}^{(1)}$ is a bias vector, $\mathbf{u}_k$ is the input at time step k, and $\sigma(\cdot)$ is the activation function.

2. For subsequent layers ($l = 2, \ldots, L$), the hidden state at time step k is computed using the hidden state of the same layer at the previous time step ($k-1$) and the hidden state of the layer directly below it ($l-1$):

$$\boxed{\mathbf{x}_k^{(l)} = \sigma\left(A^{(l)}\mathbf{x}_{k-1}^{(l)} + B^{(l)}\mathbf{x}_k^{(l-1)} + \mathbf{b}^{(l)}\right).}$$

Insert diagram.

The output at each time step k is computed from the hidden state of the topmost layer ($l = L$):

$$\boxed{\widehat{\mathbf{y}}_k = C\mathbf{x}_k^{(L)} + D\mathbf{u}_k + \mathbf{d},}$$

where C and D are the output weight matrices, and $\mathbf{d}$ is the output bias. This stacked architecture enables the model to learn both low-level and high-level temporal features, making it particularly effective for complex time-series data and sequential tasks.

Stacked RNNs share conceptual similarities with Neural-Network State-Space Models (NNSSMs), but they exhibit notable differences due to their distinct architectures and design principles. On the one hand, in Stacked RNNs, the hidden states evolve through recurrent updates governed by nonlinear activation functions, while NNSSMs employ a learned state-space equation to explicitly model the system dynamics, with a clear separation between the state-update and output equations. On the other hand, Stacked RNNs are designed to learn end-to-end mappings from inputs to outputs across multiple layers, emphasizing hierarchical feature extraction, while NNSSMs incorporate domain-specific insights by explicitly defining state and observation equations, making them more interpretable and often better suited for modeling systems with known physical or dynamical structure. This distinction makes Stacked RNNs particularly effective for problems where hierarchical temporal patterns are critical and the underlying dynamics are not explicitly known, such as natural language processing and speech recognition. NNSSMs, by contrast, excel in scenarios where a principled understanding of the system's behavior is necessary, such as control and forecasting in engineering or physics applications.

Clockwork RNNs

Clockwork RNNs introduce a novel approach to recurrent neural networks by partitioning the hidden state into distinct groups, each operating at a specific temporal scale. In many real-world applications, sequences exhibit distinct dynamics at different temporal scales. For example, power load data or financial data shows clear daily, weekly, and yearly cycles due to human activity patterns and seasonal variations. Clockwork RNNs are designed to efficiently capture such hierarchical temporal patterns by assigning distinct update frequencies to different groups of hidden states.

In a clockwork RNN, the hidden state vector is divided into G groups:

$$\boxed{\mathbf{x}_k = [\mathbf{x}_k^{(1)}; \mathbf{x}_k^{(2)}; \ldots, \mathbf{x}_k^{(G)}].}$$

Each group g is assigned a clock period τ_g, which determines the time steps at which the state vector $\mathbf{x}_k^{(g)}$ for that group updates. Specifically, group g updates its state only when $k \bmod \tau_g = 0$. Groups with smaller τ_g update more frequently, capturing fast-varying dynamics (e.g., daily cycles in power load data). In contrast, groups with larger τ_g update less frequently, focusing on slower-changing trends (e.g., weekly or yearly variations).

At time step k, only the *active groups*, i.e., those satisfying $(k \bmod \tau_g) = 0$, update their states. For an active group g, the state update equation is given by:

$$\boxed{\mathbf{x}_k^{(g)} = \sigma\left(A^{(g)}\mathbf{x}_{k-1}^{(g)} + B^{(g)}\mathbf{u}_{k-1} + \mathbf{b}^{(g)}\right), \quad \text{if } (k \bmod \tau_g) = 0.}$$

where $A^{(g)}$, $B^{(g)}$, and $\mathbf{b}^{(g)}$ are the weight matrices and bias vector specific to group g. For instance, if group g corresponds to weekly updates, its update at a given time step k satisfying $k \bmod \tau_g = 0$; hence, the input $\mathbf{u}_k$ reflects patterns at the weekly timescale. Conversely, *inactive groups*, for which $(k \bmod \tau_g) \neq 0$, retain their previous state. The state of an inactive group remains unchanged, i.e.,

$$\boxed{\mathbf{x}_k^{(g)} = \mathbf{x}_{k-1}^{(g)}, \quad \text{if } (k \bmod \tau_g) \neq 0.}$$

This mechanism allows Clockwork RNNs to efficiently allocate computational resources and focus on capturing patterns at multiple timescales, seamlessly combining fast-varying and slowly evolving dynamics within a unified architecture. The output at time step k combines the contributions of all groups, as follows:

$$\boxed{\hat{\mathbf{y}}_k = \sum_{g=1}^{G} C^{(g)}\mathbf{x}_k^{(g)} + D\mathbf{u}_k + \mathbf{d},}$$

where $C^{(g)}$ and D are output weight matrices, and $\mathbf{d}$ is the bias vector.

Clockwork RNNs offer several key benefits. First, by assigning groups different update frequencies, they efficiently capture both fast and slow dynamics within a single model. Second, groups with larger update intervals (τ_g) compute less frequently, significantly reducing computational complexity at each time step. Additionally, the partitioned hidden states and group-specific update mechanisms enhance interpretability, providing valuable insights into how the model processes and prioritizes temporal information. As a consequence, Clockwork RNNs provide an elegant framework for combining multiple timescales into a single recurrent architecture, balancing expressiveness and computational efficiency.

2.1 Training RNNs for Time-Series Forecasting

Training RNNs involves the use of *Backpropagation Through Time* (BPTT), an iterative optimization algorithm designed to adjust the network parameters to minimize the discrepancy between the predicted output sequence $(\widehat{\mathbf{y}}_k)_{k=0}^{K}$ generated by the RNN and the desired output sequences $(\mathbf{y}_k)_{k=0}^{K}$, given a particular input sequence $(\mathbf{u}_k)_{k=0}^{K}$. Our discussion of BPTT relies heavily on **tensor algebra** (refer to Appendix A for a review of tensor notation, which is essential for this section).

BPTT is a fundamental method for computing gradients in sequential models, such as LSSMs and RNNs, by unrolling the temporal dependencies inherent in their architecture. The goal is to estimate the set of parameters that define an RNN, i.e., the weight matrices A, B, C, and D, the bias vectors $\mathbf{b}$ and $\mathbf{d}$, and the initial hidden state $\mathbf{x}_0$. In practical implementations, these parameters are often collectively represented as a third-order tensor, $\boldsymbol{\theta}$, to facilitate efficient tensor-based computation. This tensorial representation is achieved through techniques such as **padding** and **stacking**, where each matrix (or vector) in the parameter set is first 'padded' with zeros to ensure consistent dimensions and then 'stacked' into a tensor of order 3. This approach ensures scalability and computational efficiency by leveraging GPUs for parallel computation. (Further details on the implementation of stacking and padding steps using `PyTorch` are provided in Appendix **??**.)

(L to K in next version; same in the other chapters...) The parameters of an RNN are trained to minimize a suitable loss function, often the *mean squared error* (MSE) defined as:

$$\boxed{\mathcal{L}(\boldsymbol{\theta}; \mathbf{u}_{0:L}, \mathbf{y}_{0:L}) = \frac{1}{L+1}\sum_{k=0}^{L} \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k) = \frac{1}{L+1}\sum_{k=0}^{L} \|\mathbf{y}_k - \widehat{\mathbf{y}}_k(\boldsymbol{\theta}; \mathbf{u}_{0:L})\|_2^2,} \tag{2.3}$$

where $\mathbf{u}_{0:L}$ and $\mathbf{y}_{0:L}$ denote the (known) input and output sequences, respectively. The term $\widehat{\mathbf{y}}_k(\boldsymbol{\theta}; \mathbf{u}_{0:L})$ represents the model's predicted output at time k, which is determined by the input sequence $\mathbf{u}_{0:L}$ and the current parameter values $\boldsymbol{\theta}$, as defined by the equations that define the RNN dynamics, (2.1) and (2.5).

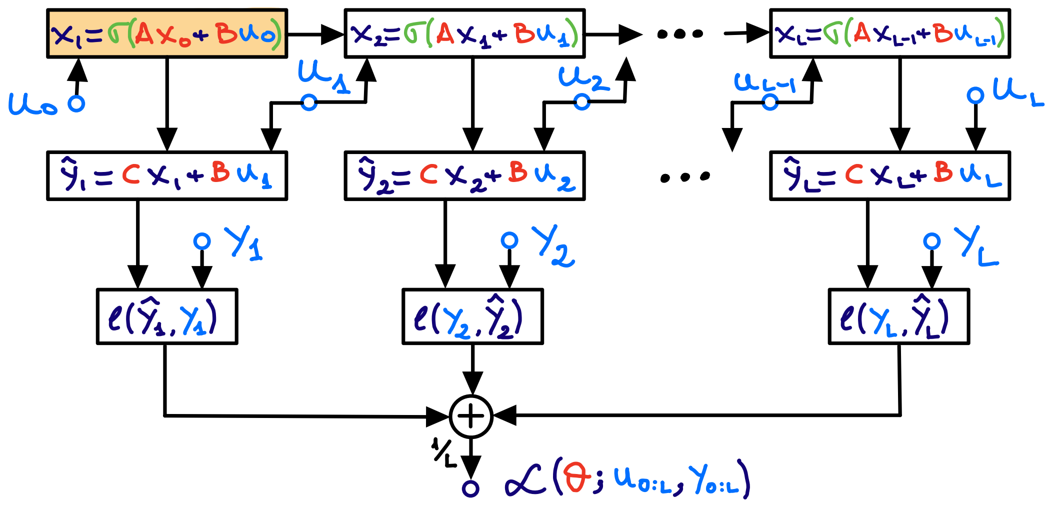
By iteratively updating the parameter values $\boldsymbol{\theta}$ to minimize $\mathcal{L}$ using gradient-based optimization methods, the RNN is trained to capture the temporal dependencies inherent in the input sequence. This process progressively refines the model parameters, enhancing its ability to generalize and improving forecasting accuracy, ultimately enabling reliable predictions on unseen data.

2.1.1 Backpropagation Through Time (BPTT)

Backpropagation Through Time (BPTT) is an adaptation of the classical backpropagation algorithm for sequential data. Unlike traditional feedforward networks (such as the multi-layer perceptron), where gradients are computed in a single pass, BPTT propagates gradients backward through each step of the computational recursion in (2.1) and (2.5). This process, known as **unrolling**, conceptually transforms the RNN into a deep network, with each layer corresponding to a specific step in the recursion.

The result of unrolling a recursive architecture can be represented as a **computational graph**, which explicitly illustrates the dependencies between hidden states and outputs.

This graph is a **Directed Acyclic Graph**[1] (DAG) that illustrates how information propagates forward through the model and how gradients flow backward during BPTT. The structure of the computational graph induced by the RNN equations is shown in Fig. 2.1, where each box represents a computational step. For clarity, we have colored in red those parameters that we aim to learn, and in blue those values that are known. According to this graph, we start by computing the first hidden state vector $\mathbf{x}_1$ using the current estimate of the trainable parameters and the given input. Then, we traverse the computational graph to compute the sequence of hidden states, $(\mathbf{x}_k)_{k=1}^{L}$, which is subsequently used to compute the sequence of outputs, $(\widetilde{\mathbf{y}}_k)_{k=1}^{L}$, the (local) loss functions, $\ell(\mathbf{y}_k, \widetilde{\mathbf{y}}_k)$, and the total loss function $\mathcal{L}$.



$x_1 = \sigma(Ax_0 + Bu_0)$
$x_2 = \sigma(Ax_1 + Bu_1)$
$x_L = \sigma(Ax_{L-1} + Bu_{L-1})$
u_0
u_1
u_2
u_{L-1}
u_L
$\hat{y}_1 = Cx_1 + Bu_1$
$\hat{y}_2 = Cx_2 + Bu_2$
$\hat{y}_L = Cx_L + Bu_L$
y_1
y_2
y_L
$\ell(\hat{y}_1, y_1)$
$\ell(y_2, \hat{y}_2)$
$\ell(y_L, \hat{y}_L)$
$1/L$
$\mathcal{L}(\theta; u_{0:L}, y_{0:L})$

Figure 2.1: Computational graph of a Recurrent Neural Network (RNN). The graph illustrates the flow of information, starting from the root node (marked in orange) and progressing through the sequence of hidden states $(\mathbf{x}_k)_{k=1}^{L}$. The graph also shows the computation of the outputs $(\widehat{\mathbf{y}}_k)_{k=1}^{L}$, the local loss functions $\ell(\mathbf{y}_k, \widetilde{\mathbf{y}}_k)$, and the total loss function $\mathcal{L}$. Parameters to be learned are highlighted in red, while known values are shown in blue.

The BPTT algorithm consists of three primary steps:

1. **Forward Pass:** During the forward pass, the RNN traverses the computational graph sequentially, calculating the hidden states, predicted outputs, and the total loss function.
2. **Backward Pass:** In the backward pass, Backpropagation Through Time (BPTT) computes the gradients of the loss function with respect to each model parameter, explicitly capturing the temporal dependencies introduced by the nonlinear activations at each time step.
3. **Gradient Update:** Finally, in the gradient update step, the parameters are iteratively refined to minimize the loss function.

[1] A Directed Acyclic Graph (DAG) is a graph with directed edges and no cycles, ensuring a partial ordering of its nodes.

This section provides a detailed exploration of these steps within the RNN framework.

Forward Pass (Prediction Update)

In the forward pass of an RNN, we use the current estimates of the parameters—the system matrices $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{p\times n}$, $D \in \mathbb{R}^{p\times m}$, the biases $\mathbf{b} \in \mathbb{R}^n$ and $\mathbf{d} \in \mathbb{R}^p$, and the initial hidden state $\mathbf{x}_0 \in \mathbb{R}^n$—to generate a sequence of hidden states $(\mathbf{x}_k)_{k=0}^L$ and predictions $(\widehat{\mathbf{y}}_k)_{k=0}^L$. These computations are governed by the RNN's state and output equations, reproduced below for convenience:

$$\mathbf{x}_k = \sigma\left(A\mathbf{x}_{k-1} + B\mathbf{u}_{k-1} + \mathbf{b}\right), \tag{2.4}$$

$$\widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{d}. \tag{2.5}$$

These equations highlight the RNN's ability to model nonlinear dynamics by applying the activation function $\sigma(\cdot)$ at each step, enabling it to capture complex temporal dependencies. The results of the forward pass form the foundation for computing a collection of **Jacobian tensors**, which are essential for evaluating the gradient of the loss function with respect to the parameters during the backward pass.

Backward Pass (Gradient Computation)

The objective of the backward pass is to compute the gradients of the total loss function $\mathcal{L}$ with respect to the parameters in $\boldsymbol{\theta}$. The total loss function is reproduced below for convenience:

$$\mathcal{L}(\boldsymbol{\theta}; \mathbf{u}_{0:L}, \mathbf{y}_{0:L}) = \frac{1}{L+1}\sum_{k=0}^{L} \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k(\boldsymbol{\theta}; \mathbf{u}_{0:L})).$$

In the following subsections, we demonstrate how to compute these gradients using different variations of the chain rule.

Gradients w.r.t. A, B, and $\mathbf{b}$

The computational graph in Fig. 2.1 illustrates that the hidden state at each time step, $\mathbf{x}_k$, depends on the previous hidden state $\mathbf{x}_{k-1}$, the input $\mathbf{u}_{k-1}$, and the parameters A, B, and $\mathbf{b}$. This layered dependency creates a recursive relationship in which each hidden state influences the next, necessitating the computation of partial derivatives across multiple layers of dependencies. The computational graph also clearly demonstrates how the parameters A, B, and $\mathbf{b}$ impact the total loss function $\mathcal{L}$ through their role in generating the sequence of hidden states $\mathbf{x}_k$ and the outputs $\widehat{\mathbf{y}}_k$. Based on this structure, we apply the following version of the chain rule to compute the gradients with respect to A, B, and $\mathbf{b}$:

$$\boxed{\frac{\partial \mathcal{L}(\boldsymbol{\theta}; \mathbf{u}_{0:L}, \mathbf{y}_{0:L})}{\partial \boldsymbol{\theta}} = \frac{1}{L+1}\sum_{k=0}^{L} \frac{\partial \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} : \frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} : \frac{\partial \mathbf{x}_k}{\partial \boldsymbol{\theta}},} \tag{2.6}$$

where ":" denotes the tensor contraction product (see Appendix **??** for a definition). This equation provides a systematic method for computing these gradients by decomposing the dependencies into manageable steps, leveraging the structure of the computational graph for efficient backpropagation.

The Jacobians on the right-hand side of (2.6) are tensors of order 1, 2, and 4, respectively. The tensor contraction ":" ensures that gradients are propagated correctly, preserving the dependencies introduced by the nonlinear activations and recurrent connections. This formulation highlights how the RNN architecture transforms parameter gradients into a chain of sequential operations, explicitly accounting for the temporal and structural dependencies within the computational graph.

Next, we analyze each Jacobian in the chain of contraction products in (2.6):

- The gradient of the (point-wise) loss function $\ell(\cdot)$ with respect to $\widehat{\mathbf{y}}_k$, and the Jacobian of $\widehat{\mathbf{y}}_k$ with respect to $\mathbf{x}_k$, are given by:

$$\boxed{\frac{\partial \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} = 2(\widehat{\mathbf{y}}_k - \mathbf{y}_k)} \quad \text{and} \quad \boxed{\frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} = C.} \tag{2.7}$$

- To compute the Jacobian of the hidden state $\mathbf{x}_k$ with respect to A, B, and $\mathbf{b}$, we must account for the nonlinearity introduced by the activation function σ. By computing the partial derivatives of both sides of (2.4), we obtain the following recursive expression for the gradient with respect to the first tensorial slice, A (see Appendix B for a detailed proof):

$$\boxed{\frac{\partial \mathbf{x}_k}{\partial A} = \Sigma_{k-1}^{(3)} \odot \left(\mathbb{I}^{(2)} \otimes \mathbf{x}_{k-1} + A : \frac{\partial \mathbf{x}_{k-1}}{\partial A} \right),} \tag{2.8}$$

where $\Sigma_k^{(3)}$ is a tensor of order 3, defined component-wise as:

$$[\Sigma_k^{(3)}]_{i,j_1,j_2} = \sigma' \left([A\mathbf{x}_k + B\mathbf{u}_k + \mathbf{b}]_i \right),$$

and σ' is the derivative of the activation function. In this equation $\mathbb{I}^{(2)}$ denotes the identity tensor of order 2, $\otimes$ represents the Kronecker tensor product, and $\odot$ is the Hadamard (element-wise) tensor product. The Hadamard product in (2.8) enables element-wise **broadcasting** of the activation derivatives across the Jacobian tensors. This ensures that the nonlinear effects introduced by the activation function are properly incorporated into the gradients with respect to the model parameters.

Equation (2.8) defines a recursion to compute $\partial \mathbf{x}_k / \partial A$ from $\partial \mathbf{x}_{k-1} / \partial A$. Starting with the initial condition $\partial \mathbf{x}_0 / \partial A = \mathbb{O}^{(3)}$, where $\mathbb{O}^{(3)}$ denotes a tensor of order 3 filled with zeros, we can iteratively solve the recursion to obtain the sequence of 3-dimensional Jacobian tensors, $(\partial \mathbf{x}_{k+1} / \partial A)_{k=0}^{L-1}$.

- The Jacobian $\partial \mathbf{x}_k / \partial B$ satisfies the following equation (proof left as an exercise):

$$\boxed{\frac{\partial \mathbf{x}_k}{\partial B} = \Sigma_{k-1}^{(3)} \odot \left(\mathbb{I}^{(2)} \otimes \mathbf{u}_{k-1} + A : \frac{\partial \mathbf{x}_{k-1}}{\partial B} \right).} \tag{2.9}$$

This equation defines a recursion that allows us to compute the sequence of Jacobian tensors $(\partial \mathbf{x}_k/\partial B)_{k=1}^L$. The recursion begins with the initial condition $\partial \mathbf{x}_0/\partial B = \mathbb{O}^{(3)}$, where $\mathbb{O}^{(3)}$ is a tensor of order 3 filled with zeros.

- The Jacobian $\partial \mathbf{x}_k/\partial \mathbf{b}$ satisfies the following equation (proof left as an exercise):

$$\boxed{\frac{\partial \mathbf{x}_k}{\partial \mathbf{b}} = \Sigma_{k-1}^{(2)} \odot \left(\mathbb{I}^{(2)} + A{:}\frac{\partial \mathbf{x}_{k-1}}{\partial \mathbf{b}}\right),} \tag{2.10}$$

where $[\Sigma_k^{(2)}]_{i,j} = \sigma'\left([A\mathbf{x}_k + B\mathbf{u}_k + \mathbf{b}]_i\right)$.

Substituting (2.7) in (2.6), and using the Jacobians obtained in (2.8), (2.9), and (2.10), we can compute the gradients $\partial\mathcal{L}/\partial A$, $\partial\mathcal{L}/\partial B$, and $\partial\mathcal{L}/\partial \mathbf{b}$.

Gradients with Respect to C, D, and $\mathbf{d}$

To compute the gradients $\partial\mathcal{L}/\partial C$, $\partial\mathcal{L}/\partial D$, and $\partial\mathcal{L}/\partial \mathbf{d}$, we use a slightly different version of the chain rule:

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta}; \mathbf{y}_{0:L})}{\partial \boldsymbol{\theta}} = \frac{1}{L+1}\sum_{k=0}^{L} \frac{\partial \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} {:} \frac{\partial \widehat{\mathbf{y}}_k}{\partial \boldsymbol{\theta}}. \tag{2.11}$$

The first factor in the chain rule above was already derived in (2.7). The second factor, $\partial\widehat{\mathbf{y}}_k/\partial\boldsymbol{\theta}$, is a tensor of order 4. The tensorial slices corresponding to C, D, and $\mathbf{d}$ can be derived from the output equation $\widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{d}$ (proof left as an exercise):

$$\frac{\partial \widehat{\mathbf{y}}_k}{\partial C} = \mathbb{I}^{(2)} \otimes \mathbf{x}_k, \quad \frac{\partial \widehat{\mathbf{y}}_k}{\partial D} = \mathbb{I}^{(2)} \otimes \mathbf{u}_k, \quad \frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{d}} = \mathbb{I}^{(2)}.$$

These expressions hold because $\mathbf{x}_k$ and $\mathbf{u}_k$ are independent of C, D, and $\mathbf{d}$.

Substituting these results into (2.11), we obtain the gradients of the total loss function with respect to C, D, and $\mathbf{d}$:

$$\boxed{\frac{\partial \mathcal{L}}{\partial C} = \frac{2}{L+1}\sum_{k=0}^{L}(\widehat{\mathbf{y}}_k - \mathbf{y}_k)\otimes \mathbf{x}_k,} \quad \boxed{\frac{\partial \mathcal{L}}{\partial D} = \frac{2}{L+1}\sum_{k=0}^{L}(\widehat{\mathbf{y}}_k - \mathbf{y}_k)\otimes \mathbf{u}_k,}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{d}} = \frac{2}{L+1}\sum_{k=0}^{L}(\widehat{\mathbf{y}}_k - \mathbf{y}_k).}$$

These expressions reveal that the gradients with respect to C and D are obtained by summing the outer products of the residuals $(\widehat{\mathbf{y}}_k - \mathbf{y}_k)$ with the hidden states $\mathbf{x}_k$ and inputs $\mathbf{u}_k$, respectively, over all time steps k. Similarly, the gradient with respect to $\mathbf{d}$ is computed by summing the residuals directly.

Gradient with Respect to $\mathbf{x}_0$

The final tensorial slice of the gradient, $\partial\mathcal{L}/\partial\mathbf{x}_0$, is computed using the following chain rule:

$$\frac{\partial\mathcal{L}(\boldsymbol{\theta};\mathbf{y}_{0:L})}{\partial\mathbf{x}_0}=\frac{1}{L+1}\sum_{k=0}^{L}\frac{\partial\ell(\mathbf{y}_k,\widehat{\mathbf{y}}_k)}{\partial\widehat{\mathbf{y}}_k}:\frac{\partial\widehat{\mathbf{y}}_k}{\partial\mathbf{x}_k}:\frac{\partial\mathbf{x}_k}{\partial\mathbf{x}_{k-1}}:\cdots:\frac{\partial\mathbf{x}_1}{\partial\mathbf{x}_0}.$$

The first two factors in this chain rule were derived in (2.7). The remaining k factors, corresponding to $\partial\mathbf{x}_\kappa/\partial\mathbf{x}_{\kappa-1}$, have the following uniform expression:

$$\frac{\partial\mathbf{x}_\kappa}{\partial\mathbf{x}_{\kappa-1}}=\Sigma_\kappa^{(2)}\odot A,\quad\text{for all }\kappa\in[k],$$

because A, B, and $\mathbf{u}_{\kappa-1}$ are independent of $\mathbf{x}_{\kappa-1}$.

Substituting all factors into the chain rule yields the following expression for the gradient:

$$\boxed{\frac{\partial\mathcal{L}(\boldsymbol{\theta};\mathbf{y}_{0:L})}{\partial\mathbf{x}_0}=\frac{2}{L+1}\sum_{k=0}^{L}(\widehat{\mathbf{y}}_k-\mathbf{y}_k)^\intercal\cdot C\cdot\prod_{\kappa=0}^{k-1}\left(\Sigma_\kappa^{(2)}\odot A\right),}\tag{2.12}$$

where the last product refers to a chain of tensor contractions:

$$\prod_{\kappa=0}^{k-1}\left(\Sigma_\kappa^{(2)}\odot A\right)=\left(\Sigma_0^{(2)}\odot A\right):\left(\Sigma_1^{(2)}\odot A\right):\cdots:\left(\Sigma_{k-1}^{(2)}\odot A\right),$$

in that order. This ensures the correct propagation of gradients through the computational graph, preserving the temporal dependencies introduced by the RNN structure.

This expression highlights how the gradient of the loss function with respect to the initial state $\mathbf{x}_0$ is influenced by both the residuals $(\widehat{\mathbf{y}}_k-\mathbf{y}_k)$ and the recurrent dynamics of the RNN. These dynamics are captured by the product of the Jacobian slices $(\Sigma_\kappa^{(2)}\odot A)$ over all intermediate time steps κ.

To elucidate these equations and provide a tangible illustration, we now consider a simplified example of an RNN with a scalar input, a scalar output, and a single hidden state. This minimalistic setup allows us to examine the gradient computation process without the complexity of tensors, offering an intuitive understanding of the underlying mechanics.

Example 3: Gradient Computation in a Simple RNN

Let us consider an RNN with a scalar input u_k, a scalar output y_k, and a single hidden state x_k. The RNN dynamics are given by the following (scalar) expressions:

$$x_{k+1}=\sigma(ax_k+bu_k+\beta),\quad\widehat{y}_k=cx_k+du_k+\delta,$$

where a,b,c,d are the system scalars, β and δ are the bias scalars, x_0 is the initial hidden state, and $\sigma(\cdot)$ is a sigmoid activation function. The (local) loss function at each time step k is defined as:

$$\ell(y_k,\widehat{y}_k)=(y_k-\widehat{y}_k)^2.$$

Our objective is to compute the gradient of the total loss function:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \left[\frac{\partial \mathcal{L}}{\partial a}, \frac{\partial \mathcal{L}}{\partial b}, \frac{\partial \mathcal{L}}{\partial \beta}, \frac{\partial \mathcal{L}}{\partial c}, \frac{\partial \mathcal{L}}{\partial d}, \frac{\partial \mathcal{L}}{\partial \delta}, \frac{\partial \mathcal{L}}{\partial x_0}\right] = \frac{1}{L+1} \sum_{k=0}^{L} \frac{\partial \ell(y_k, \widehat{y}_k)}{\partial \boldsymbol{\theta}}.$$

In this case, two of the dimensions of the 3-dimensional tensor gradient $\partial \mathcal{L}/\partial \boldsymbol{\theta}$ reduce to size one. Consequently, the tensor is *squeezed* into a 1-dimensional tensor, simplifying its representation and analysis.
In our derivations, we will use the sigmoid activation function:

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad \Rightarrow \quad \sigma'(z) = \frac{d\,\sigma(z)}{dz} = \sigma(z)\,(1-\sigma(z)).$$

Since the state equation in the RNN can be written as $x_k = \sigma(z_{k-1})$, with $z_{k-1} = a x_{k-1} + b u_{k-1} + \delta$, we have:

$$\sigma'(z_{k-1}) = \sigma(z_{k-1})\,(1-\sigma(z_{k-1})) = x_k\,(1-x_k) = \left(x_k - x_k^2\right).$$

This identity will be useful in the forthcoming derivations.

Partials w.r.t. a, b, and β. To compute the partial derivatives with respect to a, b and β (i.e. the parameters in the hidden state recursion), we use the following chain rule:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{1}{L+1} \sum_{k=0}^{L} \frac{\partial \ell(y_k, \widehat{y}_k)}{\partial \widehat{y}_k} \cdot \frac{\partial \widehat{y}_k}{\partial x_k} \cdot \frac{\partial x_k}{\partial \theta}, \quad \text{for } \theta \in \{a, b, \beta\}. \tag{2.13}$$

We now analyze each of the factors in this equation:

1. **Partial Derivative of the Local Square Error:**

$$\frac{\partial \ell(y_k, \widehat{y}_k)}{\partial \widehat{y}_k} = \frac{\partial (y_k - \widehat{y}_k)^2}{\partial \widehat{y}_k} = 2(\widehat{y}_k - y_k). \tag{2.14}$$

2. **Partial Derivative of the Estimated Output:** Using the output equation $\widehat{y}_k = c x_k + d u_k$, the partial derivative with respect to x_k is:

$$\frac{\partial \widehat{y}_k}{\partial x_k} = c, \tag{2.15}$$

where c is independent of x_k.

3. **Partial Derivative of the Hidden State:** Using the state recursion equation of the RNN, the partial derivatives of x_k with respect to a, b, and β are:

$$\frac{\partial x_k}{\partial a} = \sigma'_{k-1} \cdot \left(x_{k-1} + a \frac{\partial x_{k-1}}{\partial a}\right) = \left(x_k - x_k^2\right) \cdot \left(x_{k-1} + a \frac{\partial x_{k-1}}{\partial a}\right),$$

$$\frac{\partial x_k}{\partial b} = \sigma'_{k-1} \cdot \left(u_{k-1} + a \frac{\partial x_{k-1}}{\partial b}\right) = \left(x_k - x_k^2\right) \cdot \left(u_{k-1} + a \frac{\partial x_{k-1}}{\partial b}\right),$$

307

$$\frac{\partial x_k}{\partial \beta} = \sigma'_{k-1} \cdot \left(1 + a\frac{\partial x_{k-1}}{\partial \beta}\right) = \left(x_k - x_k^2\right) \cdot \left(1 + a\frac{\partial x_{k-1}}{\partial \beta}\right),$$

where $\sigma'_{k-1} = \sigma'(a\,x_{k-1} + b\,u_{k-1} + \beta)$. Notice that these equations define recursions for the derivatives $\partial x_k/\partial a$, $\partial x_k/\partial b$, and $\partial x_k/\partial \beta$. These recursions can be solved iteratively, starting with the initial conditions:

$$\frac{\partial x_0}{\partial a} = \frac{\partial x_0}{\partial b} = \frac{\partial x_0}{\partial \beta} = 0.$$

This process generates the sequences $(\partial x_k/\partial a)_{k=0}^{L}$, $(\partial x_k/\partial b)_{k=0}^{L}$, and $(\partial x_k/\partial \beta)_{k=0}^{L}$. Substituting these sequences into (2.13)—as well all (2.14) and (2.15)—we obtain expressions for the partial derivatives of the loss function with respect to the parameters a, b, and β, i.e., the first 3 components of $\partial \mathcal{L}/\partial \boldsymbol{\theta}$.

Partials w.r.t. c, d, and δ. To compute the partial derivatives of the total loss $\mathcal{L}$ with respect to c, d, and δ (i.e., the parameters in the output equation), we apply the following chain rule:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{1}{L+1}\sum_{k=0}^{L} \frac{\partial \ell(y_k, \widehat{y}_k)}{\partial \widehat{y}_k} \cdot \frac{\partial \widehat{y}_k}{\partial \theta}, \quad \text{for } \theta \in \{c, d, \delta\}. \tag{2.16}$$

From the output equation of the RNN, $\widehat{y}_k = cx_k + du_k + \delta$, we can trivially derive:

$$\frac{\partial \widehat{y}_k}{\partial c} = x_k, \quad \frac{\partial \widehat{y}_k}{\partial d} = u_k, \quad \frac{\partial \widehat{y}_k}{\partial \delta} = 1.$$

In these partial derivatives, we must take into account that x_k does not depend on the parameters c, d, or δ (as we can observe in the computational graph in Fig. 2.1). Substituting these derivatives into (2.16), we obtain the following components of the gradient: $\partial \mathcal{L}/\partial c$, $\partial \mathcal{L}/\partial d$, and $\partial \mathcal{L}/\partial \delta$.

Partials w.r.t. x_0. To compute the last component of the gradient $\partial \mathcal{L}/\partial \boldsymbol{\theta}$, we apply the following chain rule:

$$\frac{\partial \mathcal{L}}{\partial x_0} = \frac{1}{L+1}\sum_{k=0}^{L} \frac{\partial \ell(y_k, \widehat{y}_k)}{\partial \widehat{y}_k} \cdot \frac{\partial \widehat{y}_k}{\partial x_k} \cdot \frac{\partial x_k}{\partial x_{k-1}} \cdots \frac{\partial x_1}{\partial x_0}.$$

Since

$$\frac{\partial x_\kappa}{\partial x_{\kappa-1}} = \sigma'_{\kappa-1} a = a\left(x_\kappa - x_\kappa^2\right), \quad \text{for all } \kappa \in [k],$$

we obtain:

$$\frac{\partial \mathcal{L}}{\partial x_0} = \frac{2c}{L+1}\sum_{k=0}^{L} (\widehat{y}_k - y_k) a^k \prod_{\kappa=0}^{k-1} \sigma'_\kappa \tag{2.17}$$

$$= \frac{2c}{L+1}\sum_{k=0}^{L} (\widehat{y}_k - y_k) a^k \prod_{\kappa=1}^{k} \left(x_\kappa - x_\kappa^2\right). \tag{2.18}$$

308

This scalar example demonstrates how to systematically compute gradients in a simple RNN. (We invite the reader to verify that these partial derivatives are consistent with the tensorial expressions of the corresponding Jacobians.)
This example illustrates how to apply the chain rule step-by-step in an RNN, starting from a simple setting.

Parameter Updates

After calculating the gradients, we update the parameters in $\boldsymbol{\theta}$ using a gradient-based optimization algorithm, such as stochastic gradient descent (SGD) or adaptive moment estimation (Adam). For example, with SGD, we update $\boldsymbol{\theta}$ as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}},$$

where η is the learning rate.

By iterating through the forward pass, backward pass, and gradient update steps, the RNN progressively learns to capture dependencies in sequential data. Through the incorporation of nonlinear transformations at each time step, RNNs are able to model complex, time-dependent relationships, which enhances their ability to handle a wide range of sequential patterns.

Python Lab: Forecasting Electric Power Load Using RNNs

Use Clockwork RNNs to have a fair comparison with SARIMA...

2.2 The Gradient Problem in RNNs

Recurrent Neural Networks (RNNs) have demonstrated remarkable capabilities in modeling sequential data due to their inherent ability to retain and process information across time steps. However, a fundamental challenge arises when training RNNs over long sequences: the *vanishing and exploding gradient problem.* This phenomenon hampers the network's capacity to learn long-term dependencies, as gradients propagated backward through time can either diminish to insignificance or amplify uncontrollably. In this section, we explore the mathematical intricacies of this problem, building upon the gradient computation framework previously established, and elucidate the conditions under which gradients vanish or explode during training.

2.2.1 Analysis of the Scalar RNN

To build intuition, we begin by analyzing the gradient problem in the simplified setting of an RNN with a scalar input, scalar output, and a single hidden state. This analysis leverages the explicit expressions for the gradient of the total loss $\mathcal{L}$ derived in Example 3. Specifically, the gradient with respect to the initial condition x_0—given by Equation (2.17) and reproduced below for clarity—provides valuable insights:

$$\frac{\partial \mathcal{L}}{\partial x_0} = \frac{2c}{L+1} \sum_{k=0}^{L} r_k \, a^k \prod_{\kappa=0}^{k-1} \sigma'_\kappa, \tag{2.19}$$

where $r_k = \widehat{y}_k - y_k$ denotes the residual at time k. This equation quantifies how a small perturbation in the initial hidden state x_0 affects the total loss $\mathcal{L}$. Each term in the summation encapsulates the impact of the residual at time k as it propagates backward to influence the gradient. The equation comprises the following key components:

- **Scaling factor:** $\frac{2c}{L+1}$, where c quantifies the influence of the hidden state x_k on the output $\widehat{y}_k$, and $L+1$ normalizes the gradient by the sequence length, ensuring its magnitude remains consistent across varying sequence sizes.
- **Exponential weighting:** The term a^k captures how the residual r_k is modulated by the recurrent weight a. This exponential dependency reflects the compounding effect of a over k time steps.
- **Cumulative activations:** The product of activation function derivatives, $\prod_{\kappa=0}^{k-1} \sigma'_\kappa$, represents the cumulative influence of the activation function's slope over k time steps. This factor dictates how gradients are scaled as they propagate through the network, directly affecting their magnitude and stability.

By deconstructing these components, we gain essential insights into the dynamics of gradient propagation in RNNs. This analysis highlights the potential challenges of the vanishing or exploding gradient problem, which arise when the components above lead to excessively small or large gradient magnitudes.

To gain deeper insight into the gradient problem, let us consider the case where the activation function σ is linear, i.e., $\sigma(z) = z$, which implies that its derivative $\sigma'(z) = 1$ for all z. Under this assumption of linearity, the gradient expression in Equation (2.19) simplifies to:

$$\frac{\partial \mathcal{L}}{\partial x_0} = \frac{2c}{L+1} \sum_{k=0}^{L} r_k \, a^k. \tag{2.20}$$

This simplification illuminates the pivotal role of the recurrent weight a in determining the behavior of the gradient. Specifically, the gradient of the loss exhibits two distinct behaviors depending on the magnitude of a:

- **Vanishing Gradients ($|a| < 1$):** In this case, a^k *decays exponentially* as k increases. Consequently, the contributions to the gradient from later time steps k diminish rapidly, leading to the phenomenon of vanishing gradients.

  Each residual r_k in (2.20) is scaled by the factor a^k. For even moderate values of k, a^k approaches zero, effectively nullifying the influence of r_k on the gradient. As a result, the network is unable to adjust its parameters adequately via gradient descent to account for errors originating from later time steps. This inability to incorporate long-term contributions severely hampers the network's capacity to retain meaningful information across long sequences, thus degrading its performance in tasks requiring the modeling of long-term dependencies.

- **Exploding Gradients ($|a| > 1$):** In this case, a^k *grows exponentially* as k increases. Consequently, the contributions to the gradient from later time steps k become excessively large, resulting in exploding gradients.

  Each residual r_k is amplified by the factor a^k, and for even moderate values of k, a^k grows exponentially. This amplification causes the gradient contributions from later time steps to dominate, leading to numerical instability during training. Large gradients can overwhelm the optimization algorithm, resulting in erratic parameter updates. Consequently, the network struggles to converge, and training becomes unreliable, particularly for tasks involving long sequences.

This analysis underscores the critical role of the recurrent weight a in determining the stability of gradient propagation. It illustrates how inappropriate values of a exacerbate the vanishing or exploding gradient problem, posing significant challenges to the effective training of Recurrent Neural Networks.

2.2.2 Gradient Analysis for Multivariate RNNs

We now analyze the gradient problem in the case of a Recurrent Neural Network (RNN) with multiple inputs, multiple outputs, a vector-valued hidden state, and a linear activation function. Starting with the expression for the gradient with respect to the initial hidden state derived in (2.12), we observe that for a linear activation function, $\Sigma_\kappa^{(2)} = \mathbb{I}^{(2)}$. Consequently, the gradient simplifies to:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_0} = \frac{2}{L+1} \sum_{k=0}^{L} \mathbf{r}_k^{\intercal} C A^k. \tag{2.21}$$

The summands in this expression quantify how the residuals $\mathbf{r}_k$ at each time step influence the gradient through the recurrent structure of the RNN. To understand the behavior of these terms, we leverage the spectral properties of the state matrix A, including its eigenvalues and eigenvectors.

Assuming that A is diagonalizable, it can be expressed as:

$$A = \underbrace{[\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n]}_{U} \cdot \underbrace{\operatorname{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)}_{\Lambda} \cdot \underbrace{[\mathbf{w}_1^{\intercal}; \mathbf{w}_2^{\intercal}; \ldots; \mathbf{w}_n^{\intercal}]}_{W=V^{-1}} = U\Lambda W,$$

where U is matrix whose columns are the right eigenvectors of A, $W = V^{-1}$ is the matrix whose rows are the left eigenvectors of A, and $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of A. Using this decomposition, powers of A can be computed as:

$$A^k = U\Lambda^k W = \sum_{i=1}^{n} \lambda_i^k \mathbf{u}_i \mathbf{w}_i^{\intercal}.$$

Hence, we can write the summands in (2.21) as:

$$\mathbf{r}_k^{\top} C A^k = \mathbf{r}_k^{\top} C \sum_{i=1}^{n} \lambda_i^k \mathbf{u}_i \mathbf{w}_i^{\intercal} = \sum_{i=1}^{n} \lambda_i^k \gamma_{k,i} \mathbf{w}_i^{\intercal},$$

where $\gamma_{k,i} = \mathbf{r}_k^{\top} C \mathbf{u}_i$.

Without loss of generality, we assume that λ_1 is the eigenvalue of A with the largest modulus. Furthermore, we assume that $\gamma_{k,i} = \mathbf{r}_k^\intercal C\mathbf{u}_i \neq 0$, which would not hold only in the rare case where $\mathbf{r}_k \perp C\mathbf{u}_i$. Under these assumptions, the behavior of the gradient norm is dominated by the terms associated with λ_1. We observe two distinct scenarios:

1. **Vanishing Gradients:** If $|\lambda_1| < 1$ and $\gamma_{k,i}$ does not grow exponentially with k, then $\lambda_i^k \gamma_{k,i}$ decays exponentially to zero as k increases for all i. This leads to the **vanishing gradient problem**, where the influence of residuals from later time steps becomes negligible, hindering the network's ability to learn long-term dependencies.
2. **Exploding Gradients:** If $|\lambda_1| > 1$, then $\lambda_1^k \gamma_{k,1}$ grows exponentially with k. This results in the **exploding gradient problem**, where the contributions from residuals at later time steps (i.e., k large) dominate, causing numerical instability and difficulties in training.

This analysis reveals a direct connection between the spectral properties of the state matrix A and the stability of gradient propagation in RNNs. Understanding this connection is essential for designing RNN architectures and training algorithms that mitigate the vanishing and exploding gradient problems.

Note that our analysis of the gradient problem has primarily focused on a single component, namely, $\partial\mathcal{L}/\partial\mathbf{x}_0$. However, similar analysis and conclusions apply to gradient components corresponding to the parameters involved in the recursion of the hidden state—specifically, the state matrix A, the input matrix B, and the bias vector $\mathbf{b}$. This is because the recursive dynamics inherently amplify the effect of perturbations in these parameters, propagating their influence throughout the network and into the final gradient.

In contrast, the gradient components associated with parameters in the output equation—such as the output matrix C, the direct transmission matrix D, and the output bias vector $\mathbf{d}$—do not exhibit the vanishing or exploding gradient issue. This distinction arises because the output equation is independent of the recursive structure of the hidden state and does not involve the compounded effects of recurrent amplification or decay.

2.2.3 Effect of Nonlinear Activations

Thus far, we have analyzed the vanishing and exploding gradient problem in the particular case of a linear activation function. When the activation function $\sigma(\cdot)$ is nonlinear, such as the sigmoid function, the dynamics of gradient propagation become significantly more complex. For simplicity, we focus our analysis on a scalar RNN with one hidden state and a sigmoid activation function.

In Example 3, we derived the following expression for the gradient of the loss function in this case (repeated below for convenience):

$$\frac{\partial\mathcal{L}}{\partial x_0} = \frac{2c}{L+1}\sum_{k=0}^{L} r_k\, a^k P_k, \text{ with } P_k = \prod_{\kappa=0}^{k-1} \sigma'_\kappa. \tag{2.22}$$

Given that the maximum derivative of the sigmoid function is $1/4$ (achieved when the argument is 0), we have $\sigma'_\kappa \leq 1/4$ for all κ, and the product P_k decays exponentially as k increases:

$$P_k \leq 4^{-k}.$$

This exponential decay introduces critical challenges to gradient propagation:

- **Vanishing Gradients:** When $|a| < 1$, the exponential decay of the term P_k exacerbates the vanishing gradient problem, as it diminishes the magnitude of the summands in (2.22). This severely limits the network's ability to learn long-term dependencies, as errors from distant time steps fail to meaningfully influence parameter updates.
- **Exploding Gradients:** When $|a| > 1$, the exponentially-decaying term P_k has the potential to tame the exponential growth induced by a^k. However, in practice, this mitigation is highly unlikely. Finding a value of a that precisely balances the vanishing effect of P_k and the exploding effect of a^k is exceptionally challenging in real-world scenarios. This fine-tuned equilibrium is rarely achieved, leaving the optimization process prone to instability and divergence as gradients become overwhelmingly large or negligibly small due to contributions from later time steps.

In conclusion, the sigmoid activation function exacerbates the vanishing gradient problem due to its bounded derivative, making it challenging for gradients to propagate effectively across many time steps. This limitation is one of the key reasons why vanilla RNNs struggle to learn dependencies spanning long sequences. Understanding these effects is crucial for designing RNN architectures and selecting suitable activation functions to mitigate the gradient problem, thereby enabling effective learning of long-term dependencies.

2.2.4 Training Techniques to Mitigate the Gradient Problem

The challenges of vanishing and exploding gradients can significantly affect the training of RNNs using gradient-based methods. When the gradients either vanish or grow excessively during backpropagation, the training process becomes unstable, making it difficult for the model to converge to an optimal solution. To mitigate these challenges, several techniques have been developed, which we discuss below.

- **Gradient Clipping:** A widely used technique to mitigate exploding gradients is *gradient clipping*. In this approach, the norm of the gradient, $\|\partial\mathcal{L}/\partial\boldsymbol{\theta}\|$, is monitored during backpropagation, and if the gradient norm exceeds a predefined threshold, it is scaled down to maintain stability. Specifically, when the norm of the gradient surpasses this threshold, the gradient vector is rescaled so that its norm matches the threshold value:
$$\text{if } \|\nabla\mathcal{L}\| > \tau, \quad \nabla\mathcal{L} \leftarrow \tau\frac{\nabla\mathcal{L}}{\|\nabla\mathcal{L}\|}.$$
This normalization prevents the gradient from growing too large and destabilizing the learning process, especially in deep architectures or when training with large learning rates.
- **Eigenvalue Regularization:** Another approach specifically targets the eigenvalues of the state matrix A, which are directly connected to the gradient problem. To prevent both vanishing and exploding gradients, regularization terms are introduced to control the magnitude of the eigenvalues. By incorporating an eigenvalue penalty into the loss function, we can constrain the moduli of the eigenvalues of A to remain

below 1. A commonly used regularization term is:

$$\mathcal{L}_{\text{reg}} = \alpha \sum_i (|\lambda_i| - 1)^2,$$

where λ_i are the eigenvalues of A. This penalty term discourages eigenvalues from growing too large or becoming too small, thus promoting stable dynamics.

To implement this regularization technique, we need to compute the gradient $\partial\mathcal{L}_{\text{reg}}/\partial A$. For each eigenvalue λ_i, the derivative $\partial\lambda_i/\partial A$ is given by $\mathbf{w}_i\mathbf{v}_i^\intercal$, where $\mathbf{v}_i$ and $\mathbf{w}_i$ are the right and left eigenvectors of A, respectively, normalized such that $\mathbf{v}_i^\intercal\mathbf{w}_i = 1$. Therefore, the gradient of the regularization term becomes:

$$\frac{\partial\mathcal{L}_{\text{reg}}}{\partial A} = \alpha \sum_i 2(|\lambda_i| - 1)\frac{\lambda_i}{|\lambda_i|}\mathbf{w}_i\mathbf{v}_i^\intercal.$$

This gradient can be incorporated into backpropagation to adjust A during training. Such regularization ensures balanced gradient propagation, supporting stable and effective learning in recurrent architectures.

- **Dropout:** Dropout is a stochastic regularization technique that mitigates overfitting while indirectly stabilizing gradient flow. During training, dropout randomly sets a fraction of hidden units to zero at each time step. This introduces noise into the hidden state dynamics, forcing the network to distribute its representational capacity across multiple units. In RNNs, dropout can be applied in various ways, such as between time steps or within hidden layers, but care must be taken to preserve temporal consistency by using the same dropout mask across time steps. Dropout has been shown to improve generalization and enhance robustness, particularly in networks prone to overfitting.

- **Learning Rate Scheduling:** The learning rate is a critical hyperparameter in gradient-based optimization. Large learning rates can exacerbate gradient instability, while small learning rates can hinder convergence. Learning rate scheduling dynamically adjusts the learning rate during training to strike a balance. For instance, learning rates can be reduced when validation performance plateaus or scaled adaptively based on gradient norms. Common scheduling strategies include exponential decay, cosine annealing, and performance-based reductions.

- **Layer Normalization:** Layer normalization stabilizes the hidden state dynamics by normalizing activations within each layer. This is achieved by centering the activations around their mean and scaling them to have unit variance, followed by a learnable affine transformation. By stabilizing activations across time steps, layer normalization reduces the likelihood of exploding or vanishing gradients and enhances the overall training stability.

In summary, addressing gradient instability in RNNs requires a combination of techniques tailored to the specific challenges of the task and dataset. Gradient clipping ensures that gradients remain bounded, eigenvalue regularization controls the dynamics of the state matrix, dropout enhances generalization, and learning rate scheduling and normalization stabilize training. Together, these methods enable RNNs to model complex temporal dependencies effectively, even in the presence of long sequences or deep architectures.

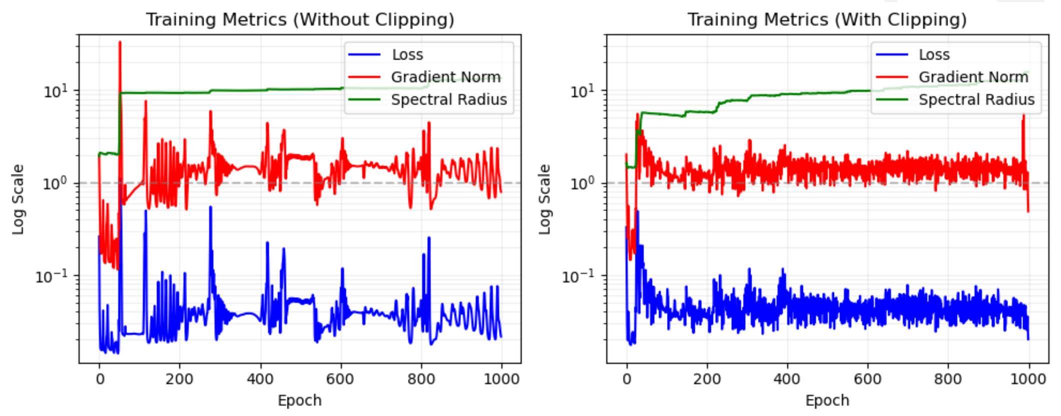


Figure 2.2: Evolution of training metrics for RNN with and without gradient clipping. The plots show the MSE loss (blue), gradient norm (red), and spectral radius (green) in logarithmic scale. Left: Training without gradient clipping exhibits high variability in gradient norms and potentially unstable spectral radius growth. Right: Training with gradient clipping (threshold = 1.0) demonstrates more controlled gradient dynamics and stable spectral radius evolution. The horizontal dashed line at $y = 1$ serves as a reference for gradient norm magnitude and spectral radius stability.

Empirical Analysis of Gradient Clipping in RNN Training

We present an empirical study demonstrating the effects of gradient clipping on RNN training dynamics. Our experiment utilizes a simple RNN architecture trained to forecast a synthetic time series comprising a sinusoidal signal with additive Gaussian noise. The network architecture consists of a single RNN layer with 32 hidden units, followed by a linear projection layer. To illustrate the impact of gradient clipping, we track three key metrics during training: the Mean Squared Error (MSE) loss, the spectral norm of the loss function gradient, and the spectral radius of the system matrix A. We compare two training scenarios: one with gradient clipping threshold set to 1.0, and another without clipping (Code available in GitHub: Gradient_Clipping_RNN.ipynb).

The results in Figure 2.2 reveal several notable phenomena. In the unclipped scenario, we observe significant fluctuations in gradient norms, occasionally exceeding multiple orders of magnitude above the reference threshold. The spectral radius of the state matrix also exhibits potential instability. Conversely, the clipped training scenario demonstrates more controlled gradient dynamics, with the gradient norm effectively bounded near the clipping threshold. This stabilization appears to promote more consistent updates to the recurrent weights, as evidenced by the smoother evolution of the spectral radius.

These observations align with theoretical expectations: gradient clipping helps mitigate the exploding gradient problem common in RNN training, leading to more stable optimization dynamics. The empirical evidence suggests that clipping not only controls gradient magnitude but also induces more regulated changes in the recurrent weight matrix, potentially aiding in the network's ability to capture long-term dependencies.

2.2.5 RNN Variants to Mitigate the Gradient Problem

Recurrent Neural Networks (RNNs) have long been celebrated for their ability to model sequential data. However, they are not without challenges; chief among them is the vanishing gradient problem, which hinders learning in long sequences. Over the years, various architectural innovations have been proposed to address these limitations. In this section, we delve into some of the notable RNN variants that enhance gradient flow and improve training stability.

Recurrent Highway Networks

Recurrent Highway Networks extend the standard RNN architecture by incorporating gating mechanisms that regulate the flow of information over time. Specifically, the hidden state space is augmented with a **transform gate** vector $\mathbf{t}_k$ and a complementary **carry gate** vector $\mathbf{c}_k$. These gates are defined as:

$$\boxed{\mathbf{t}_k = \sigma\left(\widetilde{A}\mathbf{x}_{k-1} + \widetilde{B}\mathbf{u}_{k-1} + \widetilde{\mathbf{b}}\right), \quad \mathbf{c}_k = \mathbf{1} - \mathbf{t}_k,}$$

where $\widetilde{A}$ and $\widetilde{B}$ are weight matrices, $\widetilde{\mathbf{b}}$ is a bias vector, and $\sigma(\cdot)$ denotes the sigmoid activation function. Consequently, the entries of the gate vectors lie in the range $(0, 1)$. The transform gate $\mathbf{t}_k$ determines how much new information is integrated into the hidden state, while the complementary carry gate $\mathbf{c}_k$ governs how much of the past information is retained.

The hidden state update involves a linear combination of the previous hidden state and a newly computed **candidate hidden state**, which is given by:

$$\boxed{\widehat{\mathbf{x}}_k = \phi\left(\widehat{A}\mathbf{x}_{k-1} + \widehat{B}\mathbf{u}_{k-1} + \widehat{\mathbf{b}}\right).}$$

Here, $\widehat{A}$ and $\widehat{B}$ are weight matrices, $\widehat{\mathbf{b}}$ is a bias vector, and $\phi(\cdot)$ is a non-linear activation function such as tanh or ReLU. The equation defining the candidate hidden state combines the input $\mathbf{u}_{k-1}$ and the previous hidden state $\mathbf{x}_{k-1}$ to generate a new vector $\widehat{\mathbf{x}}_k$ that encapsulates the potential contribution of 'fresh information' to be potentially combined with the hidden state.

The final hidden state $\mathbf{x}_k$ is then updated as:

$$\boxed{\mathbf{x}_k = \mathbf{t}_k \odot \widehat{\mathbf{x}}_k + \mathbf{c}_k \odot \mathbf{x}_{k-1},}$$

where $\odot$ denotes Hadamard (element-wise) multiplication. This formulation defines the updated hidden state $\mathbf{x}_k$ as a **convex combination**[2] of the candidate hidden state $\widehat{\mathbf{x}}_k$ and the previous hidden state $\mathbf{x}_{k-1}$, as the transform and carry gates satisfy $\mathbf{t}_k + \mathbf{c}_k = 1$ at every time step. The balance between new and historical information is dynamically adjusted by the gate vectors. The output is then computed using the standard output equation: $\widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{d}$.

[2] A convex combination is a weighted sum of elements where the weights are non-negative and sum to one.

(TBD: Theoretical analysis of scalar/linear case) The transform gate $\mathbf{t}_k$ acts as a filter, determining the extent to which the candidate hidden state $\widehat{\mathbf{x}}_k$ influences the updated hidden state. Conversely, the carry gate $\mathbf{c}_k$ dictates how much of the previous hidden state $\mathbf{x}_{k-1}$ is retained without modification. This gating mechanism provides several advantages:

1. *Improved Gradient Flow:* The inclusion of the term $\mathbf{c}_k \odot \mathbf{x}_{k-1}$ enables the network to retain information from the previous hidden state over time. This retention mechanism ensures that gradients can flow more effectively through the network during backpropagation, thereby alleviating the vanishing gradient problem and facilitating the learning of long-term dependencies.

2. *Dynamic Information Filtering:* By adjusting the values of $\mathbf{t}_k$ and $\mathbf{c}_k$, the network can selectively focus on relevant features of the input $\mathbf{u}_k$ while maintaining important context from the past hidden state $\mathbf{x}_{k-1}$.

3. *Modeling Temporal Dependencies:* The convex combination facilitates the modeling of both short-term and long-term dependencies, as the network adaptively blends new and old information to meet the demands of the sequence data.

Recurrent Highway Networks are a principled extension of traditional RNNs that leverage gating mechanisms to dynamically balance the integration of new information with the retention of historical context. This architecture enhances the network's ability to capture complex temporal patterns while ensuring stable and efficient training.

Residual RNNs

Residual connections, initially popularized in convolutional neural networks, have proven effective in addressing gradient-related issues in RNNs. In Residual RNNs, the hidden state update incorporates a **direct skip connection** from the previous hidden state, leading to the following update rule:

$$\boxed{\mathbf{x}_k = \mathbf{x}_{k-1} + \sigma\left(A\mathbf{x}_{k-1} + B\mathbf{u}_{k-1} + \mathbf{b}\right).}$$

Here, A and B are weight matrices, $\mathbf{b}$ is a bias vector, and $\sigma(\cdot)$ is a non-linear activation function such as the sigmoid or tanh. The output is then computed using the standard output equation: $\widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{d}$.

The central idea of residual connections is to allow the network to learn modifications to the hidden state rather than the entire transformation at each time step. By directly adding the previous hidden state $\mathbf{x}_{k-1}$ to the newly computed transformation, residual RNNs introduce a shortcut pathway that simplifies the learning process. This formulation helps to mitigate the vanishing gradient problem by ensuring that the gradients can flow through the skip connection during backpropagation. As a result, the network is better equipped to learn long-term dependencies, even in sequences with many time steps.

RNNs with Causal Attention Mechanisms

Attention mechanisms have fundamentally transformed the landscape of sequential modeling by enabling RNNs to dynamically focus on the most relevant time steps within an input sequence. Unlike traditional RNNs, which rely solely on the most recent hidden state to represent all prior information, attention mechanisms introduce a **context vector** that aggregates information from *all* past hidden states, providing a more flexible and comprehensive representation.

This global aggregation mechanism is implemented using **attention weights**, denoted by $\alpha_{k,t}$, which quantify the relative relevance of each past hidden state $\mathbf{x}_t$ in the context of the current state $\mathbf{x}_k$. These weights are computed using a **softmax function**[3] applied to similarity scores between the current hidden state $\mathbf{x}_k$ and each past[4] hidden state $\mathbf{x}_t$: use $k-h$ instead of t in next version

$$\boxed{\alpha_{k,t} = \frac{\exp\left(\mathbf{x}_k^\intercal \mathbf{x}_t\right)}{\sum_{\tau=0}^{k-1} \exp\left(\mathbf{x}_k^\intercal \mathbf{x}_\tau\right)},}$$

where the **similarity score** between two state vectors is computed using the inner product $\mathbf{x}_k^\intercal \mathbf{x}_t$.

Using these attention weights, we define a **context vector** $\mathbf{c}_k$, which effectively integrate past information by prioritizing contributions from time steps deemed most relevant by the attention mechanism. This vector is computed as a weighted sum of all *past* hidden states:

$$\boxed{\mathbf{c}_k = \textstyle\sum_{t=0}^{k-1} \alpha_{k,t} \mathbf{x}_t.}$$

This context vector The final output $\mathbf{y}_k$ is then derived by combining the current hidden state $\mathbf{x}_k$ with the context vector $\mathbf{c}_k$:

$$\boxed{\mathbf{y}_k = C\mathbf{x}_k + D\mathbf{c}_k + \mathbf{d},}$$

where C and D are weight matrices and $\mathbf{d}$ is a bias vector. This formulation allows the model to synthesize information from both the current hidden state and the aggregated context vector, enabling more informed and accurate predictions.

By explicitly attending to relevant time steps, attention-enhanced RNNs significantly improve their ability to capture long-range dependencies, overcoming the limitations of traditional RNNs in handling lengthy sequences. This innovation has not only enhanced the performance of sequential models but has also laid the groundwork for the development of more advanced architectures, such as the Transformer.

This chapter has provided a comprehensive exploration of recurrent neural networks (RNNs), focusing on their ability to model sequential data and their inherent challenges. A particular emphasis was placed on the gradient problem, which manifests as vanishing and exploding gradients during training, posing significant obstacles to learning long-term

[3]The softmax function maps a vector of real numbers to a probability distribution, where each entry is non-negative and the entries sum to one. For a vector $\mathbf{z} = [z_1, z_2, \ldots, z_n]$, the softmax function is defined as $\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}$ for $i = 1, 2, \ldots, n$.

[4]In *non-causal* applications, attention weights $\alpha_{k,t}$ are often computed for all time steps t, including future ones. For causal data, the computation of $\alpha_{k,t}$ must enforce $t < k$, effectively masking any contributions from future time steps during training and inference.

dependencies. We discussed various training techniques designed to mitigate these issues, including gradient clipping, eigenvalue regularization, learning rate scheduling, dropout, and layer normalization. Furthermore, we explored architectural innovations such as residual RNNs and highway networks, which enhance gradient flow and enable the effective training of deeper recurrent models. These approaches collectively address the limitations of standard RNNs and improve their robustness in sequential modeling tasks. Given their critical importance in time-series forecasting, Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) will be discussed in depth in the next chapter.

Exercises

1. Consider a single mass-spring-damper system consisting of a point mass m, connected to a fixed wall by a linear spring with stiffness k and a damper with damping coefficient c. Let $x(t)$ denote the displacement of the mass from its equilibrium position, and let $u(t)$ represent an external force applied to the mass. The equation of motion for the system is given by a second-order differential equation:

$$m\ddot{x}(t) + c\dot{x}(t) + kx(t) = u(t).$$

Using the Euler discretization method with a time step Δt, we can approximate the first and second derivatives as:

$$\dot{x}(t) \approx \frac{x_k - x_{k-1}}{\Delta t}, \quad \ddot{x}(t) \approx \frac{\dot{x}_k - \dot{x}_{k-1}}{\Delta t},$$

where $x_k = x(k\cdot\Delta t)$ and $\dot{x}_k = \dot{x}(k\cdot\Delta t)$. In this exercise, you are tasked with converting the result of the Euler discretization into an LSSM.

   (a) Define the discrete-time state vector $\mathbf{x}_k = [x_k^{(1)}, x_k^{(2)}]^\top$, where $x_k^{(1)} = x_k$ and $x_k^{(2)} = \dot{x}_k$

   (b) Transform the discretized equation of motion into a state-space model in the form:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}u_k,$$

   where $\mathbf{A}$ is the state transition matrix, $\mathbf{B}$ is the input matrix, and u_k is the external input.

   (c) Express $\mathbf{A}$ and $\mathbf{B}$ explicitly in terms of m, c, k, and Δt.

   (d) For what values of Δt as a function of m, c, k is the discretized system stable?

2. Consider the operation of differencing, where the d-th order differencing operator is defined as $(1-\mathcal{D})^d$, with $\mathcal{D}$ denoting the backward shift operator. In this exercise, we analyze the effects of differencing on trends and noise in a time series.

   (a) Prove that d-th order differencing removes a polynomial trend of degree d, i.e., a trend of the form:

$$\text{Trend} = a_d k^d + a_{d-1}k^{d-1} + \ldots + a_1 k + a_0,$$

   where k represents time. *Hint*: Differencing is conceptually similar to differentiation.

   (b) Consider a pure noise signal $Y_k = \epsilon_k$, where ϵ_k is white noise with variance σ_ϵ^2. Let $W_k^{(d)} = (1-\mathcal{D})^d Y_k$. Compute the variances of $W_k^{(1)}$ and $W_k^{(2)}$, and analyze how differencing amplifies noise.

   (c) Discuss the advantages and disadvantages of differencing based on your findings in the previous parts.

3. Consider the LSSM:

$$x_{k+1} = ax_k + bu_k, \quad y_k = cx_k \quad (d = 0),$$

where all variables and parameters are scalars. Use the convolution operation to compute the output y_k when the initial state is zero and the input signal u_k is one of the following:

(a) Compute the output for a **delta input**:

$$u_k = \delta_k, \quad \text{where } \delta_k = \begin{cases} 1 & \text{if } k = 0, \\ 0 & \text{otherwise.} \end{cases}$$

(b) Compute the output for a **step input**, defined as:

$$u_k = \begin{cases} 1 & \text{if } k \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

(c) Compute the output for a **ramp input**, defined as:

$$u_k = k, \quad k \geq 0.$$

(d) Compute the output for the input signal shown in the figure below:

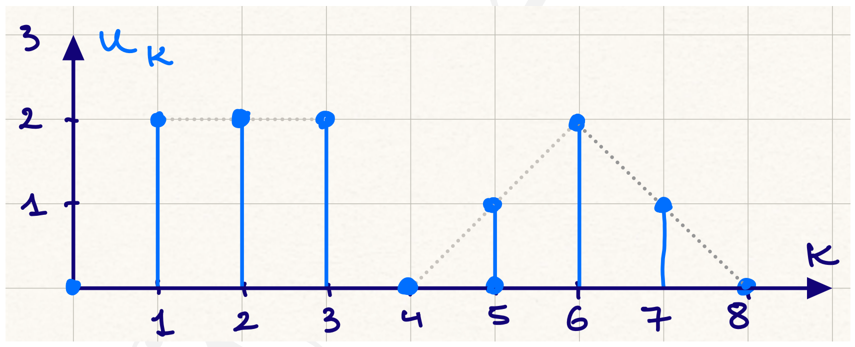


Figure 2.3: Illustration of the combined input signal for the exercise.

Hint: In a linear state-space model (LSSM), where the response to an input signal u_k is y_k, the following properties hold:

- **Time-shifting inputs** shifts the outputs, i.e., when the input is u_{k-s}, the output is y_{k-s} for any integer s.
- **Scaling inputs** scales the outputs, i.e., when the input is $\alpha \cdot u_k$, the output is $\alpha \cdot y_k$ for any real coefficient α.
- **Summing inputs** sums the outputs, i.e., when the input is $u_k^{(1)} + u_k^{(2)}$, the output is $y_k^{(1)} + y_k^{(2)}$.

Use these properties to decompose the input signal as a linear combination of shifted signals for which you know the corresponding outputs.

4. Answer the questions below:

   (a) Consider the following linear state-space model (LSSM):

   $$x_{k+1} = ax_k + bu_k + \varepsilon_k, \quad y_k = cx_k + \eta_k,$$

   where all variables and coefficients are scalars. The initial condition is $x_0 = \sqrt{2}$, and the noise terms are independently and identically distributed:

   $$\varepsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0,4), \quad \eta_k \overset{\text{iid}}{\sim} \mathcal{N}(0,1).$$

   Compute the 95%-confidence interval for the output y_k for all $k \geq 0$.

   (b) Consider the second order LSSM:

   $$\begin{bmatrix} x_{k+1}^{(1)} \\ x_{k+1}^{(2)} \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{3} \end{bmatrix} \begin{bmatrix} x_k^{(1)} \\ x_k^{(2)} \end{bmatrix} + \begin{bmatrix} \boldsymbol{\varepsilon}_k \\ \sqrt{2}\varepsilon_{k-1} \end{bmatrix}, \quad \mathbf{y}_k = \mathbf{x}_k + \begin{bmatrix} 0 \\ \eta_k \end{bmatrix},$$

   where $\varepsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0,1)$ and $\eta_k \overset{\text{iid}}{\sim} \mathcal{N}(0,2)$. Compute the 95%-confidence ellipsoids for the hidden state and the output for all $k \geq 0$ (use $\chi^2_{2,0.95} \approx 6$).
   Hint: When the matrices A and Q are diagonal, the state covariance P_k is also diagonal. Also, the solution of the series $a_{k+1} = \frac{a_k}{d} + u$ with $a_0 = 0$ is:

   $$a_k = \frac{ud}{d-1}\left(1 - \frac{1}{d^k}\right).$$

5. Consider a recursive forecasting model defined by the following equations:

   $$x_{k+1} = x_k^a,$$
   $$\widehat{y}_k = c \cdot x_k,$$

   where:

   - The state x_k and the output $\widehat{y}_k$ are scalar-valued.
   - The parameters c and the initial condition x_0 are positive scalars.
   - The parameters a and b are real numbers.

   The goal is to train the model parameters $\boldsymbol{\theta} = (a, c, x_0)$ using Backpropagation Through Time (BPTT) with a given output sequence $(y_k)_{k=0}^{L}$. Assume the loss function $\mathcal{L}$ is the mean absolute error (MAE). *Hint:*

   $$\frac{d}{dx}|x-a| = \text{sgn}(x-a) = \begin{cases} 1 & \text{if } x > a, \\ -1 & \text{if } x < a. \end{cases}$$

   Answer the following questions:

   (a) Using the appropriate chain rule, compute $\frac{\partial \mathcal{L}}{\partial c}$.

   (b) Using the appropriate chain rule, compute $\frac{\partial \mathcal{L}}{\partial x_0}$.

(c) Derive a condition under which the previous gradients explode. Your condition should be of the form `function`$(a, x_0) > 1$, where you need to derive `function`$(\cdot, \cdot)$.

(d) List techniques that can be used to mitigate the problem of exploding gradients.

6. Consider the standard Recurrent Neural Network (RNN) architecture covered in this chapter. Using tensor notation, prove the following identity for the gradient of the hidden state with respect to the weight matrix A:

$$\frac{\partial \mathbf{x}_k}{\partial A} = \Sigma_{k-1}^{(3)} \odot \left(\mathbb{I}^{(2)} \otimes \mathbf{x}_{k-1} + A{:}\frac{\partial \mathbf{x}_{k-1}}{\partial A} \right).$$

Appendix A

Tensors: A Computational Perspective

In this section, we examine tensors from a computational perspective, setting aside more abstract algebraic concepts (for a thorough theoretical treatment, see [**?**]; for an extended computational perspective, see [**?**] and [**?**]).

A tensor of order d, $T \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, is an array of values that requires exactly d indices to specify each entry. The **order** (or **rank**) of a tensor indicates the number of **dimensions** (also called **modes** or **ways**) it has. For example, a scalar is a 0th-order tensor, a vector is a 1st-order tensor, a matrix is a 2nd-order tensor, and so on. The tuple of integers $(n_1, \ldots, n_d)$ defines the **shape** of the tensor, specifying the **index range** or size of each dimension. The uppercase letter T represents the tensor as a whole, while individual entries within the tensor are denoted by lowercase letters, such as $t_{i_1 i_2 \cdots i_d}$, with $[T]_{i_1 i_2 \cdots i_d} = t_{i_1 i_2 \cdots i_d}$. As examples, a **vector** is a 1-dimensional tensor, requiring one index, while a **matrix** is a 2-dimensional tensor, requiring two indices. Tensors generalize these concepts to higher dimensions. The **order** of a tensor refers to the number of indices needed to access its elements, which corresponds to the number d in the notation $\mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$. Thus, a d-dimensional tensor is of order d. Tensors have wide applications in fields such as physics, engineering, and machine learning, where they allow for efficient representation of data in high-dimensional spaces.

In what follows, we use **multiindices** (i.e., ordered sequences of indices), such as $\mathcal{I}_p = (i_1, i_2, \ldots, i_p)$ or $\mathcal{J}_q = (j_1, j_2, \ldots, j_q)$, to index tensor elements, where we use the subscript of the multiindex to indicate its length. The concatenation of two multiindices results in another multiindex, denoted as:

$$\mathcal{I}_p, \mathcal{J}_q = (i_1, i_2, \ldots, i_p, j_1, j_2, \ldots, j_q),$$

where a *comma* represents the concatenation of two sequences. Using this notation, tensor elements are indexed as:

$$[T]_{\mathcal{I}_d} = [T]_{i_1 i_2 \cdots i_d} = t_{i_1 i_2 \cdots i_d} = t_{\mathcal{I}_d}.$$

Using multiindex notation, we define the **delta function** as:

$$\delta_{\mathcal{I}_d \mathcal{J}_d} = \begin{cases} 1, & \text{if } \mathcal{I}_d = \mathcal{J}_d, \\ 0, & \text{otherwise.} \end{cases}$$

For example, the delta function of order 2, also called the **Kronecker delta**, is defined as $\delta_{ij} = 1$ when $i = j$, and 0 otherwise. This delta function defines the identity tensor of order 2 as follows: $\left[\mathbb{I}^{(2)}\right]_{ij} = \delta_{ij}$.

A.1 List of Tensor Operations

- **Tensor Permutation**: Consider a tensor T of order d with indices $(i_1, i_2, \ldots, i_d)$. A permutation operation reorders these indices according to a specified order, say $(i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(d)})$, where σ represents a permutation of $\{1, 2, \ldots, d\}$. More formally, if σ denotes a permutation, the permuted tensor can be represented as:

$$P^{\sigma}(T_{i_1 i_2 \ldots i_d}) = T_{i_{\sigma(1)} i_{\sigma(2)} \ldots i_{\sigma(d)}}.$$

  This operation is critical in numerous applications of tensor analysis, including machine learning and scientific computing, as it enables flexible reshaping, alignment, and manipulation of tensor dimensions to meet specific computational requirements.

- **Addition and Subtraction**: The sum (or subtraction) of two tensors A and B of the same shape is another tensor of the same shape, defined entry-wise:

$$[A + B]_{\mathcal{I}_d} = [A + B]_{i_1 \cdots i_d} = a_{i_1 \cdots i_d} + b_{i_1 \cdots i_d} = a_{\mathcal{I}_d} + b_{\mathcal{I}_d}.$$

- **Scalar Product**: The product of a scalar α and a tensor $T \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is another tensor, defined element-wise as:

$$[\alpha \cdot T]_{\mathcal{I}_d} = \alpha \cdot t_{\mathcal{I}_d}.$$

- **Hadamard Product**: The Hadamard product of two tensors A and B of the same shape is another tensor of the same shape, defined entry-wise:

$$[A \odot B]_{\mathcal{I}_d} = [A \odot B]_{i_1 \cdots i_d} = a_{i_1 \cdots i_d} \cdot b_{i_1 \cdots i_d} = a_{\mathcal{I}_d} \cdot b_{\mathcal{I}_d}.$$

- **Inner Product**: The inner product of two tensors A and B of the same shape is a scalar defined as:

$$\langle A, B \rangle = \sum_{i_1, i_2, \ldots, i_d} a_{i_1 i_2 \ldots i_d} b_{i_1 i_2 \ldots i_d} = \sum_{\mathcal{I}_d} a_{\mathcal{I}_d} b_{\mathcal{I}_d},$$

  where the summation is over the set of all possible multiindexes.

- **Contraction Product**: Consider two tensors:

$$A \in \mathbb{R}^{n_1 \times \cdots \times n_p \times c_1 \times \cdots \times c_r} \text{ (order } p + r),$$
$$B \in \mathbb{R}^{c_1 \times \cdots \times c_r \times m_1 \times \cdots \times m_q} \text{ (order } r + q).$$

The contraction product of these two tensors is another tensor of order $p+q$, defined entry-wise as:

$$\begin{aligned}[A \overset{r}{:} B]_{\mathcal{I}_p,\mathcal{J}_q} &= [A \overset{r}{:} B]_{i_1\dots i_p, j_1\dots j_q} \\ &= \sum_{s_1\dots s_r} a_{i_1\dots i_p, s_1\dots s_r} b_{s_1\dots s_r, j_1\dots j_q} \\ &= \sum_{\mathcal{S}_r} a_{\mathcal{I}_p,\mathcal{S}_r} b_{\mathcal{S}_r,\mathcal{J}_q}. \end{aligned} \tag{A.1}$$

When $r=1$ (i.e., the summation is over a single index s_1), the symbol "$\overset{r}{:}$" is usually simplified to ":". For example, the contraction product of two tensors M and N of order 2 (i.e., two matrices) can be written as:

$$[M{:}N]_{i,j} = \sum_s m_{is} \cdot n_{sj} = [M \cdot N]_{i,j},$$

which corresponds to the standard matrix product. Similarly, we have that:

$$M \overset{2}{:} N = \sum_{s_1,s_2} m_{s_1 s_2} \cdot n_{s_1 s_2} = \langle M, N \rangle,$$

which corresponds to the inner product of two matrices.

For a tensor $T \in \mathbb{R}^{n_1\times\cdots\times n_p}$ of order p, the **identity tensor** for the contraction product is a tensor of order $2p$ defined entry-wise as:

$$\mathbb{I}^{(2p)}_{\mathcal{I}_p,\mathcal{J}_p} = \delta_{\mathcal{I}_p\mathcal{J}_p}.$$

With respect to the contraction product, the identity tensor satisfies:

$$[\mathbb{I}^{(2p)} \overset{p}{:} T]_{\mathcal{I}_p} = \sum_{\mathcal{J}_p} [\mathbb{I}^{(2p)}]_{\mathcal{I}_p,\mathcal{J}_p} [T]_{\mathcal{J}_p} = \sum_{\mathcal{J}_p} \delta_{\mathcal{I}_p\mathcal{J}_p} [T]_{\mathcal{J}_p} = [T]_{\mathcal{I}_p}.$$

- **Kronecker Product**: The Kronecker product of two tensors A and B of orders p and q, respectively, results in a tensor of order $p+q$ and defined entry-wise as:

$$[A \otimes B]_{\mathcal{I}_p,\mathcal{J}_q} = [A \otimes B]_{i_1\cdots i_p, j_1\cdots j_q} = a_{i_1\cdots i_p} b_{j_1\cdots j_q} = a_{\mathcal{I}_p} b_{\mathcal{J}_q}.$$

For example, the Kronecker product of two tensors[1] M and N of order 2 is a tensor of order 4 defined entry-wise as $[M\otimes N]_{i_1 i_2, j_1 j_2} = m_{i_1 i_2} n_{j_1 j_2}$.

The Kronecker product can be used to express a tensor $T \in \mathbb{R}^{n_1\times\cdots\times n_d}$ in terms of its individual components $t_{i_1\cdots i_d}$, as follows:

$$T = \sum_{i_1\cdots i_d} t_{i_1\cdots i_d} \cdot \mathbf{e}^{n_1}_{i_1} \otimes \cdots \otimes \mathbf{e}^{n_d}_{i_d} = \sum_{\mathcal{I}_d} t_{\mathcal{I}_d} \mathbf{E}_{\mathcal{I}_d},$$

where $\mathbf{e}^n_i$ is the unit vector representing the i-th element in the canonical basis of $\mathbb{R}^n$ (i.e., the one-hot encoded vector for the i-th element in an n-dimensional space) and $\mathbf{E}_{\mathcal{I}_d} = \mathbf{e}^{n_1}_{i_1} \otimes \cdots \otimes \mathbf{e}^{n_d}_{i_d}$ is a tensor of order d with all entries zero except for a 1 at the position indexed by $\mathcal{I}_d$ (i.e., a one-hot encoder tensor).

[1]In the context of matrix algebra, the Kronecker product of two matrices is typically defined as another matrix. However, when extending this operation to tensors of order 2, the result may be regarded as a higher-order tensor.

- **Tucker Product**: The Tucker product is a multi-linear operation that generalizes matrix multiplication to higher-order tensors. Given a tensor $T \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ and matrices $U^{(k)} \in \mathbb{R}^{m_k \times n_k}$, the Tucker product yields a new tensor of size $m_1 \times m_2 \times \cdots \times m_d$. Formally, this operation is defined component-wise as:

$$\left[T \times_1 U^{(1)} \times_2 U^{(2)} \cdots \times_d U^{(d)}\right]_{i_1, i_2, \ldots, i_d} = \sum_{s_2, \ldots, s_d} T_{s_1, s_2, \ldots, s_d} \, u^{(1)}_{i_1, s_1} \cdots u^{(d)}_{i_d, s_d}.$$

  In this expression, $u^{(k)}_{i_k, s_k}$ denotes the elements of the matrix $U^{(k)}$, applied to each mode k of the tensor T. The Tucker product is fundamental to the Tucker decomposition and enables a compact representation of multi-dimensional data.

A.2 Jacobian Tensor

The Jacobian matrix of a vector-valued function $\mathbf{f} \colon \mathbb{R}^n \to \mathbb{R}^m$ with respect to a vector $\mathbf{x} \in \mathbb{R}^n$ is defined as:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix},$$

where $\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) & f_2(\mathbf{x}) & \cdots & f_m(\mathbf{x}) \end{bmatrix}^\mathsf{T}$. This matrix contains all first-order partial derivatives of $\mathbf{f}$ with respect to $\mathbf{x}$. One can extend the concept of the Jacobian matrix to higher-order tensors, as follows. Consider a function F that maps an input tensor of order q to an output tensor of order p, i.e.,

$$F \colon \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_q} \to \mathbb{R}^{m_1 \times m_2 \times \cdots \times m_p}.$$

The partial derivatives of the elements of the tensor-valued function $F(X)$ with respect to all the entries of its tensor argument X can be arranged into a new tensor of order $p + q$. This tensor, denoted the **Jacobian tensor**, is defined entry-wise as:

$$\left[\frac{\partial F}{\partial X}\right]_{\mathcal{I}_p, \mathcal{J}_q} = \left[\frac{\partial F}{\partial X}\right]_{i_1 \cdots i_p, j_1 \cdots j_q} = \frac{\partial f_{i_1 \cdots i_p}(X)}{\partial x_{j_1 \cdots j_q}} = \frac{\partial f_{\mathcal{I}_p}(X)}{\partial x_{\mathcal{J}_q}},$$

where $f_{\mathcal{I}_p}(X) = f_{i_1 \cdots i_p}(X) = [F(X)]_{i_1 \cdots i_p} = [F(X)]_{\mathcal{I}_p}$ represents the indexed entry of the output tensor.

A.2.1 Properties of the Jacobian Tensor

Given two tensor-valued functions with the same tensor argument, $F(X)$ and $G(X)$, the following properties hold:

- **Linearity**: The Jacobian of a linear combination of tensor-valued functions satisfies:

$$\frac{\partial}{\partial X}(\alpha F + \beta G) = \alpha \frac{\partial F}{\partial X} + \beta \frac{\partial G}{\partial X},$$

  where α and β are scalars.

- **Product Rule** (also known as *Leibniz rule*): The Jacobian of the Kronecker product satisfies:
$$\frac{\partial}{\partial X}(F \otimes G) = \frac{\partial F}{\partial X} \otimes G + F \otimes \frac{\partial G}{\partial X},$$
where the order of the products is important, as the Kronecker product is not commutative. The Jacobian of the Hadamard product satisfies the following component-wise identity:
$$\left[\frac{\partial}{\partial X}(F \odot G)\right]_{\mathcal{I}_p,\mathcal{J}_q} = \left[\frac{\partial F}{\partial X}\right]_{\mathcal{I}_p\mathcal{J}_q} [G]_{\mathcal{I}_p} + [F]_{\mathcal{I}_p} \left[\frac{\partial G}{\partial X}\right]_{\mathcal{I}_p\mathcal{J}_q}.$$
For the Jacobian of the contraction product, the product rule is more involved, since the order of the subscripts is important. In particular, when F is of order $p+r$, G is of order $r+q$, and X is of order n, we have that (proof left as an exercise):
$$\left[\frac{\partial}{\partial X}(F \overset{r}{:} G)\right]_{(\mathcal{I}_p,\mathcal{J}_q),\mathcal{K}_n} = \sum_{S_r} \left[\frac{\partial F}{\partial X}\right]_{(\mathcal{I}_p,S_r),\mathcal{K}_n} [G]_{S_r,\mathcal{J}_q} + \left[F \overset{r}{:} \frac{\partial G}{\partial X}\right]_{\mathcal{I}_p,\mathcal{J}_q,\mathcal{K}_n}.$$

- **Tensor Chain Rule**: Consider two tensor functions $F(Y)$ and $Y(X)$, where Y is a tensor of order r. The chain rule for the tensor Jacobian is:
$$\frac{\partial F}{\partial X} = \frac{\partial F}{\partial Y} \overset{r}{:} \frac{\partial Y}{\partial X},$$
where the contraction product is taken over the indices of the tensor Y.

Example 4: Tensor Jacobian of the Identity Function

Consider the identity function $F(X) = X$ for a tensor $X \in \mathbb{R}^{n_1 \times \cdots \times n_p}$ of order p. The Jacobian of F with respect to X is a tensor of order $2p$, capturing the partial derivatives of each component of X with respect to each component of X. The components of the Jacobian tensor are given by:
$$\left[\frac{\partial X}{\partial X}\right]_{\mathcal{I}_p,\mathcal{J}_p} = \left[\frac{\partial X}{\partial X}\right]_{i_1\ldots i_p,j_1\ldots j_p} = \frac{\partial x_{i_1\ldots i_p}}{\partial x_{j_1\ldots j_p}} = \delta_{\mathcal{I}_p\mathcal{J}_p} = \left[\mathbb{I}^{(2p)}\right]_{\mathcal{I}_p,\mathcal{J}_p},$$
where $\delta_{\mathcal{I}_p\mathcal{J}_p}$ represents the multi-dimensional Kronecker delta function, which equals 1 if $\mathcal{I}_p = \mathcal{J}_p$ and 0 otherwise.

Example 5: Tensor Jacobian of a Matrix-Vector Product

Consider the matrix-vector product $X \cdot \mathbf{v}$, where $X \in \mathbb{R}^{n\times n}$ and $\mathbf{v} \in \mathbb{R}^n$. We seek to compute the partial derivative of this product with respect to the elements of X:
$$\left[\frac{\partial (X \cdot \mathbf{v})}{\partial X}\right]_{i,j_1j_2} = \frac{\partial}{\partial x_{j_1j_2}} \sum_k x_{ik} v_k = \sum_k \frac{\partial x_{ik}}{\partial x_{j_1j_2}} v_k.$$

Since $\frac{\partial x_{ik}}{\partial x_{j_1 j_2}} = \delta_{ij_1}\delta_{kj_2}$, we have:

$$\left[\frac{\partial (X \cdot \mathbf{v})}{\partial X}\right]_{i,j_1j_2} = \sum_k \delta_{ij_1}\delta_{kj_2} v_k = \delta_{ij_1} v_{j_2} = \left[\mathbb{I}^{(2)}\right]_{ij_1} v_{j_2}.$$

Thus, we obtain the following identity in tensor form:

$$\frac{\partial (X \cdot \mathbf{v})}{\partial X} = \mathbb{I}^{(2)} \otimes \mathbf{v}.$$

This result shows that the derivative of the matrix-vector product $X \cdot \mathbf{v}$ with respect to X can be expressed as the Kronecker product of the identity tensor $\mathbb{I}^{(2)}$ and the vector $\mathbf{v}$. The Kronecker product ensures that the derivative accounts for the appropriate mapping of indices between the matrix X and the vector $\mathbf{v}$.

103

Example 6: Jacobian Tensor of Matrix Products

In this example, we compute the Jacobian tensor of the product $U \cdot F(X) \cdot \mathbf{v}$, where $U \in \mathbb{R}^{p\times n}$, $F(X) \in \mathbb{R}^{n\times n}$, and $\mathbf{v} \in \mathbb{R}^n$. We aim to show that:

$$\frac{\partial (U \cdot F(X) \cdot \mathbf{v})}{\partial X} = U{:}P^{\sigma}\left(\frac{\partial F(X)}{\partial X}\right){:}\mathbf{v},$$

which is a tensor of order 3 and σ denotes the permutation that rearranges the index sequence $(1,2,3,4)$ into $(1,3,4,2)$.
To compute the partial derivative of this product with respect to the entries of the matrix X, we begin by expressing the individual elements as:

$$[U \cdot F(X) \cdot \mathbf{v}]_i = \sum_{k_1,k_2} u_{ik_1} [F(X)]_{k_1k_2} v_{k_2}.$$

We then take the partial derivative with respect to X:

$$\begin{aligned}
\left[\frac{\partial (U \cdot F(X) \cdot \mathbf{v})}{\partial X}\right]_{i,j_1j_2} &= \frac{\partial}{\partial x_{j_1j_2}} \sum_{k_1,k_2} u_{ik_1} [F(X)]_{k_1k_2} v_{k_2} \\
&= \sum_{k_1,k_2} u_{ik_1} \frac{\partial [F(X)]_{k_1k_2}}{\partial x_{j_1j_2}} v_{k_2} \\
&= \sum_{k_1,k_2} u_{ik_1} \left[\frac{\partial F(X)}{\partial X}\right]_{k_1k_2,j_1j_2} v_{k_2} \\
&= \sum_{k_1,k_2} u_{ik_1} \left[P^{\sigma}\left(\frac{\partial F(X)}{\partial X}\right)\right]_{k_1j_1j_2k_2} v_{k_2},
\end{aligned}$$

where σ denotes the permutation that rearranges the index sequence $(1,2,3,4)$ into $(1,3,4,2)$, ensuring that the resulting tensor maintains the correct structure. Thus, the Jacobian tensor of the product satisfies the identity at the beginning of the example.

104

105

Example 7: Jacobian Tensor of Power Matrix

In this example, we seek to prove that the Jacobian tensor of the power of a matrix, X^p, satisfies

$$\frac{\partial X^p}{\partial X} = \sum_{q=1}^{p} P^{\sigma}\left(X^{q-1} \otimes X^{p-q}\right),$$

where σ permutes the sequence $(1,4,2,3)$ into $(1,2,3,4)$.
We prove this identity by induction. We start with the following decomposition:

$$\left[\frac{\partial X^p}{\partial X}\right]_{i_1 i_2, j_1 j_2} = \left[\frac{\partial\left(X \cdot X^{p-1}\right)}{\partial X}\right]_{i_1 i_2, j_1 j_2} = \frac{\partial\left[X \cdot X^{p-1}\right]_{i_1 i_2}}{\partial x_{j_1 j_2}}.$$

Expanding the matrix product, we obtain:

$$\left[X \cdot X^{p-1}\right]_{i_1 i_2} = \sum_k x_{i_1 k}\left[X^{p-1}\right]_{k i_2}.$$

Hence,

$$\frac{\partial\left[X^p\right]_{i_1 i_2}}{\partial x_{j_1 j_2}} = \frac{\partial}{\partial x_{j_1 j_2}} \sum_k x_{i_1 k}\left[X^{p-1}\right]_{k i_2} = \sum_k \frac{\partial}{\partial x_{j_1 j_2}}\left(x_{i_1 k}\left[X^{p-1}\right]_{k i_2}\right).$$

Applying the product rule for derivatives, this becomes:

$$\left[\frac{\partial X^p}{\partial X}\right]_{i_1 i_2, j_1 j_2} = \sum_k \frac{\partial x_{i_1 k}}{\partial x_{j_1 j_2}}\left[X^{p-1}\right]_{k i_2} + \sum_k x_{i_1 k} \frac{\partial\left[X^{p-1}\right]_{k i_2}}{\partial x_{j_1 j_2}}.$$

Since $\frac{\partial x_{i_1 k}}{\partial x_{j_1 j_2}} = \delta_{i_1 j_1}\delta_{k j_2}$, we obtain:

$$\left[\frac{\partial X^p}{\partial X}\right]_{i_1 i_2, j_1 j_2} = \delta_{i_1 j_1}\left[X^{p-1}\right]_{j_2 i_2} + \sum_k x_{i_1 k}\left[\frac{\partial X^{p-1}}{\partial X}\right]_{k i_2, j_1 j_2},$$

which is a recursion that expresses $\frac{\partial X^p}{\partial X}$ in terms of $\frac{\partial X^{p-1}}{\partial X}$. To solve this recursion, we start with the initial case:

$$\left[\frac{\partial X}{\partial X}\right]_{k i_2, j_1 j_2} = \delta_{k j_1}\delta_{i_2 j_2}.$$

To obtain a general expression for $\frac{\partial X^p}{\partial X}$, we use induction as follows:

- For $p = 2$:

$$\left[\frac{\partial X^2}{\partial X}\right]_{i_1 i_2, j_1 j_2} = \delta_{i_1 j_1} x_{j_2 i_2} + \sum_k x_{i_1 k}\delta_{k j_1}\delta_{i_2 j_2} = \delta_{i_1 j_1} x_{j_2 i_2} + x_{i_1 j_1}\delta_{i_2 j_2}.$$

106

- For $p = 3$:

$$\left[\frac{\partial X^3}{\partial X}\right]_{i_1 i_2, j_1 j_2} = \delta_{i_1 j_1}\left[X^2\right]_{j_2 i_2} + \sum_k x_{i_1 k}\left(\delta_{k j_1} x_{j_2 i_2} + x_{k j_1}\delta_{i_2 j_2}\right)$$
$$= \delta_{i_1 j_1}\left[X^2\right]_{j_2 i_2} + x_{i_1 j_1} x_{j_2 i_2} + \delta_{i_2 j_2}\left[X^2\right]_{i_1 j_1}.$$

- For $p = 4$:

$$\left[\frac{\partial X^4}{\partial X}\right]_{i_1 i_2, j_1 j_2} = \delta_{i_1 j_1}\left[X^3\right]_{j_2 i_2} + x_{i_1 j_1}\left[X^2\right]_{j_2 i_2} + \left[X^2\right]_{i_1 j_1} x_{j_2 i_2} + \left[X^3\right]_{i_1 j_1}\delta_{i_2 j_2}.$$

By induction, we obtain the general form:

$$\left[\frac{\partial X^p}{\partial X}\right]_{i_1 i_2, j_1 j_2} = \sum_{q=1}^{p}\left[X^{q-1}\right]_{i_1 j_1}\left[X^{p-q}\right]_{j_2 i_2} = \sum_{q=1}^{p}\left[X^{q-1}\otimes X^{p-q}\right]_{i_1 j_1 j_2 i_2}.$$

Notice that the order of the indices is not appropriate to render a tensor equality. This can be easily fixed by applying a permutation operator to the tensor product, such as:

$$\left[P^{\sigma}\left(X^{q-1}\otimes X^{p-q}\right)\right]_{i_1 i_2, j_1 j_2} = \left[X^{q-1}\otimes X^{p-q}\right]_{i_1 j_1 j_2 i_2},$$

where σ permutes the sequence $(1,2,3,4)$ into $(1,4,2,3)$. Therefore, we obtain the compact expression at the beginning of the example.

TBD: Hidden notes

Appendix B

Expression for the Jacobians of $\mathbf{x}_k$ w.r.t. A

This section derives a recursive expression for the Jacobian tensor of the hidden state $\mathbf{x}_k$ with respect to the parameter matrix A in a recurrent neural network, incorporating the effects of the nonlinear activation function σ.

Lemma 1. *The Jacobian tensor of the hidden state $\mathbf{x}_k$ with respect to the parameter matrix A in a recurrent neural network satisfies the recursive expression:*

$$\boxed{\frac{\partial \mathbf{x}_k}{\partial A} = \Sigma_{k-1}^{(3)} \odot \left(\mathbb{I}^{(2)} \otimes \mathbf{x}_{k-1} + A{:}\frac{\partial \mathbf{x}_{k-1}}{\partial A} \right),}$$

where $\Sigma_{k-1}^{(3)}$ is a tensor with entries $\sigma'(z_{k-1}^{(i)})$, $\mathbb{I}^{(2)}$ is the second-order identity tensor, and $A{:}\frac{\partial \mathbf{x}_{k-1}}{\partial A}$ denotes the contraction of A with the Jacobian of $\mathbf{x}_{k-1}$.

Proof. The hidden state update is given by:

$$\mathbf{x}_k = \sigma(A\mathbf{x}_{k-1} + B\mathbf{u}_{k-1} + \mathbf{b}),$$

where σ is a nonlinear activation function applied element-wise. To compute the Jacobian tensor $\frac{\partial \mathbf{x}_k}{\partial A}$, we consider the individual entries:

$$\left[\frac{\partial \mathbf{x}_k}{\partial A} \right]_{i,j_1j_2} = \frac{\partial \left[\mathbf{x}_k\right]_i}{\partial a_{j_1j_2}}.$$

Differentiating through the activation function σ, we have:

$$\frac{\partial \left[\mathbf{x}_k\right]_i}{\partial a_{j_1j_2}} = \sigma'\left(z_{k-1}^{(i)}\right) \frac{\partial z_{k-1}^{(i)}}{\partial a_{j_1j_2}},$$

where $z_{k-1}^{(i)} = \left[A\mathbf{x}_{k-1} + B\mathbf{u}_{k-1} + \mathbf{b}\right]_i$. Since $B\mathbf{u}_{k-1}$ and $\mathbf{b}$ do not depend on A, their derivatives vanish, leaving:

$$\frac{\partial z_{k-1}^{(i)}}{\partial a_{j_1j_2}} = \frac{\partial \left[A\mathbf{x}_{k-1}\right]_i}{\partial a_{j_1j_2}}.$$

Applying the product rule for differentiation:

$$\frac{\partial z_{k-1}^{(i)}}{\partial a_{j_1 j_2}} = \sum_s \frac{\partial a_{is} x_{k-1}^{(s)}}{\partial a_{j_1 j_2}}.$$

The product rule yields:

$$\frac{\partial z_{k-1}^{(i)}}{\partial a_{j_1 j_2}} = \sum_s \frac{\partial a_{is}}{\partial a_{j_1 j_2}} x_{k-1}^{(s)} + \sum_s a_{is} \frac{\partial x_{k-1}^{(s)}}{\partial a_{j_1 j_2}}.$$

The first term evaluates to $\delta_{ij_1} x_{k-1}^{(j_2)}$, where δ_{ij_1} is the Kronecker delta. Thus:

$$\frac{\partial z_{k-1}^{(i)}}{\partial a_{j_1 j_2}} = \delta_{ij_1} x_{k-1}^{(j_2)} + \sum_s a_{is} \frac{\partial x_{k-1}^{(s)}}{\partial a_{j_1 j_2}}.$$

Substituting back into the Jacobian:

$$\left[\frac{\partial \mathbf{x}_k}{\partial A}\right]_{i,j_1 j_2} = \sigma'\left(z_{k-1}^{(i)}\right)\left(\delta_{ij_1} x_{k-1}^{(j_2)} + \sum_s a_{is} \frac{\partial x_{k-1}^{(s)}}{\partial a_{j_1 j_2}}\right).$$

The term $\delta_{ij_1} x_{k-1}^{(j_2)}$ can be expressed as $[\mathbb{I}^{(2)} \otimes \mathbf{x}_{k-1}]_{i,j_1 j_2}$, and the second term as $\left[A{:}\frac{\partial \mathbf{x}_{k-1}}{\partial A}\right]_{i,j_1 j_2}$. Thus:

$$\left[\frac{\partial \mathbf{x}_k}{\partial A}\right]_{i,j_1 j_2} = \sigma'\left(z_{k-1}^{(i)}\right)\left([\mathbb{I}^{(2)} \otimes \mathbf{x}_{k-1}]_{i,j_1 j_2} + \left[A{:}\frac{\partial \mathbf{x}_{k-1}}{\partial A}\right]_{i,j_1 j_2}\right).$$

Finally, writing the result in tensor form:

$$\frac{\partial \mathbf{x}_k}{\partial A} = \Sigma_{k-1}^{(3)} \odot \left(\mathbb{I}^{(2)} \otimes \mathbf{x}_{k-1} + A{:}\frac{\partial \mathbf{x}_{k-1}}{\partial A}\right),$$

where $[\Sigma_{k-1}^{(3)}]_{i,j_1 j_2} = \sigma'(z_{k-1}^{(i)})$. This completes the proof. □