Topic 2B: State-Space Models for Time Series Forecasting

Victor M. Preciado

Contents

1 State-Space Models for Time Series

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:

$$\mathbf{X}_{k+1} = f_{\boldsymbol{\theta}_x}(\mathbf{X}_k, \mathbf{u}_k, \boldsymbol{\eta}_k), \quad \text{with initial condition } \mathbf{X}_0 = \mathbf{x}_0, \tag{1}$$

  where $\mathbf{X}_k$ is a random vector[1] called the **hidden state** at time k and $\mathbf{u}_k$

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

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 $\boldsymbol{\eta}_k$ 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:

$$\mathbf{Y}_k = g_{\boldsymbol{\theta}_y}(\mathbf{X}_k, \mathbf{u}_k, \boldsymbol{\varepsilon}_k), \tag{2}$$

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 $\boldsymbol{\varepsilon}_k$ 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: Hodgkin-Huxley Model

The Hodgkin-Huxley model describes the dynamical behavior of a biological neuron using a four-dimensional state vector. Let us represent the states as follows: $\mathbf{x}_k = [x_{k1}, x_{k2}, x_{k3}, x_{k4}]^\intercal$, where x_{k1} is the membrane potential and x_{k2}, x_{k3}, and x_{k4} are hidden gating variables. The evolution of these states over time is governed by a set of ordinary differential equations that can be discretized as follows:

- **Membrane potential** x_{k1} (discretized form):

$$\begin{aligned} x_{k+1,1} = x_{k1} + \frac{\Delta t}{C_m}\Big(u_k - \gamma_1 x_{k3}^3 x_{k4}(x_{k1} - \delta_1) \\ - \gamma_2 x_{k2}^4 (x_{k1} - \delta_2) - \gamma_3 (x_{k1} - \delta_3)\Big) + \eta_k, \end{aligned}$$

where the γ's and δ's are model parameters, u_k is an external input,

and η_k is the process noise accounting for random perturbations.

- **Hidden gating variables:** The gating variables x_{k2}, x_{k3}, and x_{k4} evolve according to the following discretized update equations, where $\sigma_2(x_{k1})$, $\sigma_3(x_{k1})$, and $\sigma_4(x_{k1})$ represent three different sigmoidal functions of the membrane potential x_{k1}:

$$
\begin{aligned}
x_{k+1,2} &= x_{k2} + \Delta t \cdot (\sigma_2(x_{k1})(1 - x_{k2}) - (1 - \sigma_2(x_{k1})) \cdot x_{k2}) \,, \\
x_{k+1,3} &= x_{k3} + \Delta t \cdot (\sigma_3(x_{k1})(1 - x_{k3}) - (1 - \sigma_3(x_{k1})) \cdot x_{k3}) \,, \\
x_{k+1,4} &= x_{k4} + \Delta t \cdot (\sigma_4(x_{k1})(1 - x_{k4}) - (1 - \sigma_4(x_{k1})) \cdot x_{k4}) \,.
\end{aligned}
$$

  The sigmoidal functions describe the probability of the gating variables opening or closing based on the membrane potential x_{k1}.

- **Observation Equation:** The observed output y_k at each time step is a noisy measurement of the membrane potential x_{k1}: $y_k = x_{k1} + \varepsilon_k$, where ε_k is the observation noise.

Note that the set of discretized equations described above can be written as a standard state-space model in (1) and (2). The hidden latent state variables are both the membrane potential and the gating variables, while the output is a noisy version of the membrane potential. In the following figure, we plot the joint evolution of the three hidden gating variables in a 3D plot (left) and the membrane potential (right) for a particular choice of parameters (code available in GitHub).

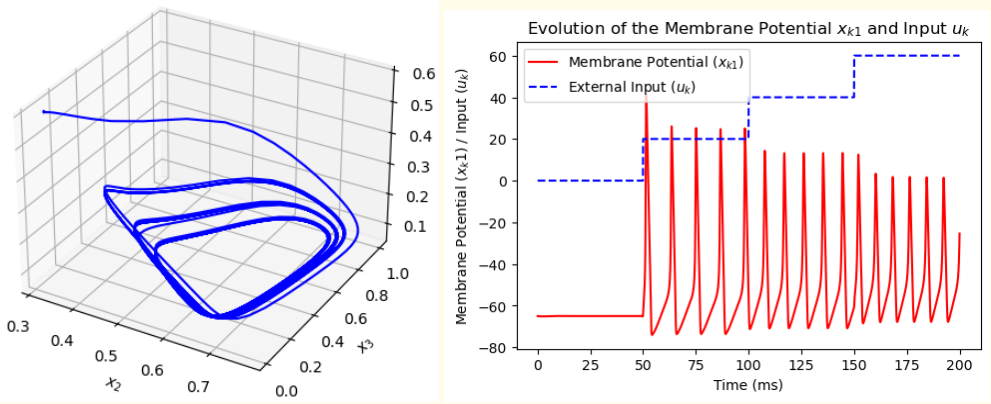

Figure 1: (Left) 3D plot for the joint evolution of gating variables, x_{k2}, x_{k3}, and x_{k4}. (Right) Evolution of the membrane potential x_{k1} and the input signal u_k.

1.2 Hidden Markov Models (HMMs)

Building on the state-space framework, **Hidden Markov Models (HMMs)** offer a specialized yet powerful instance of state-space models, where the hidden states follow a Markov process over a *discrete* set of H possible hidden states, $\mathcal{S} = \{s_1, \ldots, s_H\}$. Like general state-space models, HMMs rely on hidden states that evolve over time according to certain probabilistic rules, while the observed data are linked to the hidden states via a probabilistic observation process. The main distinction between HMMs and SSMs is that the hidden states in an HMM take values from a discrete set, making them particularly suited for modeling systems where the underlying process can be represented by a finite number of states.

Although HMMs can produce either discrete or continuous outputs, we will focus on the case of discrete observations. In particular, at each time step k, the current hidden state X_k generates an observable output Y_k, drawn from a discrete set of M possible outcomes, i.e., $Y_k \in \mathcal{O} = \{o_1, o_2, \ldots, o_M\}$, according to a predefined **emission probability distribution**. This distribution specifies the conditional probability of observing a given output Y_k based on the current hidden state X_k.

Similar to a state-space model, we can characterize an HMM by two main components:

1. **State Transition Process:** The hidden state X_k evolves according to a *time-homogeneous Markov chain* $\mathcal{X} = (X_0, X_1, \ldots, X_L)$, where $X_k \in \mathcal{S} = \{s_1, s_2, \ldots, s_H\}$ for all k. The probabilities of transitioning between states are described by a row-stochastic **state transition matrix** $P \in [0,1]^{H \times H}$, where each element p_{ij} represents the probability of moving from state s_i to state s_j. Specifically,

$$[P]_{ij} = p_{ij} = \mathbb{P}(X_{k+1} = s_j \mid X_k = s_i), \quad \text{for all } i, j \in \{1, \ldots, H\}.$$

   The process starts with an **initial state distribution**, represented by the vector:

$$\boldsymbol{\pi}_0 = \big[\mathbb{P}(X_0 = s_1), \quad \mathbb{P}(X_0 = s_2), \quad \cdots, \quad \mathbb{P}(X_0 = s_H)\big]^\intercal \in [0,1]^H,$$

   where the i-th entry $\pi_{0,i}$ gives the probability that the process begins in state s_i. This initial distribution, while not directly observable, strongly influences the system?s future evolution. According to the propagation rules for homogeneous Markov chains, the state distribution at time k is given by:

$$\boxed{\boldsymbol{\pi}_k = P^\intercal \boldsymbol{\pi}_{k-1} = (P^\intercal)^k \boldsymbol{\pi}_0.}$$

2. **Observation Process:** At each time step k, the HMM generates an output $Y_k \in \mathcal{O} = \{o_1, o_2, \ldots, o_E\}$ stochastically, based on the hidden state X_k.

This relationship is governed by a set of **emission probabilities**, $\{c_{he} \colon h = [H];\, e = [E]\}$, where each element c_{he} represents the conditional probability of observing $Y_k = o_e$ given that $X_k = s_h$. These emission probabilities can be organized into an **emission matrix** $C \in [0,1]^{H \times E}$. Formally, the entries of this matrix are defined as:

$$[C]_{he} = c_{he} = \mathbb{P}(Y_k = o_e \mid X_k = s_h), \quad \text{for all } h \in \{1, \dots, H\},\, e \in \{1, \dots, E\}.$$

Since the HMM generates an output at every time step k, the emission matrix C is a row-stochastic matrix, meaning that the sum of the probabilities across each row is 1, i.e., $\sum_{e=1}^{E} C_{he} = 1$ for all $h \in \{1, \dots, H\}$. Using the total probability theorem, we can express the probability of observing a specific output $Y_k = o_e$, given the initial information set $\mathcal{F}_0$, as:

$$\mathbb{P}(Y_k = o_e \mid \mathcal{F}_0) = \sum_{h=1}^{H} \underbrace{\mathbb{P}(Y_k = o_e \mid X_k = s_h)}_{c_{he}} \underbrace{\mathbb{P}(X_k = s_h \mid \mathcal{F}_0)}_{\pi_{k,h}}. \tag{3}$$

Now, define the vector $\boldsymbol{\chi}_k$, whose entries represent the probabilities of observing each possible output given the initial distribution, as follows:

$$\boldsymbol{\chi}_k = \begin{bmatrix} \mathbb{P}(Y_k = o_1 \mid \mathcal{F}_0) & \mathbb{P}(Y_k = o_2 \mid \mathcal{F}_0) & \cdots & \mathbb{P}(Y_k = o_E \mid \mathcal{F}_0) \end{bmatrix}^{\intercal}.$$

Thus, using matrix multiplication, we can rewrite (3) in vector form as:

$$\boxed{\boldsymbol{\chi}_k = C^{\intercal} \boldsymbol{\pi}_k = C^{\intercal} (P^{\intercal})^k \boldsymbol{\pi}_0.}$$

This formulation encapsulates the probabilistic nature of both the state transitions and the observations, providing a powerful framework for modeling sequential data where the underlying structure is not directly observable. HMMs excel in scenarios where the system alternates between distinct regimes that are not directly observable but can be inferred from the data. They have been widely applied in areas such as speech recognition, biological sequence analysis, and financial market modeling.

Example 2: Disease Progression as a Hidden Markov Model

In epidemiology, we can model the progression of a disease in a particular individual using HMMs, as follows.

1. **Hidden States:** Define a set of hidden states such that each state represents the health status of an individual. For example, we could use the following set:

   $$X_k \in \{s_1 = \texttt{Susceptible}, s_2 = \texttt{Infected}, s_3 = \texttt{Recovered}, s_4 = \texttt{Dead}\}.$$

These hidden states evolve over time according to a Markov process, capturing how an individual transitions from being susceptible to infected, to either recovery or death.

2. **State Transition Matrix:** The matrix P governs the transitions between the different health states. As an example:

$$P = \begin{bmatrix} 0.9 & 0.1 & 0 & 0 \\ 0 & 0.79 & 0.2 & 0.01 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where each row represents the probabilities of transitioning from one health state to another. For instance, an individual who is currently `Susceptible` has a 90% chance of remaining healthy and a 10% chance of becoming `Infected`.

Insert diagram for example.

3. **Observations:** The observations represent the measurable behaviors or conditions of the individual, such as being masked, visiting a hospital, or being in the mortuary. The set of possible observations in this example is:

$$Y_k \in \{o_1 = \texttt{Unmasked}, o_2 = \texttt{Masked}, o_3 = \texttt{Hospitalized}, o_4 = \texttt{Mortuary}\},$$

These observations provide indirect evidence of the individual's underlying health state.

4. **Emission Probability Matrix:** The matrix C defines the probability of observing each behavior given the individual's hidden health state. For example:

$$C = \begin{bmatrix} 0.8 & 0.2 & 0 & 0 \\ 0.2 & 0.5 & 0.3 & 0 \\ 0.9 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where each row corresponds to the observation probabilities conditioned on a hidden health state. For instance, if the individual is `Infected`, there is a 50% chance they will wear a mask, a 30% chance they will be hospitalized, and a 20% chance they show no symptoms.

5. **Initial State Distribution:** The vector of initial state probabilities $\boldsymbol{\pi}_0$ provides the probabilities of an individual starting in each health state. For example:

$$\boldsymbol{\pi}_0 = \begin{bmatrix} 0.95 & 0.05 & 0 & 0 \end{bmatrix}^\intercal,$$

meaning that at the beginning, there is a 95% chance the individual is `Healthy` and a 5% chance they are already `Infected`.

6. **Evolution of Emission Probabilities:** The vector $\boldsymbol{\chi}_k = C^\intercal (P^\intercal)^k \boldsymbol{\pi}_0$ represents the evolution of the emission probabilities over time, i.e., how likely we are to observe certain outputs at each time step k. For example, at time $k = 1$, we can compute $\boldsymbol{\chi}_1$ as follows:

$$\boldsymbol{\pi}_1 = (P^\intercal)\boldsymbol{\pi}_0 = \begin{bmatrix} 0.855 \\ 0.095 \\ 0.05 \\ 0 \end{bmatrix}, \quad \boldsymbol{\chi}_1 = C^\intercal \boldsymbol{\pi}_1 = \begin{bmatrix} 0.704 \\ 0.2095 \\ 0.086 \\ 0 \end{bmatrix}.$$

Repeating this process for subsequent k values gives the evolution of the emission probabilities over time, reflecting how the individual's health state influences observable behaviors. In Fig. 2, we show the evolution of the hidden states and the observable behaviors for 50 time steps.

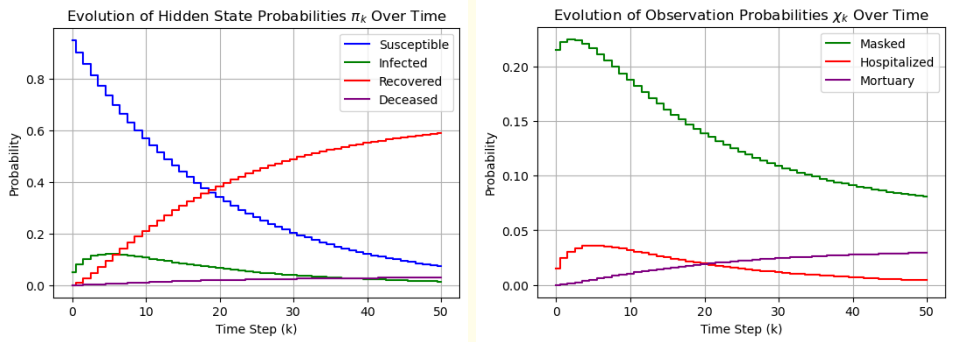

Figure 2: (Left) Evolution of the probabilities of being in each hidden state (e.g., Susceptible, Infected, Recovered, or Deceased). (Right) Evolution of the probabilities of observing a particular measurable behavior (e.g., Masked, Hospitalized, or Mortuary).

1.2.1 Parameter Estimation in Hidden Markov Models ⚠

EM ALGORITHM...

2 Linear State-Space Models for Time Series Analysis

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:

$$\mathbf{X}_{k+1} = A_k \mathbf{X}_k + B_k \mathbf{u}_k + \boldsymbol{\eta}_k, \quad \text{with initial condition } \mathbf{X}_0 = \mathbf{x}_0,$$

  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 $\boldsymbol{\eta}_k$ represents **process noise**, typically modeled as a multivariate white Gaussian noise, i.e., $\boldsymbol{\eta}_k \sim_{\text{iid}} \mathcal{N}(\mathbf{0}, Q)$, where Q is the process noise covariance matrix. This state equation governs the system's hidden state dynamics, describing how the state vector

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

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:

$$\mathbf{Y}_k = C_k \mathbf{X}_k + D_k \mathbf{u}_k + \boldsymbol{\varepsilon}_k,$$

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, 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 $\boldsymbol{\varepsilon}_k$ represents **measurement noise**, typically modeled as a multivariate white Gaussian noise (independent of the process noise), i.e., $\boldsymbol{\varepsilon}_k \sim_{\text{iid}} \mathcal{N}(\mathbf{0}, R)$, 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. 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. 3.

Insert diagram.

Figure 3: Block diagram representation of the linear state-space equations, where the D-block represents a one-time-step delay, i.e., $D\,\boldsymbol{X}_{k+1} = \boldsymbol{X}_k$.

Example 3: Autoregressive model as an LSSM

Consider the AR(p) process given by:

$$Y_{k+1} = \phi_1 Y_k + \phi_2 Y_{k-1} + \cdots + \phi_p Y_{k-p+1} + \epsilon_k, \quad \text{where } \epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2).$$

We can express this AR(p) process as a Linear State-Space Model (LSSM) by defining the state vector, state equation, and measurement equation as follows.

1. **State Equation**: The state vector $\mathbf{X}_k^{\text{AR}} \in \mathbb{R}^p$ is defined as:

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

The state equation describes how the state vector evolves over time. Specifically, the state transition equation is given by:

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

where A^{AR} is the state transition matrix and $\boldsymbol{\varepsilon}_k^{\mathrm{AR}}$ is the noise vector.

2. **Measurement Equation**: The measurement equation relates the state vector $\mathbf{X}_k^{\mathrm{AR}}$ to the observed data Y_k. The observation equation is:

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

where C^{AR} is the observation matrix, which extracts the first element of the state vector (i.e., the current value Y_k).

Example 4: Moving Average as an LSSM

Consider the MA(q) process given by:

$$Y_k = \epsilon_k + \theta_1 \epsilon_{k-1} + \cdots + \theta_q \epsilon_{k-q}, \quad \text{where } \epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2).$$

We can express this MA(q) process as a Linear State-Space Model (LSSM) by defining the state vector, state equation, and measurement equation as follows.

1. **State Equation**: The state vector $\mathbf{X}_k^{\mathrm{MA}} \in \mathbb{R}^{q+1}$ is defined as:

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

The state equation describes how the state vector evolves over time.

Specifically, the state transition equation is given by:

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

where A^{MA} is the state transition matrix and $\boldsymbol{\varepsilon}_k^{\mathrm{MA}}$ is the noise vector.

2. **Measurement Equation**: The measurement equation relates the state vector $\mathbf{X}_k^{\mathrm{MA}}$ to the observed data Y_k. The observation equation is:

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

where C^{MA} is the observation matrix. This matrix applies the MA coefficients $\theta_1, \theta_2, \ldots, \theta_q$ to the lagged noise terms and adds them to the current noise ϵ_k.

Example 5: Autoregressive Moving-Average Model as an LSSM

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

$$Y_k = \phi_1 Y_{k-1} + \phi_2 Y_{k-2} + \epsilon_k + \theta_1 \epsilon_{k-1} + \theta_2 \epsilon_{k-2}, \quad \text{where } \epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2).$$

where ϕ_1, ϕ_2 are the autoregressive coefficients, while θ_1, θ_2 are the moving average coefficients.
To represent this ARMA(2,2) process as a Linear State-Space Model (LSSM), we use an augmented state vector that includes both past observations and past error terms.

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 the state vector:

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

2. **Measurement Equation**: The measurement equation relates the state vector to the observed output:

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

Thus, the observed output is directly taken from the first component of the state vector:

$$Y_k = Y_k.$$

This representation captures both the autoregressive and moving average components within a unified state-space framework, enabling the application of state-space analysis techniques to the ARMA(2,2) model.

2.1 Temporal Evolution of the LSSM

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:

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k, \tag{4}$$
$$\mathbf{y}_k = C\mathbf{x}_k + D\mathbf{u}_k, \tag{5}$$

with given initial condition $\mathbf{x}_0$. The state equation (4) 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:

$$\mathbf{x}_k = A^k \mathbf{x}_0 + \sum_{i=1}^{k} A^{i-1} B \mathbf{u}_{k-i}. \tag{6}$$

This expression illustrates how the current state depends on the initial state $\mathbf{x}_0$ and the history of inputs $\mathbf{u}_i$, and the cumulative effect of the process noise $\boldsymbol{\eta}_i$.

The output equation (5) 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 (6) into the output equation, we have:

$$\boxed{\mathbf{y}_k = CA^k \mathbf{x}_0 + \sum_{i=1}^{k} CA^{i-1} B \mathbf{u}_{k-i} + D\mathbf{u}_k.} \tag{7}$$

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:

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

Using the Markov parameters, the summation terms in (7) 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:

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

In the context of state-space models, the summation term in (7) 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:

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

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.

Insert diagram.

Figure 4: Pictorial representation of the convolution of two temporal sequences.

2.2 Similarity Transformations of LSSMs

In a Linear State-Space Model (LSSM), the system's input-output behavior—how input sequence $(\mathbf{u}_k)_{k\geq 0}$ is mapped to an output sequence $(\mathbb{Y}_k)_{k\geq 0}$—encapsulates its fundamental observable dynamics. However, the internal state representation, defined by the state vector $\mathbf{x}_k$, serves as an abstract mathematical construct and is not uniquely determined. The specific realization of the internal state may vary, as multiple distinct internal representations can describe the same input-output behavior. In fact, for any given input-output relationship, there exists an infinite family of state-space realizations that yield the same external dynamics but differ in their internal state descriptions. To demonstrate this, we will introduce a formal mechanism that modifies the internal state representation while preserving the input-output mapping, thereby illustrating that the observable behavior remains invariant despite changes in the internal realization.

For simplicity, let us consider the noiseless LSSM, described by:

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

where we assume that there are n hidden states, i.e., $\mathbf{x}_k \in \mathbb{R}^n$, the inputs are r-dimensional vectors $\mathbf{u}_k \in \mathbb{R}^r$, and the outputs are m-dimensional vectors $\mathbf{y}_k \in \mathbb{R}^m$. Here, $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times r}$, $C \in \mathbb{R}^{m\times n}$, and $D \in \mathbb{R}^{m\times r}$ are the system matrices. We will demonstrate that there exists an infinite family of LSSMs that can realize the same input-output mapping as the system above. Specifically, for any given sequence of inputs, the same sequence of outputs can be generated using a different collection of system matrices.

To construct an LSSM that realizes the same input-output mapping as the system defined by matrices (A, B, C, D), we apply a similarity transformation to the state vector. Let $T \in \mathbb{R}^{n\times n}$ be an invertible matrix and define a new state vector $\boldsymbol{\xi}_k$ as follows:

$$\boldsymbol{\xi}_k = T^{-1}\mathbf{x}_k.$$

Substituting this into the state-space equations, we can rewrite the LSSM in terms of the new state vector $\boldsymbol{\xi}_k$ as follows:

$$\begin{aligned}\boldsymbol{\xi}_{k+1} = T^{-1}\mathbf{x}_{k+1} = T^{-1}A\mathbf{x}_k + T^{-1}B\mathbf{u}_k &= \overbrace{T^{-1}AT}^{\widetilde{A}}\boldsymbol{\xi}_k + \overbrace{T^{-1}B}^{\widetilde{B}}\mathbf{u}_k,\\ \mathbf{y}_k = C\mathbf{x}_k + D\mathbf{u}_k &= \underbrace{CT}_{\widetilde{C}}\boldsymbol{\xi}_k + D\mathbf{u}_k.\end{aligned}$$

Thus, the transformed LSSM is defined by the new system matrix $(\widetilde{A}, \widetilde{B}, \widetilde{C}, \widetilde{D})$, i.e., the state and output equations of the transformed LSSM are given by:

$$\boldsymbol{\xi}_{k+1} = \widetilde{A}\boldsymbol{\xi}_k + \widetilde{B}\mathbf{u}_k, \quad \mathbf{y}_k = \widetilde{C}\boldsymbol{\xi}_k + D\mathbf{u}_k.$$

Note that the initial state of the transformed LSSM is also transformed as $\mathbf{x}_0 = T\boldsymbol{\xi}_0$.

To demonstrate that the input-output mapping is invariant under this transformation, we compare the input-output behavior of the original system with that of the transformed system. Using (7) to compute the output to the new LSSM with system matrices $(\widetilde{A}, \widetilde{B}, \widetilde{C}, \widetilde{D})$, we compute the output $\widetilde{\mathbf{y}}_k$ as follows:

$$\begin{aligned}\widetilde{\mathbf{y}}_k &= \widetilde{C}\widetilde{A}^k\boldsymbol{\xi}_0 + \sum_{i=1}^{k} \widetilde{C}\widetilde{A}^{i-1}\widetilde{B}\mathbf{u}_{k-i} + D\mathbf{u}_k \\ &= CT(T^{-1}AT)^k\boldsymbol{\xi}_0 + \sum_{i=1}^{k} CT(T^{-1}AT)^{i-1}T^{-1}B\mathbf{u}_{k-i} + D\mathbf{u}_k. \end{aligned} \tag{8}$$

Using the following three identities: $(T^{-1}AT)^p = T^{-1}A^pT$ for all $p \geq 0$, $\boldsymbol{\xi}_0 = T^{-1}\mathbf{x}_0$ and $TT^{-1} = T^{-1}T = \mathbb{I}_n$, we can simplify (8) as follows:

$$\widetilde{\mathbf{y}}_k = CA^k\mathbf{x}_0 + \sum_{i=1}^{k} CA^{i-1}B\mathbf{u}_{k-i} + D\mathbf{u}_k = \mathbf{y}_k, \tag{9}$$

where the last equality comes from (7). Therefore, the output $\widetilde{\mathbf{y}}_k$ of the transformed LSSM with system matrices $(\widetilde{A}, \widetilde{B}, \widetilde{C}, \widetilde{D})$ is identical to the output $\mathbf{y}_k$ of the original LSSM with system matrices (A, B, C, D), for all invertible transformation matrices $T \in \mathbb{R}^{n\times n}$. In other words, the input-output relationship of any LSSM is preserved under this type of transformations, called **similarity transformations**, indicating that the input-output behavior of the system is invariant to changes in the internal state representation.

This similarity transformation reveals the existence of an infinite family of equivalent LSSMs, all of which produce the same input-output behavior. Therefore, when identifying the system matrices of an LSSM from input/output data, we are effectively searching for one point on a manifold[3] of solutions, where each point corresponds to a distinct internal representation of the system.

[3]A manifold is a mathematical structure that generalizes the concept of curves and surfaces to higher dimensions, providing a framework for continuous transformations between different coordinate systems or representations.

Example 6: Canonical Forms of an LSSM

Consider the following (noiseless) Linear State-Space Model (LSSM):

$$\mathbf{x}_{k+1} = \underbrace{\begin{bmatrix} a_1 & a_2 & a_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}}_{A_c} \mathbf{x}_k + \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}}_{B_c} u_k,$$

$$y_k = \underbrace{\begin{bmatrix} b_0 & b_1 & b_2 \end{bmatrix}}_{C_c} \mathbf{x}_k,$$

where $\mathbf{x}_k \in \mathbb{R}^3$ is the hidden state vector, $u_k \in \mathbb{R}$ represents the scalar input, and $y_k \in \mathbb{R}$ is the scalar output. The system is fully characterized by the matrices (A_c, B_c, C_c) with $D_c = 0$. This particular configuration of the system matrices is known as the **controllable canonical form** and is particularly useful when designing input signals to design input signals to drive the hidden states towards desired values.

An alternative internal representation that preserves the input-output mapping is given by:

$$\boldsymbol{\xi}_{k+1} = \underbrace{\begin{bmatrix} a_1 & 1 & 0 \\ a_2 & 0 & 1 \\ a_3 & 0 & 0 \end{bmatrix}}_{A_o = A_c^\intercal} \boldsymbol{\xi}_k + \underbrace{\begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}}_{B_o = C_c^\intercal} u_k,$$

$$y_k = \underbrace{\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}}_{C_o = B_c^\intercal} \boldsymbol{\xi}_k.$$

In this form, $\boldsymbol{\xi}_k \in \mathbb{R}^3$ represents the new state vector and the new state matrices are A_o, B_o, C_o. This new representation is called the **observable canonical form** and is particularly useful when reconstructing the hidden states from the output y_k.

2.3 Parameter Identification

Linear State-Space Models (LSSMs) can be seen as a special case of Recurrent Neural Networks (RNNs), with the key difference being that LSSMs are limited to linear transformations and assume Gaussian noise. These constraints make LSSMs simpler and more interpretable compared to the more general, nonlinear RNN architectures. Our interest in LSSM identification arises from this relationship to RNNs: whereas RNNs typically rely on non-linear activation functions—complicating and

often obscuring the learning process—LSSMs provide a transparent, mathematically tractable alternative. In this section, we approach the identification of LSSMs within the broader context of RNN learning, utilizing similar optimization techniques, but benefiting from the clarity that comes with the linear structure.

While classical methods for identifying the parameters of an LSSM are well-established and efficient [1, 2, 4], we introduce a modern optimization framework that parallels methods commonly used for RNNs. For simplicity, we focus on the noiseless LSSM case, where both process and measurement noise are absent. This assumption leads to a more straightforward estimation problem, though the framework outlined here serves as a basis for more advanced techniques that account for noise.

The goal is to estimate the LSSM's system matrices (A, B, C, D) along with the initial condition $\mathbf{x}_0$, collectively forming the parameter set $\boldsymbol{\theta} = (A, B, C, D, \mathbf{x}_0)$. These parameters will be estimated using methodologies commonly applied in the training of RNNs. The available data consist of an input sequence $\mathbf{u}_{0:L} = (\mathbf{u}_0, \dots, \mathbf{u}_L)$ and its corresponding output sequence $\mathbf{y}_{0:L} = (\mathbf{y}_0, \dots, \mathbf{y}_L)$. The identification process involves a gradient descent iterative algorithm, which is outlined in the following steps:

1. **Initialization:** The first step in the identification process is the initialization of the system matrices A, B, C, D and the initial state $\mathbf{x}_0$. In practice, initialization is often done using small random values, or by applying domain-specific heuristics. A judicious choice of initial values can significantly improve the convergence speed of the optimization process. For instance, initializing the matrix A with eigenvalues inside the unit circle ensures the stability of the state evolution, which is particularly important when propagating the hidden states over long time horizons.

2. **Loss Function:** In the noiseless case, the identification problem is formulated as minimizing the prediction error between the sequence of observed outputs $(\mathbf{y}_0, \dots, \mathbf{y}_L)$ and the predicted outputs $(\widehat{\mathbf{y}}_0, \dots, \widehat{\mathbf{y}}_L)$. The predicted output at time step k is obtained from the recursive state-space equations:

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k, \quad \widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k, \tag{10}$$

   where A, B, C, D and $\mathbf{x}_0$ are the current estimates of the LSSM's parameters. Note that these parameters are iteratively refined during the optimization process. The prediction error at each time step is evaluated using a loss function ℓ, which, in this case, is chosen to be the squared ℓ_2-norm[4]. Hence, the error at time step k is given by:

$$\ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k) = \|\mathbf{y}_k - \widehat{\mathbf{y}}_k\|_2^2.$$

[4]The squared ℓ_2-norm of a vector $\mathbf{z}$ is defined as $\|\mathbf{z}\|_2^2 = \mathbf{z}^\intercal\mathbf{z} = \sum_i z_i^2$.

The **total loss** over a time horizon of length L is defined as the average loss across all time steps:

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

where $\boldsymbol{\theta} = (A, B, C, D, \mathbf{x}_0)$ denotes the parameters of the system. This loss function serves as a measure of the alignment between the model's predicted outputs and the observed data. Minimizing the total loss thus provides an estimate of the model parameters that best explain the observed system behavior.

3. **Identification Using Backpropagation Through Time (BPTT):** The identification of the LSSM parameters is framed as an optimization problem, where the goal is to find the set of parameters $\boldsymbol{\theta}$ that minimizes the total loss function $\mathcal{L}$. The **Backpropagation Through Time (BPTT)** method, commonly used in RNN training, can be adapted for LSSMs. BPTT is a gradient-based method that iteratively refines the parameter values until they converge towards a local minimum.

   The challenge in BPTT lies in computing the gradients of the loss function, also known as error gradients, with respect to the set of parameters across multiple time steps. These **error gradients** represent how sensitive the loss function $\mathcal{L}$ is to changes in the system parameters. Specifically, the error gradient is the derivative of the loss with respect to the parameters A, B, C, D, and $\mathbf{x}_0$. It quantifies how much the prediction error (the difference between the predicted and observed outputs) changes as each parameter is adjusted. In a time-series context, the error at each time step depends not only on the current state but also on the states at all previous time steps, due to the recursive nature of the state equation. This temporal dependency means that the gradient of the loss function at each time step involves a chain of derivatives that trace back through previous time steps.

In the following subsection, we present the computational details behind BPTT.

2.4 BPTT Algorithm ⚠

Our coverage of the BPTT algorithm heavily relies on tensor algebra (see Appendix 2.7 for a brief review of tensor notation). Backpropagation Through Time (BPTT) is a key method used to compute gradients over sequential data by unrolling the temporal dependencies of the model. It is important to observe that the definition of the total loss function, together with the state and output equations, naturally gives rise to the following **computational graph**:

$$\boxed{\mathcal{L}(\boldsymbol{\theta}; \mathbf{y}_{0:L})} \leftarrow \boxed{\ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)} \leftarrow \boxed{\widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k} \leftarrow \boxed{\mathbf{x}_k = A\mathbf{x}_{k-1} + B\mathbf{u}_{k-1}} \leftarrow \cdots$$

$$\cdots \boxed{\leftarrow \mathbf{x}_2 = A\mathbf{x}_1 + B\mathbf{u}_1} \leftarrow \boxed{\mathbf{x}_1 = A\mathbf{x}_0 + B\mathbf{u}_0}.$$

The arrows in this computational graph indicate the directions of the computational dependencies. For clarity, the given data (i.e., input and output signals, $\mathbf{u}_k$ and $\mathbf{y}_k$) are colored in blue, while the elements to be identified (i.e., the parameters in $\boldsymbol{\theta}$, A, B, C, D, and $\mathbf{x}_0$) are colored in red. These dependencies form a **directed chain**, a characteristic feature of recurrent models such as RNNs.

In the following, we interpret $\boldsymbol{\theta}$ as a 3-dimensional tensor, constructed by **stacking** the matrices $A, B, C, D, \mathbf{x}_0$ and **padding** the smaller matrices with zeros to ensure uniform dimensions across all matrices:

$$\boldsymbol{\theta}_{1,j_1j_2} = a^{\text{pad}}_{j_1j_2}, \quad \boldsymbol{\theta}_{2,j_1j_2} = b^{\text{pad}}_{j_1j_2}, \quad \boldsymbol{\theta}_{3,j_1j_2} = c^{\text{pad}}_{j_1j_2},$$

$$\boldsymbol{\theta}_{4,j_1j_2} = d^{\text{pad}}_{j_1j_2}, \quad \text{and} \quad \boldsymbol{\theta}_{5,j_1j_2} = u^{\text{pad}}_{j_1j_2},$$

where the superscript `pad` indicates that we are referring to the padded version of the matrix. (Further details on the implementation of stacking and padding steps using `PyTorch` will be provided in a subsequent Python Lab.)

As a result, the gradient computation proceeds through tensor operations, as outlined below:

1. **Forward Pass (Prediction Update):** Using the current estimate of the parameters $A, B, C, D, \mathbf{x}_0$, we compute the time-series predictions $\widehat{\mathbf{y}}_k$. This is referred to as the **forward pass**, during which the model propagates the input sequence $\mathbf{u}_{0:L}$ through the state-space equations to generate new updated predictions. Specifically, the predictions are computed recursively as follows:

   $$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k, \quad \widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k,$$

   for $k = 0, 1, \ldots, L$, starting from the updated initial state $\mathbf{x}_0$.

2. **Backward Pass (Gradient Computation):** In this step, we employ the current values of the parameters $A, B, C, D, \mathbf{x}_0$ and the sequence of predictions $(\widehat{\mathbf{y}}_0, \ldots, \widehat{\mathbf{y}}_L)$ from the forward pass to compute the gradient of the total loss function with respect to the parameter tensor $\boldsymbol{\theta}$:

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

   Since the total loss function $\mathcal{L}$ is scalar-valued, its gradient with respect to $\boldsymbol{\theta}$ is a tensor of the same dimensions as $\boldsymbol{\theta}$, i.e., a 3-dimensional tensor. By utilizing

the computational graph, we can apply the chain rule to express the gradient tensor as follows:

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta};\, \mathbf{y}_{0:L})}{\partial \boldsymbol{\theta}} = \frac{1}{L+1}\sum_{k=0}^{L} \frac{\partial \ell(\mathbf{y}_k,\, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} \frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} \frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}} \cdots \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} \frac{\partial \mathbf{x}_1}{\partial \boldsymbol{\theta}}. \tag{11}$$

Next, we dissect each term in this chain of products.

Firstly, the **gradient** of the squared error loss function is given by:

$$\frac{\partial \ell(\mathbf{y}_k,\, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} = \frac{\partial \left\|\mathbf{y}_k - \widehat{\mathbf{y}}_k\right\|_2^2}{\partial \widehat{\mathbf{y}}_k} = 2(\widehat{\mathbf{y}}_k - \mathbf{y}_k)^\intercal. \tag{12}$$

We also have the following **Jacobian matrices**:

$$\frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} = C \quad \text{and} \quad \frac{\partial \mathbf{x}_{j+1}}{\partial \mathbf{x}_j} = A, \quad \text{for all } j = 1, \ldots, k-1. \tag{13}$$

The last term in equation (11), $\partial \mathbf{x}_1 / \partial \boldsymbol{\theta}$, is a 4-dimensional **Jacobian tensor**, where one dimension indexes the components of $\mathbf{x}_1$ and the other three index the elements of $\boldsymbol{\theta}$. This tensor can be decomposed into the following 3-dimensional tensorial slices, whose elements are computed as:

$$\left[\frac{\partial \mathbf{x}_1}{\partial A}\right]_{i,j_1 j_2} = \frac{\partial x_{1i}}{\partial a_{j_1 j_2}}, \quad \left[\frac{\partial \mathbf{x}_1}{\partial B}\right]_{i,j_1 j_2} = \frac{\partial x_{1i}}{\partial b_{j_1 j_2}}, \quad \left[\frac{\partial \mathbf{x}_1}{\partial C}\right]_{i,j_1 j_2} = \frac{\partial x_{1i}}{\partial c_{j_1 j_2}},$$

$$\left[\frac{\partial \mathbf{x}_1}{\partial D}\right]_{i,j_1 j_2} = \frac{\partial x_{1i}}{\partial d_{j_1 j_2}}, \quad \text{and} \quad \left[\frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0}\right]_{i,j} = \frac{\partial x_{1i}}{\partial x_{0j}}.$$

As an illustrative example, consider the first tensorial slice, i.e., $\partial \mathbf{x}_1 / \partial A$. The elements of this 3-dimensional tensor satisfy:

$$\left[\frac{\partial \mathbf{x}_1}{\partial A}\right]_{i,j_1 j_2} = \frac{\partial \left[A\mathbf{x}_0 + B\mathbf{u}_0\right]_i}{\partial a_{j_1 j_2}} = \frac{\partial \left[A\mathbf{x}_0\right]_i}{\partial a_{j_1 j_2}} = \frac{\partial}{\partial a_{j_1 j_2}} \sum_j a_{ij} x_{0j},$$

where in the last equality we use the component-wise definition of matrix-vector multiplication. This last term can be written as:

$$\sum_j \frac{\partial a_{ij}}{\partial a_{j_1 j_2}} x_{0j} = \sum_j \delta_{i,j_1} \delta_{j,j_2} x_{0j} = \delta_{i,j_1} \sum_j \delta_{j,j_2} x_{0j} = \delta_{i,j_1} x_{0 j_2},$$

where $\delta_{i,j}$ is the **Kronecker delta** function. Thus, we have:

$$\left[\frac{\partial \mathbf{x}_1}{\partial A}\right]_{i,j_1 j_2} = \delta_{i,j_1} x_{0 j_2} \Longrightarrow \frac{\partial \mathbf{x}_1}{\partial A} = \textstyle\sum_{i,j_1,j_2} \delta_{i,j_1} x_{0 j_2} \mathbf{e}_i \otimes \mathbf{e}_{j_1} \otimes \mathbf{e}_{j_2}. \tag{14}$$

Using the algebraic properties of the Kronecker product, we can express the Jacobian tensor as:

$$\frac{\partial \mathbf{x}_1}{\partial A} = \left(\sum_{i,j_1} \mathbf{e}_i \otimes \mathbf{e}_{j_1}\right) \otimes \left(\sum_{j_2} \delta_{i,j_1} x_{0j_2} \mathbf{e}_{j_2}\right),$$

which simplifies to:

$$\frac{\partial \mathbf{x}_1}{\partial A} = (\sum_i \mathbf{e}_i \otimes \mathbf{e}_i) \otimes \left(\sum_{j_2} \mathbf{e}_{j_2} x_{0j_2}\right) = \sum_i \mathbf{e}_i \otimes \mathbf{e}_i \otimes \mathbf{x}_0,$$

where we have used the associativity of the Kronecker product and its compatibility with scalar multiplication.

Therefore, combining equations (12), (13), and the expression above, we obtain the following from equation (11):

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta};\, \mathbf{y}_{0:L})}{\partial A} = \frac{1}{L+1} \sum_{k=0}^{L} \underbrace{2(\widehat{\mathbf{y}}_k - \mathbf{y}_k)^\intercal}_{\frac{\partial \ell}{\partial \widehat{\mathbf{y}}_k}} \cdot \underbrace{C}_{\frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k}} \cdot \underbrace{A^{k-1}}_{\prod_{j=1}^{k-1} \frac{\partial \mathbf{x}_{j+1}}{\partial \mathbf{x}_j}} \cdot \underbrace{(\sum_i \mathbf{e}_i \otimes \mathbf{e}_i \otimes \mathbf{x}_0)}_{\frac{\partial \mathbf{x}_1}{\partial A}}.$$

Observe that the product $2(\widehat{\mathbf{y}}_k - \mathbf{y}_k)^\intercal C A^{k-1}$ is a row vector (a 1-dimensional tensor), and $\sum_i \mathbf{e}_i \otimes \mathbf{e}_i \otimes \mathbf{x}_0$ is a 3-dimensional tensor; thus, their contraction yields a 2-dimensional matrix. This 2-dimensional matrix can be written explicitly by defining the row vector:

$$\mathbf{v}_k^\intercal = 2(\widehat{\mathbf{y}}_k - \mathbf{y}_k)^\intercal C A^{k-1}.$$

Using this vector and equation (14), we can rewrite the gradient with respect to A as:

$$\left(\frac{\partial \mathcal{L}(\boldsymbol{\theta};\, \mathbf{y}_{0:L})}{\partial A}\right)_{j_1 j_2} = \frac{1}{L+1} \sum_{k=0}^{L} \sum_i v_{ki} \delta_{i,j_1} x_{0j_2} = \frac{1}{L+1} \sum_{k=0}^{L} v_{kj_1} x_{0j_2}.$$

Thus, since $v_{kj_1} x_{0j_2} = [\mathbf{v}_k \mathbf{x}_0^\intercal]_{j_1 j_2}$, we can express the tensor gradient as:

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta};\, \mathbf{y}_{0:L})}{\partial A} = \frac{1}{L+1} \sum_{k=0}^{L} \mathbf{v}_k \mathbf{x}_0^\intercal = \frac{2}{L+1} \sum_{k=0}^{L} \left(A^{k-1}\right)^\intercal C^\intercal (\widehat{\mathbf{y}}_k - \mathbf{y}_k) \mathbf{x}_0^\intercal.$$

This provides the final closed-form expression for the first two-dimensional tensorial slice of the gradient $\partial \mathcal{L}(\boldsymbol{\theta};\, \mathbf{y}_{0:L})/\partial \boldsymbol{\theta}$. This expression involves the unknown parameters in $\boldsymbol{\theta}$, the observed output sequence $\mathbf{y}_{0:L}$, and the current sequence of estimates $\widehat{\mathbf{y}}_{0:L}$, which are computed from the given input-output sequences and the current estimates of the parameters in $\boldsymbol{\theta}$. Similar derivations can be performed for the remaining tensorial slices corresponding to B, C, D, and $\mathbf{x}_0$.

3. **Parameter Updates:** Once the gradients of the loss function with respect to the parameters $A, B, C, D, \mathbf{x}_0$ have been computed, the parameters are iteratively updated using a gradient-based optimization algorithm[5]. The parameter updates follow the rule:

$$A \leftarrow A - \eta \frac{\partial \mathcal{L}}{\partial A}, \quad B \leftarrow B - \eta \frac{\partial \mathcal{L}}{\partial B},$$

$$C \leftarrow C - \eta \frac{\partial \mathcal{L}}{\partial C}, \quad D \leftarrow D - \eta \frac{\partial \mathcal{L}}{\partial D}, \quad \mathbf{x}_0 \leftarrow \mathbf{x}_0 - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{x}_0},$$

where η is the learning rate, which controls the step size in the optimization process. This update set of parameters will then be used in the next iteration of the forward pass.

We iteratively repeat the forward and backward passes until the loss $\mathcal{L}(\boldsymbol{\theta}; \mathbf{y}_{1:L})$ converges to a satisfactory value, indicating that the tensor of model parameters $\boldsymbol{\theta}$ has been successfully identified. This same algorithm is used for identifying the parameters in RNNs, with additional complexity due to the nonlinear mappings involved in the recursion (see Chapter ? for details).

Python Lab: LSSM Identification Using BPTT

In this section, we demonstrate how to identify a Linear State-Space Model (LSSM) that can replicate the dynamics of an AR(2) process using Backpropagation Through Time (BPTT). The AR(2) process in this case is driven by deterministic input signals. Our goal is to model the system using an LSSM framework, iteratively refining the parameters to match the observed time-series data. The code is presented step-by-step, and each segment is explained thoroughly, with special emphasis on the PyTorch methods employed, as some readers may not be familiar with this library.

We begin by generating the input signals and AR(2) process data and proceed to implement the BPTT algorithm to learn the system's parameters. Afterward, we compare the estimated output from the identified model with the original system output. Each block of code is described below.

1. **Generating deterministic input signals and AR(2) process data**: We start by generating a deterministic input sequence using a combination of three sinusoidal functions with randomly selected, incommensurate frequencies. These signals will serve as the input for the AR(2) process. The AR(2) process is then simulated using known parameters.

[5]In practice, extensions of gradient descent, such as stochastic gradient descent (SGD) and its variants, are often used.

```python
import numpy as np
import torch

# Step 1: Generate input signals
L = 500  # Length of the time series
frequencies = np.random.uniform(0, 10, 3)  # Three
    incommensurate frequencies

u_k = np.sin(2 * np.pi * frequencies[0] * np.arange(L)) +
      np.sin(2 * np.pi * frequencies[1] * np.arange(L)) +
      np.sin(2 * np.pi * frequencies[2] * np.arange(L))

# AR(2) true parameters
phi1_true = 0.6
phi2_true = -0.2
B_true = 0.5

# Function to generate noiseless AR(2) process with
    deterministic input
def generate_noiseless_ar2_with_input(phi1,phi2,B,u_k, L):
    y = np.zeros(L)
    for k in range(2, L):
        y[k] = phi1*y[k-1] + phi2*y[k-2] + B*u_k[k-1]
    return y

# Generate the output of the AR(2) process
y_noiseless = generate_noiseless_ar2_with_input(phi1_true,
     phi2_true, B_true, u_k, L)
```

The input signal, u_k, is constructed by combining three sinusoidal functions with random, incommensurate frequencies. The function `generate_noiseless_ar2_with_input` simulates the AR(2) process based on the true parameters ϕ_1, ϕ_2, and B, and generates the noiseless output y.

2. **Defining the LSSM model in PyTorch**: Next, we define an LSSM using PyTorch[6]. The model is parameterized by the system matrices A, B, C, D, and the initial hidden state $\mathbf{x}_0$, which we aim to learn.

```python
# Step 2: Define the LSSM model class
class LSSM(torch.nn.Module):
    def __init__(self):
        super(LSSM, self).__init__()
        # Define system matrices of appropriate dimensions
```

[6]PyTorch is an open-source machine learning library primarily developed by Facebook's AI Research lab (FAIR). It provides flexible tools for building deep learning models, with a focus on automatic differentiation and GPU acceleration, making it suitable for research in neural networks and machine learning. PyTorch's dynamic computational graph system, called Autograd, is particularly useful for tasks like Backpropagation Through Time (BPTT), as it automatically computes the gradients required for optimizing model parameters. More details in Appendix ?

```
        self.A = torch.nn.Parameter(torch.randn(2, 2))
        self.B = torch.nn.Parameter(torch.randn(2, 1))
        self.C = torch.nn.Parameter(torch.randn(1, 2))
        self.D = torch.nn.Parameter(torch.randn(1, 1))
        # Define the vector of initial conditions
        self.x_0 = torch.nn.Parameter(torch.zeros(2))

    def forward(self, u_k, L):
        x_k = self.x_0
        y_pred = []
        for k in range(L):
            y_k = self.C @ x_k + self.D @ u_k[k]
            y_pred.append(y_k)
            x_k = self.A @ x_k + self.B @ u_k[k]
        return torch.stack(y_pred).squeeze()
```

The `LSSM` class defines the state-space model. The system matrices A, B, C, and D are initialized as trainable parameters, and the forward pass computes the predicted output sequence, iteratively updating the hidden states using the system dynamics.

3. **Training the model using BPTT**: This block implements the Backpropagation Through Time (BPTT) algorithm to update the model parameters based on the observed output and input sequences. We use stochastic gradient descent (SGD) for parameter updates.

```
# Step 3: Train the model using BPTT
def train_model(y_observed, u_k, epochs=5000,
    learning_rate=0.01):
    model = LSSM()
    optimizer = torch.optim.SGD(model.parameters(), lr=
        learning_rate)
    loss_fn = torch.nn.MSELoss()

    y_observed_tensor = torch.tensor(y_observed, dtype=
        torch.float32)
    u_k_tensor = torch.tensor(u_k, dtype=torch.float32)

    for epoch in range(epochs):
        optimizer.zero_grad()  # Zero gradients
        y_pred = model(u_k_tensor, L)  # Forward pass
        loss = loss_fn(y_pred, y_observed_tensor)  # Loss
            computation
        loss.backward()  # Backward pass (compute
            gradients)
        optimizer.step()  # Parameter update

        if epoch % 500 == 0:
            print(f'Epoch {epoch}, Loss: {loss.item()}')

    return model
```

```
21
22 # Train the model
23 model = train_model(y_noiseless, u_k)
```

The function `train_model` trains the LSSM by minimizing the mean squared error (MSE) between the predicted and observed outputs. The backward pass computes the gradients of the loss with respect to the parameters using PyTorch's **automatic differentiation**, and the optimizer updates the parameters accordingly.

4. **Comparing the model's predicted output with the observed output**: Finally, we compare the predicted output from the trained LSSM with the actual observed output to evaluate the model's performance.

```
1  # Step 4: Compare predicted vs actual output
2  import matplotlib.pyplot as plt
3
4  def compare_model_output(model, u_k, y_actual):
5      u_k_tensor = torch.tensor(u_k, dtype=torch.float32)
6      y_pred = model(u_k_tensor, L).detach().numpy()
7
8      plt.plot(y_actual, label='Actual Output')
9      plt.plot(y_pred, label='Predicted Output', linestyle='
           dashed')
10     plt.legend()
11     plt.show()
12
13 compare_model_output(model, u_k, y_noiseless)
```

In this final block, the function `compare_model_output` plots the actual and predicted outputs, allowing us to visually compare the two. The orange line represents the model's prediction, while the blue line represents the true output of the AR(2) process.

HERE!!!

Lab

Issue of exploding vanishing gradients... Easier to explain in LSSMs Gradient clipping for exploding gradients... adaptive learning rates to improve convergence...

CROSS-VALIDATION... HERE OR IN TRADITIONAL ML

HYPERPARAMETER SELECTION... HERE OR IN TRADITIONAL ML

PYTHON LAB...

REGULARIZATION IN THE LOSS FUNCTION (L2 AND SPARSITY REGULATIZATION) AND IN TRAINING (DROPOUT)... HERE OR IN TRADITIONAL ML

PYTHON LAB...

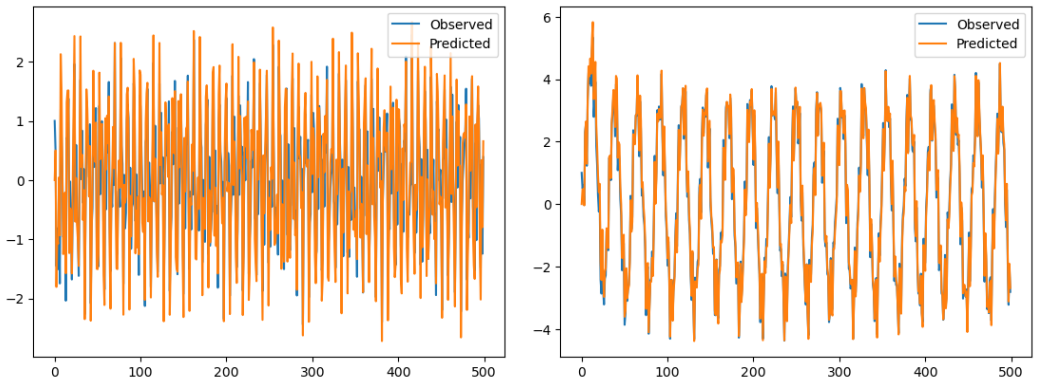

Figure 5: (Left) Comparison of the model predictions at early training stages with the actual output of the AR(2) model. (Right) Same comparison for a model in the later stages of training.

2.5 Order Reduction of an LSSM

Low-rank approximation of the state recursion...

2.6 LSSM with Feedback

??? LANGUAGE MODELS!!! CONTEXT WINDOW...

Together, the state and measurement equations provide a comprehensive framework that captures how the unobserved dynamics of the system evolve and how these dynamics are reflected in the observed data. This dual structure allows LSSMs to handle a wide variety of real-world scenarios, making them powerful tools for analyzing time series data where both the underlying system behavior and the measurement process are subject to uncertainty. By explicitly modeling these uncertainties, LSSMs offer robust forecasts and insights that account for both the hidden states and the observable data.

Linear State-Space Models (LSSMs) are highly flexible frameworks that can be enhanced by incorporating **feedback mechanisms**, where input signals are generated from the model's outputs using the following **linear feedback equation**:

$$\mathbf{u}_k = K \cdot \mathbf{X}_k,$$

where K is a **feedback matrix** of appropriate dimensions. This feedback creates a **closed-loop system** where the model dynamically adjusts its internal state using its own observed outputs, enabling continuous adaptation. By allowing inputs to be functions of the outputs, LSSMs can capture complex dynamics and enhance predictive performance. In future chapters, we will extend this concept to nonlinear feedback loops, which are capable of modeling even richer and more intricate

behaviors. For now, we focus on straightforward linear cases, as illustrated in the example below.

Example 7: Exponential Smoothing in LSSM Form with Feedback

Exponential smoothing is a forecasting technique that applies exponentially decreasing weights to past observations, allowing the model to prioritize recent data while gradually diminishing the influence of older data. This method is particularly effective in capturing trends and filtering out noise. Consider the exponential smoothing model of order s, defined as:

$$Y_{k+1} = \alpha Y_k + \alpha(1-\alpha)Y_{k-1} + \alpha(1-\alpha)^2 Y_{k-2} + \cdots + \alpha(1-\alpha)^s Y_{k-s},$$

where α is the smoothing parameter, with $0 < \alpha < 1$. This equation assigns exponentially decreasing weights to past values, making recent data more influential in the forecast.
To represent this exponential smoothing process as a Linear State-Space Model (LSSM), we define a scalar-valued state and output equations as follows:

1. **State Equation:** Define the scalar state variable X_k as the internal smoothed value at time k. The state equation describes the evolution of this state using the previous state, feedback from the output, and noise. The recursive relationship can be expressed as:

   $$X_{k+1} = (1-\alpha)X_k + U_k + W_k,$$

   where the scalar equality $U_k = Y_k$ feeds back the observed output Y_k into the input, ensuring that the latest observed value influences the state update.

2. **Output Equation:** The output equation defines how the scalar observed value Y_k is generated from the smoothed state. It is expressed as:

   $$Y_k = \alpha X_k,$$

   where X_k determines the observed output based on the current smoothed state scaled by the smoothing parameter α. This scaling reflects how much of the smoothed state is directly visible in the output.

The matrices defining the LSSM corresponding to the exponential smoothing process are the following 1×1 scalars:

$$A = (1-\alpha), \quad B = 1, \quad C = \alpha, \quad D = 0, \quad \text{and} \quad K = 1.$$

By including the feedback equation $U_k = KY_k$, the system uses the most recent observed value to adjust the internal smoothed state, creating a dynamic interplay between past and present information.

BAYESIAN LSSM

NONLINEAR SSM

EXAMPLE: LORENTZ EQUATION

CONTROLABILITY/OBSERVABILITY? BOOK BUT NO COURSE...

2.7 Koopman Operator

LSSM in infinite dimensions...

EM Algorithm: Technical Details

TBD

Appendix: Tensors

In this section, we cover computational aspects of tensors, leaving more abstract algebraic concepts such as subspaces to other references (see [3] for details).

A **d-dimensional tensor** $T \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is an array of values with d indices, where the set of integers $(n_1, \ldots, n_d)$ defines the **shape** of the tensor. The uppercase letter T denotes the tensor itself, while its entries are denoted by lowercase letters $t_{i_1 i_2 \cdots i_d}$, i.e., $[T]_{i_1 i_2 \cdots i_d} = t_{i_1 i_2 \cdots i_d}$. For instance, a vector is a 1-dimensional tensor, while a matrix is a 2-dimensional tensor.

An important example is the **delta (or identity) tensor**, defined componentwise as:

$$\delta_{i_1 i_2 \cdots i_d} = \begin{cases} 1 & \text{if } i_1 = i_2 = \cdots = i_d, \\ 0 & \text{otherwise.} \end{cases}$$

For example, the 2-dimensional identity tensor, also called the **Kronecker delta**, satisfies $\delta_{i,j} = 1$ when $i = j$; and 0 otherwise. This tensor defines the identity matrix as follows: $[\mathbb{I}_n]_{ij} = \delta_{i,j}$, for all $i, j \in [n]$.

In what follows, we use **multiindexes** (i.e., ordered sequences of indexes), such as $\mathcal{I}_p = (i_1, i_2, \ldots, i_p)$, $\mathcal{J}_q = (j_1, j_2, \ldots, j_q)$, and $\mathcal{K}_r = (k_1, k_2, \ldots, k_r)$, to index tensor elements. The concatenation of two multiindexes is another multiindex, denoted by:

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

where a *comma* is used to concatenate two index sequences. Using this notation, we index tensor elements as $[T]_{\mathcal{I}_d} = [T]_{i_1 i_2 \cdots i_d} = t_{i_1 i_2 \cdots i_d} = t_{\mathcal{I}_d}$. Furthermore, we can extend the definition of the delta tensor to sequence indexes, as follows:

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

List of Tensor Operations

- **Sum 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 as the element-wise as:

$$(\alpha\, T)_{\mathcal{I}_d} = \alpha\, t_{\mathcal{I}_d}.$$

- **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:

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

where $\mathcal{N}^d = [n_1] \times [n_2] \times \cdots \times [n_d]$ is the set of all possible multiindexes.

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

$$A \in \mathbb{R}^{n_1 \times \cdots \times n_p \times c_1 \times \cdots \times c_r} \text{ (dimension } p + r),$$

$$B \in \mathbb{R}^{c_1 \times \cdots \times c_r \times m_1 \times \cdots \times m_q} \text{ (dimension } q + r).$$

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

$$\begin{aligned}(A \cdot B)_{\mathcal{I}_p, \mathcal{J}_q} &= (A \cdot B)_{i_1 \ldots i_p, j_1 \ldots j_q} \\ &= \sum_{k_1 \ldots k_r} a_{i_1 \ldots i_p, k_1 \ldots k_r} b_{k_1 \ldots k_r, j_1 \ldots j_q} \\ &= \sum_{\mathcal{K}_r \in \mathcal{C}^r} a_{\mathcal{I}_p, \mathcal{K}_r} b_{\mathcal{K}_r, \mathcal{J}_q}.\end{aligned}$$

where the summation is performed over all r indices, i.e., $\mathcal{C}^r = [c_1] \times [c_2] \times \cdots \times [c_r]$. For example, the contraction product of two 2-dimensional tensors M and N (i.e., two matrices) can be written as:

$$(M \cdot N)_{i,j} = \sum_k m_{i,k} n_{k,j},$$

which is the standard matrix product.

- **Kronecker Product**: The Kronecker product of two tensors A and B of dimensions p and q, respectively, results in a tensor of dimension $p+q$, 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 matrices M and N is a 4-dimensional tensor defined entry-wise as $[M \otimes N]_{i_1 i_2, j_1 j_2} = m_{i_1 i_2} n_{j_1 j_2}$.

Among other algebraic properties, the Kronecker product is *associative*, i.e., $(A \otimes B) \otimes C = A \otimes (B \otimes C)$ and is compatible with the scalar multiplication, i.e., $(\alpha A) \otimes B = \alpha(A \otimes B) = A \otimes (\alpha B)$. Furthermore, 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. Let us denote by $\mathbf{e}_i^n$ 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. Hence, we can write:

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

where $\mathcal{I}_d = (i_1, \cdots, i_d)$ and $\mathbf{E}_{\mathcal{I}_d} = \mathbf{e}_{i_1}^{n_1} \otimes \cdots \otimes \mathbf{e}_{i_d}^{n_d}$, which is a d-dimensional tensor full of zeros except for a 1 in the element indexed by $\mathcal{I}_d$.

Jacobian Tensor

In many machine learning problems, it is necessary to compute the derivative of a tensor-valued function with respect to tensor arguments. For example, consider a function that maps a p-dimensional input tensor X to a q-dimensional output tensor, i.e., $F \colon \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_p} \to \mathbb{R}^{m_1 \times m_2 \times \cdots \times m_q}$. The derivatives of a tensor-valued function $F(X)$ with respect to its tensor argument X can be arranged into a new tensor of dimensions $p+q$, denoted as the **Jacobian tensor**, 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. For instance, consider the identity function of a 2-dimensional tensor X, i.e., $F(X) = X$ for $X \in \mathbb{R}^{n_1 \times n_2}$. The Jacobian is then the following 4-dimensional tensor:

$$\left[\frac{\partial X}{\partial X}\right]_{\mathcal{I}_2, \mathcal{J}_2} = \left[\frac{\partial X}{\partial X}\right]_{i_1 i_2, j_1 j_2} = \frac{\partial x_{i_1 i_2}}{\partial x_{j_1 j_2}} = \delta_{\mathcal{I}_2, \mathcal{J}_2}.$$

Properties of the Tensor Jacobian

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

- **Linearity**: The Jacobian of a linear combination of tensor 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*): For all tensor products defined above, the Jacobian of the product satisfies:

$$\begin{aligned}\frac{\partial}{\partial X}(F \cdot G) &= \frac{\partial F}{\partial X} \cdot G + F \cdot \frac{\partial G}{\partial X},\\ \frac{\partial}{\partial X}(F \otimes G) &= \frac{\partial F}{\partial X} \otimes G + F \otimes \frac{\partial G}{\partial X},\\ \frac{\partial}{\partial X}(F \odot G) &= \frac{\partial F}{\partial X} \odot G + F \odot \frac{\partial G}{\partial X},\end{aligned}$$

  where the order of the products is important for the *contraction* and *Kronecker* products, since they are not commutative.

- **Tensor Chain Rule**: Consider two tensor functions $F(Y)$ and $Y(X)$. Then, the chain rule for the tensor Jacobian is:

$$\frac{\partial F}{\partial X} = \frac{\partial F}{\partial Y} \cdot \frac{\partial Y}{\partial X},$$

  where the *contraction* product is taken over the indices of the tensor Y.

Example 8: Jacobian Tensor of Power Matrix

In blackboard, needs to be refined...

Consider a square matrix $X \in \mathbb{R}^{n\times n}$ and its power X^p for some positive integer p. We aim to compute the Jacobian of X^p with respect to X, denoted by $\frac{\partial X^p}{\partial X}$. The result is a 4th-order tensor that represents the Jacobian of the matrix function X^p with respect to X.

1. **Base Case:** $p = 1$**.**

   For $p = 1$, we simply have:

$$X^1 = X.$$

   The Jacobian of X with respect to itself is the 4th-order identity tensor $I^{\otimes 4}$, representing the Kronecker delta $\delta_{ij}\delta_{kl}$. Therefore,

$$\frac{\partial X}{\partial X} = I^{\otimes 4}.$$

2. **Case** $p = 2$**.**

   Now consider $p = 2$, where $X^2 = X \cdot X$. Using the product rule for matrix differentiation, we have:

$$\frac{\partial X^2}{\partial X} = \frac{\partial (X \cdot X)}{\partial X} = I^{\otimes 4} \cdot X + X \cdot I^{\otimes 4}.$$

Since $I^{\otimes 4} \cdot X = X$ and $X \cdot I^{\otimes 4} = X$, this simplifies to:

$$\frac{\partial X^2}{\partial X} = I^{\otimes 4} \otimes X + X \otimes I^{\otimes 4}.$$

Hence, for $p = 2$, we have the result:

$$\frac{\partial X^2}{\partial X} = X^0 \otimes X^1 + X^1 \otimes X^0.$$

3. **Case $p = 3$.**

   For $p = 3$, we have $X^3 = X \cdot X^2$. Applying the product rule again, we obtain:

$$\frac{\partial X^3}{\partial X} = \frac{\partial (X \cdot X^2)}{\partial X} = I^{\otimes 4} \cdot X^2 + X \cdot \frac{\partial X^2}{\partial X}.$$

   Substituting the result for $\frac{\partial X^2}{\partial X}$ from the previous step, we get:

$$\frac{\partial X^3}{\partial X} = X^2 + X \cdot (X^0 \otimes X^1 + X^1 \otimes X^0).$$

   Expanding this, we have:

$$\frac{\partial X^3}{\partial X} = X^2 \otimes I^{\otimes 4} + X^1 \otimes X^1 + X^0 \otimes X^2.$$

   This simplifies to:

$$\frac{\partial X^3}{\partial X} = X^0 \otimes X^2 + X^1 \otimes X^1 + X^2 \otimes X^0.$$

4. **Induction Step.**

   Now, we prove the general case by induction. Assume that for $p = n$, the result holds:

$$\frac{\partial X^n}{\partial X} = \sum_{k=0}^{n-1} X^k \otimes X^{n-1-k}.$$

   We will prove that the result holds for $p = n + 1$. Using the recursive product rule, we differentiate $X^{n+1} = X \cdot X^n$:

$$\frac{\partial X^{n+1}}{\partial X} = I^{\otimes 4} \cdot X^n + X \cdot \frac{\partial X^n}{\partial X}.$$

   Substituting the induction hypothesis for $\frac{\partial X^n}{\partial X}$, we have:

$$\frac{\partial X^{n+1}}{\partial X} = X^n + X \cdot \sum_{k=0}^{n-1} X^k \otimes X^{n-1-k}.$$

Expanding this, we obtain:

$$\frac{\partial X^{n+1}}{\partial X} = X^n \otimes I^{\otimes 4} + \sum_{k=0}^{n-1} X^{k+1} \otimes X^{n-1-k}.$$

Shifting the index in the summation, this simplifies to:

$$\frac{\partial X^{n+1}}{\partial X} = \sum_{k=0}^{n} X^k \otimes X^{n-k}.$$

Therefore, by induction, the general form of the Jacobian for X^p is:

$$\frac{\partial X^p}{\partial X} = \sum_{k=0}^{p-1} X^k \otimes X^{p-1-k}.$$

...

Exercises

References

[1] L. Ljung. System identification. In *Signal analysis and prediction*, pages 163–173. Springer, 1998.

[2] R. Pintelon and J. Schoukens. *System identification: a frequency domain approach*. John Wiley & Sons, 2012.

[3] J. G. Simmonds. *A Brief on Tensor Analysis*. Springer Science & Business Media, 2012.

[4] P. Van Overschee and B. De Moor. *Subspace Identification for Linear Systems: Theory—Implementation—Applications*. Springer Science & Business Media, 2012.

