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

Change to pendulum... exercise with double pendula... Lagrangian and Hamiltonian dynamics... Identification of the Lagrangian or the Hamiltonian...

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):

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

  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).

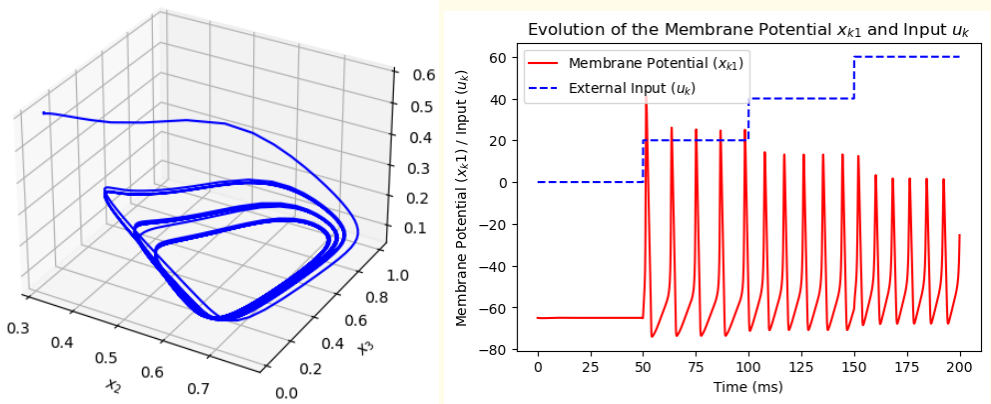

Figure 1: (Left) 3D plot for the joint evolution of gating variables, x_{k2}, x_{k3}, and x_{k4}. (Right) Evolution of the membrane potential x_{k1} and the input signal u_k.

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,

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

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

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

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 $Y_k \in \mathcal{O} = \{o_1, o_2, \dots, o_E\}$ 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.

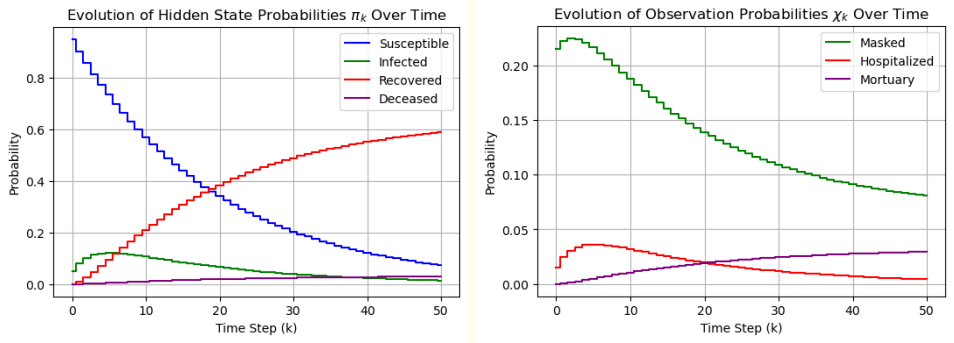

Figure 2: (Left) Evolution of the probabilities of being in each hidden state (e.g., Susceptible, Infected, Recovered, or Deceased). (Right) Evolution of the probabilities of observing a particular measurable behavior (e.g., Masked, Hospitalized, or Mortuary).

1.2.1 Parameter Estimation in Hidden Markov Models ⚠

EM ALGORITHM... Forward algorithm... Viterbi algorithm... example in fault detection...

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 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

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

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:

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

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

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

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

   where C^{AR} is the observation matrix, 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 $\mathbf{X}_k^{\text{MA}} \in \mathbb{R}^{q+1}$ is defined as:

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

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

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

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

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

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

   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-Invariance 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:

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

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:

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

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.

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:

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

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

[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.

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 [**?**, **?**, **?**], 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 $\mathbf{u}_{0:L} = (\mathbf{u}_0, \ldots, \mathbf{u}_L)$ and its corresponding output sequence $\mathbf{y}_{0:L} = (\mathbf{y}_0, \ldots, \mathbf{y}_L)$. 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 $(\mathbf{y}_0, \ldots, \mathbf{y}_L)$ and the predicted outputs $(\widehat{\mathbf{y}}_0, \ldots, \widehat{\mathbf{y}}_L)$. 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:

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

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

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

where $\boldsymbol{\theta} = (A, B, C, D, \mathbf{x}_0)$ denotes the parameters of the system. Notice that given these parameters and the input sequence $\mathbf{u}_{1:L}$, we can compute the sequence of output estimates $\mathbf{y}_{0:L}$; hence, we express $\mathbf{y}_k$ as $\mathbf{y}_k(\boldsymbol{\theta}; \mathbf{u}_{1:L})$. This total 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

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

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 B for a review of tensor notation, needed for this section). 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**:

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

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

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 tensor of order 3, constructed by **padding**[5] the matrices $A, B, C, D, \mathbf{x}_0$ with zeros to ensure uniform shapes across all dimensions and **stacking**[6] the padded matrices along a third dimension, i.e.,

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

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

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

[5] Padding is the process of adding zero entries to matrices or tensors to standardize dimensions, ensuring all matrices have the same shape for alignment in a single tensor.

[6] Stacking refers to the operation of aligning multiple matrices or tensors along a new dimension to create a higher-order tensor, in this case arranging matrices $A, B, C, D, \mathbf{x}_0$ along a third dimension after padding.

using `PyTorch` will be provided in a subsequent Python Lab, and Appendix E.) As a result, the gradient computation proceeds through tensor operations, as outlined below

2.4.1 Forward Pass (Prediction Update)

Using the current estimate of the parameters $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{p\times n}$, $D \in \mathbb{R}^{p\times m}$, and $\mathbf{x}_0 \in \mathbb{R}^n$, we compute the time-series predictions $\widehat{\mathbf{y}}_k \in \mathbb{R}^p$. This is referred to as the **forward pass**, during which the model propagates the input sequence $\mathbf{u}_{0:L} \in \mathbb{R}^m$ through the state-space equations to generate new updated predictions. Specifically, the predictions are computed recursively as follows:

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k, \quad \widehat{\mathbf{y}}_k = C\mathbf{x}_k + D\mathbf{u}_k, \tag{11}$$

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

2.4.2 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 $\boldsymbol{\theta}$:

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

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 tensor of order 3. By utilizing the computational graph, we can apply the chain rule to express the gradient tensor as the following sum of contraction products:

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta}; \mathbf{u}_{0:L}, \mathbf{y}_{0:L})}{\partial \boldsymbol{\theta}} = \frac{1}{L+1} \sum_{k=0}^{L} \frac{\partial \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} : \frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} : \frac{\partial \mathbf{x}_k}{\partial \boldsymbol{\theta}}, \tag{12}$$

where p and n are, respectively, the dimensions of the output and state vectors, $\widehat{\mathbf{y}}_k$ and $\mathbf{x}_k$. Next, we analyze each term in the chain of contraction products in (12).

The first term in the chain of products is the **gradient** of the loss function with respect to the predicted output. This gradient is a tensor of order 1 given by:

$$\frac{\partial \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} = \frac{\partial \left\|\mathbf{y}_k - \widehat{\mathbf{y}}_k\right\|_2^2}{\partial \widehat{\mathbf{y}}_k} = 2(\widehat{\mathbf{y}}_k - \mathbf{y}_k)^\intercal \in \mathbb{R}^p. \tag{13}$$

where p is the number of outputs of the LSSM (i.e., the dimension of $\widehat{\mathbf{y}}_y$). We also have the following **Jacobian tensor** of order 2 for the second term in the chain of products in (12):

$$\frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} = C \in \mathbb{R}^{p\times n}, \tag{14}$$

where n is the number of hidden states (i.e., the dimension of $\mathbf{x}_k$).

The last factor in equation (12), $\partial \mathbf{x}_k / \partial \boldsymbol{\theta}$, is a **Jacobian tensor** of order 4, where one dimension indexes the components of $\mathbf{x}_k$ and the other three index the elements of $\boldsymbol{\theta}$. This tensor can be decomposed into several tensorial slices of order 3. For example, the elements of the first tensorial slice of order 3 correspond to:

$$\left[\frac{\partial \mathbf{x}_k}{\partial \boldsymbol{\theta}}\right]_{1,i,j_1 j_2} = \left[\frac{\partial \mathbf{x}_k}{\partial A}\right]_{i,j_1 j_2} = \frac{\partial [\mathbf{x}_k]_i}{\partial a_{j_1 j_2}}.$$

As an illustrative example, we derive an explicit expression for the first tensorial slice, i.e., $\partial \mathbf{x}_k / \partial A$, using a recursive approach.

In order to find an appropriate recursion, we differentiate both sides of the state equation (11) with respect to A, as follows (proof left as an exercise):

$$\frac{\partial \mathbf{x}_{k+1}}{\partial A} = \frac{\partial (A\mathbf{x}_k)}{\partial A} + \frac{\partial (B\mathbf{u}_k)}{\partial A} = \mathbb{I}^{(2)} \otimes \mathbf{x}_k + A \colon \frac{\partial \mathbf{x}_k}{\partial A}, \tag{15}$$

where $\mathbb{I}^{(2)}$ is the identity tensor of order 2 (i.e., the identity matrix). Notice that B and $\mathbf{u}_k$ do not depend on A; hence, the Jacobian of $B\mathbf{u}_k$ disappears. Equation (15) provides us with a recursion to compute $\partial \mathbf{x}_{k+1} / \partial A$ from $\partial \mathbf{x}_k / \partial A$. We can iteratively solve this recursion starting with the initial condition:

$$\frac{\partial \mathbf{x}_0}{\partial A} = \mathbb{O}^{(3)} \in \mathbb{R}^{n \times n \times n},$$

where $\mathbb{O}^{(3)}$ is a tensor of order 3 filled with zeros. As a result of solving this recursion, we obtain the sequence of 3-dimensional Jacobian tensors $(\partial \mathbf{x}_{k+1} / \partial A)_{k=0}^{L-1}$.

This sequence of Jacobian tensors can then be substituted in the first tensorial slice of the expression in (12), i.e.,

$$\begin{aligned}
\frac{\partial \mathcal{L}(\boldsymbol{\theta};\, \mathbf{y}_{0:L})}{\partial A} &= \frac{1}{L+1} \sum_{k=0}^{L} \frac{\partial \ell(\mathbf{y}_k,\, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} \colon \frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} \colon \frac{\partial \mathbf{x}_k}{\partial A} \\
&= \frac{1}{L+1} \sum_{k=0}^{L} 2(\widehat{\mathbf{y}}_k - \mathbf{y}_k) \colon C \colon \frac{\partial \mathbf{x}_k}{\partial A}, \\
&= \frac{2}{L+1} \sum_{k=0}^{L} \left(\mathbf{r}_k^\intercal C\right) \colon \frac{\partial \mathbf{x}_k}{\partial A},
\end{aligned}$$

where we have used (13) and (14) in the second equality; in the third equality, we defined the residual $\mathbf{r}_k = \widehat{\mathbf{y}}_k - \mathbf{y}_k$ and use the identity $\mathbf{r}_k \colon C = \mathbf{r}_k^\intercal C$.

Therefore, using the recursive technique described above, we are able to compute the first tensorial slice of $\partial \mathcal{L} / \partial \boldsymbol{\theta}$. This technique can be trivially adapted to compute the rest of tensorial slices of $\partial \mathcal{L} / \partial \boldsymbol{\theta}$. This approach simplifies the gradient

computation and is scalable to systems with large time horizons and is almost identical to how BPTT is used to train RNNs. As we will see in the coming chapter, the main difference between the BPTT algorithm for LSSMs and RNNs is the inclusion of an extra element in the chain rule to account for nonlinear transformations in the computational chain.

2.4.3 Parameter Updates

After computing the tensorial slices of $\partial\mathcal{L}/\partial\boldsymbol{\theta}$, the parameters in $\boldsymbol{\theta}$ are iteratively updated using a gradient-based optimization algorithm[7]. Using a vanilla gradient descent, the parameter updates would be applied to the whole tensor $\boldsymbol{\theta}$ or order 3, as follows:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}),$$

where η is the **learning rate**, controlling the step size in the optimization process. This updated parameter set is then used in the subsequent forward pass iteration.

In order to find a set of parameters $\boldsymbol{\theta}$ that (locally) minimizes the total loss function $\mathcal{L}$, we iteratively repeat the forward and backward passes until convergence to a satisfactory value. It is important to note that this algorithm can be straightforwardly adapted to identify the parameters of 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 the parameters of an LSSM with 7 hidden states with the aim of replicating the dynamics of a noiseless AR(5) process using Backpropagation Through Time (BPTT). The AR(5) process is driven by a deterministic input signal, as follows:

$$Y_k = \phi_1 Y_{k-1} + \phi_2 Y_{k-2} + \cdots + \phi_5 Y_{k-5} + \beta u_k,$$

where u_k is a superposition of three sinusoidals, i.e.,

$$u_k = \sin(2\pi f_1 k) + \sin(2\pi f_2 k) + \sin(2\pi f_3 k),$$

where the frequencies f_1, f_2, and f_3 are chosen at random. Our goal is to iteratively refine the parameters of an LSSM of order 7 to match time-series generated by the AR(5) models. In practice, the order of the LSSM is a hyperparameter that should be chosen via cross-validation; however, in this example we focus on the training algorithm, leaving the cross-validation issue for future chapters. The code is implemented in `PyTorch` and available in GitHub. In the first block of code, we generate the input signals and implement the AR(5) process.

[7] In practice, variants of the vanilla gradient descent, such as *stochastic gradient descent* (SGD) or *adaptive moment estimation* (ADAM), are commonly employed.

1. **Generating the input signals and AR(5) process data**: We start by generating a deterministic input sequence using a combination of three sinusoidal functions with randomly selected frequencies. These signals will serve as the input for the AR(5) process. The AR(5) process is then simulated using known parameters.

```python
import numpy as np
import torch

# Step 1: Generate input signals
L = 500  # Length of the time series
# Select three random frequencies
frequencies = np.random.uniform(0, 10, 3)
# The input signal
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(5) true parameters
phi_true = [0.6, -0.2, 0.1, 0.05, -0.05]
beta_true = 0.5  # B multiplies the input signal

# Function to generate noiseless AR(5) process with
    deterministic input
def generate_ar5(phi, beta, u_k, L):
    y = np.zeros(L)
    for k in range(5, L):
        y[k] = sum(phi[j] * y[k - j - 1] for j in range(5)
            ) + beta * u_k[k - 1]
    return y

# Generate the output of the AR(5) process
y_AR5 = generate_ar5(phi_true, beta_true, u_k, L)
```

The scalar-valued input signal, u_k, is constructed by combining three sinusoidal functions with random frequencies. The function `generate_ar5` simulates the AR(5) process based on the true parameters ϕ and B, and generates the output `y_AR5`.

2. **Defining the LSSM model of order 7 in `PyTorch`**: Next, we define an LSSM using `PyTorch`[8]. 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):
```

[8] `PyTorch` is an open-source machine learning library that provides flexible tools for building deep learning models, with a focus on automatic differentiation and GPU acceleration. More details in Appendix E

```
        super(LSSM, self).__init__()
        # Define system matrices of appropriate dimensions
        self.A = torch.nn.Parameter(torch.randn(7, 7))
        self.B = torch.nn.Parameter(torch.randn(7, 1))
        self.C = torch.nn.Parameter(torch.randn(1, 7))
        self.D = torch.nn.Parameter(torch.zeros(1))
        # Define the vector of initial conditions
        self.x_0 = torch.nn.Parameter(torch.zeros(7))

    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` Python class defines the state-space model with 7 hidden states. The system matrices A, B, and C are initialized as trainable parameters using `torch.nn.Parameter`, while D is fixed as 0. This function creates a tensor of trainable parameters that `PyTorch`**'s autograd**[9] mechanism will automatically track, allowing these parameters to be updated during the optimization process.

During the forward pass, the predicted output sequence is computed iteratively using the current estimate of the parameters. The list of predicted outputs is converted into a tensor using `torch.stack(y_pred)`, which concatenates the elements along a new dimension. The `squeeze()` operation is then applied to remove any extra dimensions of size 1, resulting in a properly formatted tensor of predicted outputs.

3. **Training the model using BPTT**: The next 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()

```

[9] Autograd is `PyTorch`'s automatic differentiation engine that automatically computes the gradients of tensors with respect to specified parameters during backpropagation. More details in Appendix E.

```
    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
        # Forward pass
        y_pred = model(u_k_tensor, L)
        # Loss computation
        loss = loss_fn(y_pred, y_observed_tensor)
        # Backward pass (compute gradients)
        loss.backward()
        optimizer.step()  # Parameter update

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

    return model

# Train the model
model = train_model(y_AR5, u_k)
```

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

The `torch.optim.SGD` method is used to define the optimizer, specifically Stochastic Gradient Descent (SGD), which adjusts the model's parameters based on the gradients. The loss function used in this case is `torch.nn.MSELoss`, which computes the Mean Squared Error (MSE) between the predicted outputs and the actual observed data.

To ensure that the observed data can be used in `PyTorch`'s computation graph, the data is first converted into a `PyTorch` tensor using `torch.tensor`. This method transforms Python lists or NumPy arrays into `PyTorch` tensors, which support operations like automatic differentiation and tensor algebra, necessary for gradient-based optimization methods.

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.

```
# Step 4: Compare predicted vs actual output
import matplotlib.pyplot as plt

def compare_model_output(model, u_k, y_actual):
    u_k_tensor = torch.tensor(u_k, dtype=torch.float32)
    y_pred = model(u_k_tensor, L).detach().numpy()

```

```
    plt.plot(y_actual, label='Actual Output')
    plt.plot(y_pred, label='Predicted Output', linestyle='
        dashed')
    plt.legend()
    plt.show()

compare_model_output(model, u_k, y_AR5)
```

In this final block, the function `compare_model_output` plots the actual and predicted outputs, allowing us to visually compare the two. The output plots are included in the Fig. 5, where the orange line represents the model's prediction, while the blue line represents the true output of the AR(5) process. To convert the `PyTorch` tensors back into NumPy arrays for plotting, the method `.detach().numpy()` is used. The `.detach()` function ensures that the tensor is removed from the computation graph, preventing any further tracking of gradients, while `.numpy()` converts the tensor into a NumPy array for easy manipulation with libraries like `matplotlib`.

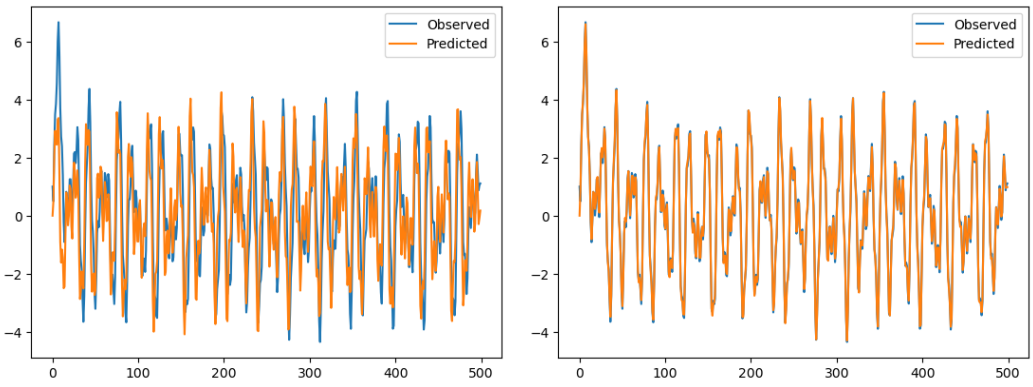

Figure 5: (Left) Comparison of the model predictions at early training stages with the actual output of the AR(5) model. (Right) Same comparison for a model in the later stages of training.

2.5 Vanishing and Exploding Gradients in LSSMs

When training recursive predictive architectures, such as LSSMs or RNNs, with gradient-based methods, a primary challenge that arises is the vanishing and exploding gradient problem. As we will demonstrate, this issue is intrinsically linked to the eigenvalues of the state matrix A , which dictates the system's dynamics.

In Appendix D, we show that the tensor gradient of the total loss with respect to the state matrix A is a complex tensorial expression. To gain clearer insight into the

behavior of the tensor gradient, let us consider an *autonomous* LSSM with a single hidden state. In this case, the set of parameters $\boldsymbol{\theta}$ simplifies to $(a, \mathbf{b}^\intercal, \mathbf{c}, D, x_0)$, where a is a scalar. In this case, the unrolled dynamics of the LSSM (7) and its Jacobian simplify into:

$$\widehat{\mathbf{y}}_k = \mathbf{c}a^k x_0 \Longrightarrow \frac{\partial \widehat{\mathbf{y}}_k}{\partial a} = \mathbf{c}ka^{k-1}x_0.$$

Hence, the gradient of the total loss satisfies:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial a} &= \frac{1}{L+1}\sum_{k=0}^{L}\frac{\partial \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)}{\partial a} \\ &= \frac{1}{L+1}\sum_{k=0}^{L}\frac{\partial \|\mathbf{y}_k - \widehat{\mathbf{y}}_k\|_2^2}{\partial \widehat{\mathbf{y}}_k}\frac{\partial \widehat{\mathbf{y}}_k}{\partial a} \\ &= \frac{1}{L+1}\sum_{k=0}^{L} 2\left(\mathbf{r}_k{:}\mathbf{c}\right) ka^{k-1}x_0, \end{aligned} \tag{16}$$

where $\mathbf{r}_k = (\widehat{\mathbf{y}}_k - \mathbf{y}_k)$ is the k-th residual. Since $\mathbf{r}_k{:}\mathbf{c} = \langle \mathbf{r}_k, \mathbf{c}\rangle$, (16) can be rewritten as the following weighted sum:

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{1}{L+1}\sum_{k=0}^{L}\gamma_k \cdot \langle \mathbf{r}_k, \mathbf{c}\rangle, \text{ with } \gamma_k = 2ka^{k-1}x_0. \tag{17}$$

Note that, depending on the value of a, the sequence of weights $(\gamma_k)_{k=0}^{L}$ exhibits two contrasting behaviors: When $a > 1$, the sequence of weights grows *superexponentially*, inducing an **exploding gradient** effect. Conversely, when $a < 1$, the sequence decays *subexponentially*, resulting in a **vanishing gradient** effect.

When the number of hidden states exceeds one, the state matrix A is no longer a scalar, and the role of the scalar a in our simplified analysis is effectively played by the eigenvalues of A. Denote the eigenvalues of A by $\lambda_1, \lambda_2, \ldots, \lambda_n$ and define the spectral radius of A as the largest modulus of its eigenvalues, i.e.,

$$\rho(A) = \max_i\{|\lambda_i|\}.$$

The spectral radius provides a measure of the growth rate of powers of A. Specifically, for any norm $\|\cdot\|$, the growth of $\|A^k\|$ as $k \to \infty$ is governed by $\rho(A)$. If $\rho(A) < 1$, the norm of A^k tends to zero as k increases, implying stability in discrete dynamical systems modeled by A. Conversely, if $\rho(A) > 1$, the norm of the powers A^k diverges, suggesting instability.

In order to extend the previous simplified analysis to the case where the number of hidden states is arbitrary., we make use of the following expression for the gradient

of the total loss in the case of an *autonomous* LSSM (see Appendix D for a full derivation):

$$\frac{\partial \mathcal{L}}{\partial A} = \frac{2}{L+1} \sum_{k=0}^{L} \sum_{q=1}^{k} \mathbf{r}_k : C : P^{\sigma} \left(A^{q-1} \otimes A^{k-q} \right) : \mathbf{x}_0,$$

where P^{σ} is a permutation operator. Furthermore, the spectral radius of the Kronecker product in the previous equation satisfies [**?**]:

$$\rho \left(A^{q-1} \otimes A^{k-q} \right) = \rho \left(A \right)^{k-1}.$$

Therefore, depending on the value of the spectral radius $\rho(A)$, we obtain two contrasting behaviors in the gradient of the total loss:

- **Exploding Gradients:** If the spectral radius of A is greater than 1, i.e., $\rho(A) > 1$, the norm of the gradient grows superexponentially as k increases. During backpropagation, this means that residuals $\mathbf{r}_k$ associated with larger k values are multiplied by superexponentially growing weights. This phenomenon leads to the **exploding gradient problem**, where gradients become excessively large, introducing instability in the training process. As a result, convergence becomes challenging, particularly for long time sequences.
- **Vanishing Gradients:** Conversely, if the spectral radius of A is less than 1, i.e., $\rho(A) < 1$, the norm of the gradient will decay subexponentially as k increases. This results from the residuals $\mathbf{r}_k$ in the summation being weighted by subexponentially shrinking coefficients, leading to the **vanishing gradient problem**. Here, gradients become too small to contribute meaningfully to parameter updates, significantly slowing or halting the learning process.
- **Stable Dynamics:** Ideally, the eigenvalues of A should be close to, but not exceed, the unit circle (i.e., $|\lambda_i| \leq 1$). This configuration promotes marginally stable dynamics, allowing the system to strike a balance where gradients neither explode nor vanish excessively, thus enabling stable and effective learning.

In summary, the eigenvalues of the state matrix A play a pivotal role in determining the behavior of the gradients during training, and thus, the overall stability of the learning process. In the following subsection, we discuss techniques commonly used to address the exploding and vanishing gradient problems.

2.5.1 Addressing the Gradient Problem

The challenges of vanishing and exploding gradients can significantly affect the training of LSSMs using gradient-based methods. When the gradients either shrink 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 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. This normalization prevents the gradient from growing too large and destabilizing the learning process.

  Gradient clipping is particularly effective in scenarios involving recurrent neural networks (RNNs) and other deep architectures, where long-term dependencies can cause gradients to accumulate to exceedingly large values. By capping the gradients, the model can continue training without the risk of numerical overflow or erratic updates that hinder convergence. This technique helps maintain a consistent learning trajectory, ensuring that gradient updates contribute meaningfully to parameter optimization and model performance.

- **Eigenvalue Regularization:** Another approach specifically targets the eigenvalues of the state matrix A, which are directly responsible for the long-term behavior of the system dynamics. 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 eigenvalues of A to remain within the unit circle. 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 compute the gradient $\partial \mathcal{L}_{\text{reg}}/\partial A$, we use the chain rule. For each eigenvalue λ_i, the derivative $\partial \lambda_i/\partial A$ is given by $\mathbf{u}_i \mathbf{v}_i^\intercal$, where $\mathbf{v}_i$ and $\mathbf{u}_i$ are the right and left eigenvectors of A, respectively, normalized such that $\mathbf{u}_i^\intercal \mathbf{v}_i = 1$. Therefore, the gradient of the regularization term becomes:

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

  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.

It is worth mentioning that similar techniques are also commonly employed in the training of Recurrent Neural Networks (RNNs) and related architectures, including Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs). Beyond gradient clipping and regularization, additional methods such as input normalization, batch normalization, and adaptive learning rate scheduling are frequently used to further stabilize training and improve model performance in these cases. We will expand on several of these techniques in the chapters that follow.

3 Further Topics in SSMs

TBD. Internal note.

3.1 Bayesian Estimation of Parameters in LSSM

TBD. Internal note.

3.2 Order Reduction of an LSSM ⚠

TBD. Internal note.

3.3 Controllability and Observability of LSSMs ⚠

TBD. Internal note.

3.4 Nonlinear SSMs

TBD. Internal note.

3.5 Koopman Operator ⚠

TBD. Internal note.

A EM Algorithm: Technical Details

TBD

B 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: $\left[\mathbb{I}^{(2)}\right]_{ij} = \delta_{ij}$.

B.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:

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

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:

$$\begin{aligned} [A::B]_{\mathcal{I}_p, \mathcal{J}_q} &= [A::B]_{i_1 \ldots i_p, j_1 \ldots j_q} \\ &= \sum_{s_1 \ldots s_r} a_{i_1 \ldots i_p, s_1 \ldots s_r} b_{s_1 \ldots s_r, j_1 \ldots j_q} \\ &= \sum_{\mathcal{S}_r} a_{\mathcal{I}_p, \mathcal{S}_r} b_{\mathcal{S}_r, \mathcal{J}_q}. \end{aligned} \tag{18}$$

When $r = 1$ (i.e., the summation is over a single index s_1), the symbol "::" is usually simplified to ":"; when $r > 1$ and is not clear from context, the symbol "::" is complemented as "$\overset{r}{::}$" For example, the contraction product of two tensors M and N of order 2 (i.e., two matrices) can be written as:

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

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[10] 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).

[10] 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 componentwise 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.

B.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 $\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) & f_2(\mathbf{x}) & \cdots & f_m(\mathbf{x}) \end{bmatrix}^\intercal$. 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.

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:
$$\frac{\partial}{\partial X}\left(\alpha F + \beta G\right) = \alpha\frac{\partial F}{\partial X} + \beta\frac{\partial G}{\partial X},$$
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 7: 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:

$$\left[\frac{\partial X}{\partial X}\right]_{\mathcal{I}_p,\mathcal{J}_p} = \left[\frac{\partial X}{\partial X}\right]_{i_1\dots i_p,j_1\dots j_p} = \frac{\partial x_{i_1\dots i_p}}{\partial x_{j_1\dots j_p}} = \delta_{\mathcal{I}_p\mathcal{J}_p} = \left[\mathbb{I}^{(2p)}\right]_{\mathcal{I}_p,\mathcal{J}_p},$$

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 8: 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 $\frac{\partial x_{ik}}{\partial x_{j_1j_2}} = \delta_{ij_1}\delta_{kj_2}$, 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:

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

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 $\mathbf{v}$.

Example 9: 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:

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

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:

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

We then take the partial derivative with respect to X:

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

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.

Example 10: 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

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

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:

$$\left[\frac{\partial X^p}{\partial X}\right]_{i_1i_2,j_1j_2} = \left[\frac{\partial \left(X \cdot X^{p-1}\right)}{\partial X}\right]_{i_1i_2,j_1j_2} = \frac{\partial \left[X \cdot X^{p-1}\right]_{i_1i_2}}{\partial x_{j_1j_2}}.$$

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 $p = 2$:

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

- For $p = 3$:

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

- 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:

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

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.

B.3 Tensor Norms

Tensor norms are generalizations of matrix norms used to measure the magnitude or size of tensors, playing a critical role in applications like data compression, optimization, and regularization in machine learning models. These norms are fundamental for assessing convergence, stability, and efficiency in high-dimensional data and tensor decompositions. Below, we discuss common tensor norms and their relevance in machine learning and data science.

ℓ_2-Norm (Frobenius Norm)

The Frobenius norm is commonly used for tensors due to its straightforward extension from matrices. For a tensor $T \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, it is defined as:

$$\|T\|_F^2 = \sum_{\mathcal{I}_d} |t_{\mathcal{I}_d}|^2 = \langle T, T \rangle,$$

representing the inner product of the tensor with itself. This norm is equivalent to the Euclidean norm of the vectorized tensor and measures the overall magnitude of its elements, making it widely used in optimization tasks and loss functions.

ℓ_1-Norm

The ℓ_1 norm sums the absolute values of all tensor entries. For $T \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, it is given by:

$$\|T\|_1 = \sum_{\mathcal{I}_d} |t_{\mathcal{I}_d}| .$$

This norm is especially useful in applications requiring sparsity-inducing regularization, promoting zero-valued elements in tensor decompositions or factor matrices.

ℓ_∞-Norm

The ℓ_∞ norm, also known as the max norm, identifies the maximum absolute value among the tensor's entries. For $T \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, it is defined as:

$$\|T\|_\infty = \max_{\mathcal{I}_d} |t_{\mathcal{I}_d}| .$$

This norm provides a measure of the largest value in the tensor and is useful for bounding and analyzing the behavior of tensor components in machine learning models.

Spectral Norm

The spectral (or operator) norm of a tensor is the maximum singular value when the tensor is considered as a multilinear map. It is defined by:

$$\|T\|_\sigma = \max_{\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(d)}} \frac{T \times_1 \mathbf{u}^{(1)} \times_2 \mathbf{u}^{(2)} \cdots \times_d \mathbf{u}^{(d)}}{\|\mathbf{u}^{(1)}\|_2 \|\mathbf{u}^{(2)}\|_2 \cdots \|\mathbf{u}^{(d)}\|_2},$$

where the numerator represents the Tucker product of tensor T with vectors $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(d)}$, yielding a scalar. The spectral norm is particularly important in stability analysis and growth rate evaluation, such as in analyzing the vanishing and exploding gradient problems in recurrent neural networks.

Other Tensor Norms

Additional norms, such as the nuclear norm and other specialized norms, are used in specific applications depending on the context. The nuclear norm is often applied in low-rank tensor completion tasks and approximations.

In summary, tensor norms are vital for managing tensor complexity in machine learning models. Selecting the right norm can affect the behavior of optimization methods, the effectiveness of regularization strategies, and the robustness of models handling multi-dimensional or multi-modal data.

B.4 Tensor Decompositions

Internal note.

C Gradient Computation of the Loss Function

In this section, we derive an explicit expression for the gradient of the total loss function $\mathcal{L}$ with respect to the state transition matrix A in an LSSM. We utilize the unrolled state equation to avoid recursive methods typically employed in Backpropagation Through Time (BPTT). As we demonstrate below, this expression can provide insights into gradient-related issues, such as vanishing and exploding gradients.

The unrolled output equation for the LSSM over k time steps is given by (7), which is repeated below for convenience:

$$\widehat{\mathbf{y}}_k = CA^k\mathbf{x}_0 + \sum_{i=1}^{k} CA^{i-1}B\mathbf{u}_{k-i} + D\mathbf{u}_k. \tag{19}$$

We define the total loss function $\mathcal{L}$ over a time horizon L using the average ℓ_2 loss:

$$\mathcal{L} = \frac{1}{L+1}\sum_{k=0}^{L} \|\mathbf{y}_k - \widehat{\mathbf{y}}_k\|_2^2,$$

where $\mathbf{y}_k \in \mathbb{R}^p$ is the true output at time k. Applying the chain rule, we have the following expression:

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

Since $\partial \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)/\partial \widehat{\mathbf{y}}_k = 2(\widehat{\mathbf{y}}_k - \mathbf{y}_k)$ for the ℓ_2 loss, we have that

$$\frac{\partial \mathcal{L}}{\partial A} = \frac{2}{L+1}\sum_{k=0}^{L} (\widehat{\mathbf{y}}_k - \mathbf{y}_k) : \frac{\partial \widehat{\mathbf{y}}_k}{\partial A}. \tag{20}$$

The next task is to compute $\partial \widehat{\mathbf{y}}_k/\partial A$ for each time step k. Note that $D\mathbf{u}_k$ does not depend on A, so the derivative of both sides of (19) with respect to A yields the following tensor of order 3 (see Example 9 for justification):

$$\frac{\partial \widehat{\mathbf{y}}_k}{\partial A} = C : \frac{\partial A^k}{\partial A} : \mathbf{x}_0 + \sum_{i=1}^{k} C : \frac{\partial A^{i-1}}{\partial A} : (B\mathbf{u}_{k-i}). \tag{21}$$

Using the expression for the Jacobian of the matrix power from Example 10, we obtain:

$$\frac{\partial \widehat{\mathbf{y}}_k}{\partial A} = \sum_{q=1}^{k} C : P^{\sigma}\left(A^{q-1} \otimes A^{k-q}\right) : \mathbf{x}_0 + \sum_{i=1}^{k}\sum_{q=1}^{i-1} C : P^{\sigma}\left(A^{q-1} \otimes A^{i-1-q}\right) : B\mathbf{u}_{k-i},$$

where σ permutes the sequence $(1, 4, 2, 3)$ into $(1, 2, 3, 4)$.

Substituting this expression into (20), we obtain an explicit form for the total loss gradient in tensor notation:

$$\frac{\partial \mathcal{L}}{\partial A} = \frac{2}{L+1} \sum_{k=0}^{L} (\widehat{\mathbf{y}}_k - \mathbf{y}_k) : \left(\sum_{q=1}^{k} C : P^{\sigma} \left(A^{q-1} \otimes A^{k-q} \right) : \mathbf{x}_0 + \sum_{i=1}^{k} \sum_{q=1}^{i-1} C : P^{\sigma} \left(A^{q-1} \otimes A^{i-1-q} \right) : B\mathbf{u}_{k-i} \right).$$

This expression for the tensor gradient $\partial \mathcal{L}/\partial A$ provides a direct method for computing the gradient without resorting to recursive techniques such as BPTT. Although this approach may be computationally intensive due to matrix powers and Kronecker products, it is valuable for analyzing how variations in A impact the overall loss $\mathcal{L}$.

D Introduction to `PyTorch`

`PyTorch` is an open-source deep learning framework developed by Facebook's AI Research lab that provides extensive capabilities for building, training, and deploying machine learning models. Unlike many traditional machine learning libraries, `PyTorch` is particularly optimized for deep learning applications, with built-in support for tensor operations, automatic differentiation, and GPU acceleration. In this section, we introduce the core concepts of `PyTorch`, with a focus on its tensor functionalities and autograd capabilities. We assume the reader is already familiar with Python programming, as well as basic principles of linear algebra and machine learning.

At the heart of `PyTorch` is the `Tensor` object, a generalization of matrices to arbitrary dimensions, designed to represent multi-dimensional data efficiently. `PyTorch` tensors, similar to NumPy arrays, are highly optimized for linear algebra operations. However, they extend beyond standard arrays by supporting operations on GPUs, which greatly accelerates computations for large datasets or complex models.

`PyTorch` offers a range of functions to create tensors, from manually specifying values to initializing with random values. Each tensor in `PyTorch` is defined by three main properties:

- **Shape**: The dimensions of the tensor. For example, a shape of $(3, 3)$ represents a 2D tensor (matrix) with 3 rows and 3 columns.
- **Data Type (dtype)**: The type of data stored in the tensor, such as `float32` or `int64`. Specifying the data type is essential for ensuring computational precision. The default data type for floating-point tensors is `float32`.
- **Device**: The hardware where the tensor is stored and processed, typically a CPU or GPU. Tensors default to the CPU, but can be explicitly moved to a GPU for accelerated computation.

Below are examples of tensor creation with illustrations of shape, data type, and device in specific cases:

```
import torch

# Creating a 1D tensor (vector) with specific values
tensor_a = torch.tensor([1.0, 2.0, 3.0])

# Creating a 2D tensor (matrix) of zeros with shape (3, 3)
tensor_b = torch.zeros((3, 3))

# Creating a 3D tensor with random values and shape (2, 4, 5)
tensor_c = torch.rand((2, 4, 5))

```

```
# Creating a 2x2x2 tensor with specific values in float64
tensor_d = torch.tensor([[[1.0, 2.0],
                          [3.0, 4.0]],

                         [[5.0, 6.0],
                          [7.0, 8.0]]], dtype=torch.float64)
```

D.1 Tensor Operations

Tensors support a variety of operations essential for building machine learning models, including basic arithmetic, matrix multiplication, reshaping, slicing, broadcasting, and more. PyTorch optimizes these operations to leverage parallel computing on CPUs or GPUs where available. Below, we explore common tensor manipulations in PyTorch.

Basic Arithmetic and Matrix Multiplication

Element-wise arithmetic and matrix multiplication are foundational tensor operations. PyTorch provides straightforward implementations for both.

```
# Element-wise addition
tensor_sum = tensor_a + tensor_b

# Hadamard (element-wise) product
hadamard_product = torch.mul(tensor_a, tensor_a)
```

In PyTorch, the function `torch.matmul` applies standard matrix multiplication rules to two-dimensional tensors (matrices). For higher-dimensional tensors, it performs **batch matrix multiplication**. Specifically, given two tensors $A \in \mathbb{R}^{b\times m\times n}$ and $B \in \mathbb{R}^{b\times n\times p}$, where b is the batch size, the resulting tensor $C = \text{matmul}(A, B)$ has shape $\mathbb{R}^{b\times m\times p}$, with entries given by:

$$C_{k,i,j} = \sum_{s=1}^{n} A_{k,i,s}\, B_{k,s,j},$$

where k indexes each batch, i and j are the row and column indices in the resulting matrix for each batch, and s represents the summation index.

Below is an example of matrix multiplication with `torch.matmul`:

```
# Batch matrix multiplication
product = torch.matmul(tensor_a, tensor_b)
```

Reshaping Tensors

Reshaping adjusts the dimensions of a tensor without altering its data, allowing compatibility with various model requirements. This operation is particularly useful in scenarios where tensors need to be aligned with specific layer requirements in neural networks. `PyTorch` provides the `view` function (or alternatively, `reshape`) to enable the creation of tensors with new shapes.

Reshaping changes only the dimensional structure of the tensor while preserving the total number of elements. For example, if a tensor has a shape of 2×4 (which includes $2 \times 4 = 8$ elements), it can be reshaped to 4×2 or 8×1, but not 3×3, since that shape would require 9 elements. The order of the elements is preserved during reshaping by using **lexicographical order**, also known as row-major order. In this order, elements are read in sequence across dimensions, which is especially important for keeping the element positions consistent when reshaping or flattening tensors in memory.

For instance, a 2×4 tensor, such as:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix},$$

is stored in memory in lexicographical order as $[1, 2, 3, 4, 5, 6, 7, 8]$. When reshaped, the data sequence remains the same, but the dimensions adjust to the specified new shape. For example, consider this code snippet:

```
# Creating a 2x4 tensor (8 elements)
tensor_a = torch.tensor([[1, 2, 3, 4],
                         [5, 6, 7, 8]])

# Reshaping it to 4x2
reshaped_tensor = tensor_a.view(4, 2)
```

This code results in the following 4×2 tensor:

$$\texttt{reshaped\_tensor} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

The elements are rearranged into the new shape 4×2, following lexicographical order to maintain the element positions within the tensor.

Reshaping operations are essential in preparing data for neural network layers, ensuring that tensors are compatible with specific dimensional requirements in machine learning models.

Broadcasting.

Broadcasting allows element-wise operations between two tensors of different shapes by automatically expanding the smaller tensor's dimensions to match those of the larger tensor. For broadcasting to be valid, all dimensions of both tensors must either match exactly, or one dimension must have a size of 1 in one of the tensors. This allows broadcasting to *stretch* the dimension of size 1 by repeating its elements to match the size of the corresponding dimension in the larger tensor. This feature eliminates the need for explicit reshaping or tiling, streamlining operations across tensors with mismatched shapes.

For example, consider two matrices $A \in \mathbb{R}^{2\times 1}$ and $B \in \mathbb{R}^{2\times 3}$:

$$A = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}.$$

Suppose we wish to add A and B, despite their differing shapes. Broadcasting virtually expands A along its second dimension to match the shape of B while leaving its original data intact. The addition proceeds as follows:

$$A + B = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix} = \begin{bmatrix} 11 & 21 & 31 \\ 42 & 52 & 62 \end{bmatrix}.$$

This mechanism allows for efficient computations without increasing memory usage, as the smaller tensor is conceptually expanded but not physically duplicated. The following code demonstrates broadcasting using two tensors initialized with random values. In this example, `tensor_a` is automatically *stretched* along its dimensions to match the shape of `tensor_b`, enabling element-wise addition without the need for explicit reshaping:

```
# Broadcasting example with random values
tensor_a = torch.rand(2, 1)
tensor_b = torch.rand(2, 3)
result = tensor_a + tensor_b
# tensor_a is "stretched" to match the shape of tensor_b
```

Slicing and Indexing.

Slicing and indexing enable efficient access to subtensors, allowing selection of specific elements or blocks within a tensor. `PyTorch` supports standard Python indexing notation for slicing. You can use slicing to access specific rows, columns, or even full dimensions of a tensor, depending on the indices specified.

```
# Slicing a tensor of order 2 to obtain a submatrix
submatrix = tensor_b[0:2, 1:3]
```


In this example, `tensor_b[0:2, 1:3]` selects rows 0 and 1 (from 0:2) and columns 1 and 2 (from 1:3). The result will be a submatrix of shape (2, 2), containing two rows and two columns.

To select an entire dimension, you can use a colon `:` without specifying a range, effectively sliding across that dimension. For example, in a 3-dimensional tensor, accessing `tensor[ :, :, 2 ]` will retrieve all elements along the first two dimensions where the third dimension is fixed at index 2.

Concatenation

Concatenation joins two or more tensors along an existing dimension, which is useful for merging data from multiple sources or consolidating batch results. In `PyTorch`, the `torch.cat` function performs concatenation, with the `dim` parameter specifying the dimension along which to join the tensors.

```
# Concatenating two 3x4 tensors along the first (row) dimension
tensor_a=torch.tensor([[1,2,3,4],[5,6,7,8],[9,0,1,2]])
tensor_b=torch.tensor([[3,4,5,6],[7,8,9,0],[1,2,3,4]])
concatenated = torch.cat((tensor_a, tensor_b), dim=0)
```

In this example, `torch.cat` joins `tensor_a` and `tensor_b` along the first dimension (rows), resulting in a 6×4 tensor:

$$\texttt{concatenated} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 0 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix}. \quad \text{Insert figure.}$$

Stacking

Stacking is similar to concatenation but introduces a new dimension, effectively increasing the tensor's order. The `torch.stack` function is commonly used for creating a batch of tensors or for operations requiring an additional dimension. Here, the `dim` parameter specifies where to insert the new dimension.

```
# Stacking two 3x4 tensors along a new dimension
stacked = torch.stack((tensor_a, tensor_b), dim=0)
```

This operation combines `tensor_a` and `tensor_b` along a new dimension at position 0, resulting in a $2 \times 3 \times 4$ tensor, where each layer along the new dimension corresponds to one of the original tensors. In this structure, each of the original matrices occupies a distinct layer along the new dimension, facilitating batching and higher-dimensional data handling in various applications.

Insert figure.

Transposing

Transposing swaps two dimensions in a tensor, a useful operation for reshaping data to match specific requirements in various operations, such as dot products and matrix multiplications. In `PyTorch`, the `transpose` function performs this operation efficiently. For a matrix, this means swapping rows and columns, but transposition can also be applied to higher-dimensional tensors.

For example, in a 3D tensor $T \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, transposition allows swapping of any two dimensions. Suppose we have a tensor T with dimensions $2 \times 3 \times 4$ as follows:

```
# Creating a 3D tensor of shape (2, 3, 4)
tensor = torch.tensor([[[ 1,  2,  3,  4],
                        [ 5,  6,  7,  8],
                        [ 9, 10, 11, 12]],

                       [[13, 14, 15, 16],
                        [17, 18, 19, 20],
                        [21, 22, 23, 24]]])

# Transposing the first and third dimensions
transposed_tensor = tensor.transpose(0, 2)
```

Here, we apply `tensor.transpose(0, 2)`, which swaps the first and third dimensions. This results in a tensor of shape $4 \times 3 \times 2$, where each element (i, j, k) in the original tensor is now located at (k, j, i) in the transposed tensor. The resulting tensor would look like this:

$$\texttt{transposed\_tensor} = \left[\begin{bmatrix} 1 & 13 \\ 5 & 17 \\ 9 & 21 \end{bmatrix}, \begin{bmatrix} 2 & 14 \\ 6 & 18 \\ 10 & 22 \end{bmatrix}, \begin{bmatrix} 3 & 15 \\ 7 & 19 \\ 11 & 23 \end{bmatrix}, \begin{bmatrix} 4 & 16 \\ 8 & 20 \\ 12 & 24 \end{bmatrix} \right].$$

For tensors of even higher order, transposing can rearrange any two of the multiple dimensions to achieve the desired ordering. This flexibility is particularly valuable in machine learning, where tensors may need to be reshaped and reordered for compatibility with network architectures and computational operations.

Permuting Dimensions

Permuting dimensions is a generalization of transposing that allows reordering multiple dimensions in a tensor. This operation is essential when preparing multi-dimensional data, like images, in different formats (e.g., converting between channel-first and channel-last formats in image processing). Unlike simple transposition,

which only swaps two dimensions, permuting allows you to reorder any number of dimensions simultaneously, providing greater flexibility for data manipulation.

For example, suppose we have a 3D tensor $T \in \mathbb{R}^{2\times3\times4}$, where the dimensions represent different aspects of the data. By permuting the dimensions, we can reorganize the structure to meet the input requirements of different neural network layers or processing functions.

```
# Creating a 3D tensor of shape (2, 3, 4)
tensor_3d = torch.rand(2, 3, 4)

# Permuting the dimensions to change the shape to (4, 2, 3)
permuted = tensor_3d.permute(2, 0, 1)
```

In this example, `tensor_3d.permute(2, 0, 1)` changes the order of dimensions from $(2, 3, 4)$ to $(4, 2, 3)$, effectively moving the third dimension to the first position, the first dimension to the second position, and the second dimension to the third position. The new shape of the tensor is $4 \times 2 \times 3$.

This function is especially useful in deep learning pipelines, where data with multiple channels or time steps often requires reordering for compatibility with various layers and models. Permuting dimensions ensures that data is in the correct format without changing the actual data values, enabling efficient model processing.

Squeezing and Unsqueezing Dimensions

Squeezing removes dimensions of size 1 (also called *singleton dimensions*) from a tensor, making it more compact. For example, if a tensor has a shape of 1×3, squeezing will remove the singleton dimension, transforming it into a 1D tensor of shape 3.

```
# Squeezing a 1x3 tensor to a 1D tensor of size 3
tensor_1d = torch.tensor([[1, 2, 3]])
squeezed = tensor_1d.squeeze()  # Output: tensor([1, 2, 3])
```

In this example, `tensor_1d` initially has a shape of 1×3, but after applying `.squeeze()`, the singleton dimension is removed, resulting in a tensor of shape 3.

Unsqueezing, on the other hand, adds a new dimension of size 1 at a specified position, effectively increasing the tensor's order. For instance, unsqueezing can transform a 1D tensor of shape 3 into a 2D tensor of shape 1×3 or 3×1, depending on the specified dimension.

```
# Adding a singleton dimension to a 1D tensor
tensor_1d = torch.tensor([1, 2, 3])
unsqueezed=tensor_1d.unsqueeze(0)  #Output: tensor([[1,2,3]])
```

In this code snippet, `tensor_1d` starts as a 1D tensor with shape 3. The `.unsqueeze(0)` operation adds a singleton dimension at position 0, resulting in a shape of 1×3.

These `squeeze` and `unsqueeze` operations are integral to tensor manipulation in deep learning, as they allow for seamless reshaping of data to match model architectures, enhance computational efficiency, and ensure compatibility with layer requirements. By adding or removing dimensions, these operations enable flexible handling of complex, multi-dimensional tensors within `PyTorch` models.

D.2 Automatic Differentiation in MLPs

Automatic differentiation is a powerful technique that enables the efficient computation of gradients in neural networks, particularly in models like Multi-Layer Perceptrons (MLPs). In MLPs, gradients are essential for optimizing model parameters during training. Specifically, automatic differentiation computes the derivative of the loss function with respect to each parameter, allowing optimization algorithms, such as gradient descent, to iteratively adjust these parameters to minimize the error. In this section, we explore the principles of automatic differentiation within the context of MLPs, explain how gradients are computed, and demonstrate an implementation using PyTorch's `autograd`.

An MLP consists of multiple layers, where each layer applies an affine transformation followed by a non-linear activation. For example, an MLP with two hidden layers is described by:

$$\mathbf{x}_1 = \sigma(A\mathbf{u} + \mathbf{b}), \tag{22}$$

$$\mathbf{x}_2 = \sigma(B\mathbf{x}_1 + \mathbf{c}), \tag{23}$$

$$\widehat{\mathbf{y}} = C\mathbf{x}_2 + \mathbf{d}, \tag{24}$$

where $\mathbf{u}$ is the input vector, A, B, and C are the weight matrices for the first, second, and output layers, respectively, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ are bias vectors for each layer, σ is a non-linear activation function.

During training, the MLP learns to minimize a loss function $\mathcal{L}(\widehat{\mathbf{y}}, \mathbf{y})$, which measures the difference between the predicted output $\widehat{\mathbf{y}}$ and the true output $\mathbf{y}$. The gradients of $\mathcal{L}$ with respect to each parameter (A, B, C, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$) are computed using automatic differentiation.

Automatic Differentiation with PyTorch

PyTorch's `autograd` module enables automatic differentiation by recording operations on tensors and constructing a computational graph. Once the forward pass is complete, PyTorch can backpropagate through this graph to compute gradients for all parameters. In the backward pass, PyTorch applies the chain rule of calculus:

$$\frac{\partial \mathcal{L}}{\partial A} = \frac{\partial \mathcal{L}}{\partial \widehat{\mathbf{y}}} : \frac{\partial \widehat{\mathbf{y}}}{\partial \mathbf{x}_2} : \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} : \frac{\partial \mathbf{x}_1}{\partial A},$$

where ":" indicates the tensor contraction product that contracts over a single index. By applying the chain rule, the gradient is computed efficiently layer-by-layer, from the output layer back to the input layer.

Implementation of Gradient Computation with `autograd`

The following example demonstrates a simple MLP with two hidden layers in PyTorch, illustrating the steps involved in defining the model, performing a forward pass, calculating the loss, and using `autograd` to compute gradients with respect to the model's parameters.

Below, we break down the code and explanation into a sequence of steps.

1. **Import Libraries and Define the Model Structure**: First, we import the necessary PyTorch libraries:

```
import torch
import torch.nn as nn
import torch.optim as optim
```

   Next, we define the architecture of the MLP model, with two hidden layers. The `MLP` class below specifies each layer, including the input layer, two hidden layers, and the output layer:

```
# Define the MLP model
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size1,
        hidden_size2, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, output_size)
```

   In the `forward` function, we apply a ReLU activation after each of the first two layers, producing the final output after the last layer without an activation function, as it suits our regression task:

```
    def forward(self, u):
        x1 = torch.relu(self.fc1(u))
        x2 = torch.relu(self.fc2(x1))
        y = self.fc3(x2)
        return y
```

2. **Initialize Model, Define Loss, and Optimizer**: With the model defined, we initialize it by specifying the input and output sizes, along with the sizes of the two hidden layers. Additionally, we define a Mean Squared Error (MSE) loss function and use the Adam optimizer to train the model:

```
# Instantiate the model, define a loss function, and an
    optimizer
input_size = 3
hidden_size1 = 5
hidden_size2 = 5
output_size = 1
model = MLP(input_size, hidden_size1, hidden_size2,
    output_size)
```

For this example, we provide a dummy input $\mathbf{u} = [1.0, 2.0, 3.0]$ with `requires_grad=True` to enable gradient tracking, and a target output of $y_{\text{true}} = 1.0$:

```
# Dummy input and target output
u = torch.tensor([[1.0, 2.0, 3.0]], requires_grad=True)
y_true = torch.tensor([[1.0]])
```

3. **Forward Pass**: Now, we perform the forward pass by feeding the input $\mathbf{u}$ through the model to obtain the predicted output y_{pred}:

```
# Forward pass
y_pred = model(u)
```

4. **Loss Calculation**: Next, we calculate the loss by comparing y_{pred} to the target output y_{true}, using the MSE loss function:

```
# Define loss function (Mean Squared Error)
loss_fn = nn.MSELoss()
loss = loss_fn(y_pred, y_true)
```

5. **Backward Pass and Gradient Computation**: To compute the gradients of the loss with respect to each parameter, we perform the backward pass using `loss.backward()`. PyTorch's `autograd` automatically calculates these gradients, which are stored in the `.grad` attribute of each parameter:

```
# Backward pass (compute gradients)
loss.backward()
```

6. **Displaying Gradients**: Finally, we inspect the computed gradients for each parameter by iterating through the model's parameters. This allows us to view the gradient values, which indicate the direction and magnitude of change needed to minimize the loss:

```
# Display gradients for each parameter
for name, param in model.named_parameters():
    if param.requires_grad:
        print(f"Gradient of {name}: {param.grad}")
```

In this process, PyTorch builds a computational graph during the forward pass, which it then uses to compute gradients efficiently during the backward pass by applying the chain rule of calculus. Automatic differentiation, as shown here, is essential for training neural networks and allows for flexible and efficient gradient computation without manual derivation.

Table 1: Gradients of Model Parameters

Parameter	**Gradient Values**
`fc1.bias`	$\begin{bmatrix} -0.2030 & 0.0000 & 0.0000 & 0.0000 & -0.4359 \end{bmatrix}$
`fc2.weight`	$\begin{bmatrix} 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ -0.9312 & 0.0000 & 0.0000 & 0.0000 & -0.6712 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ -0.8982 & 0.0000 & 0.0000 & 0.0000 & -0.6474 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \end{bmatrix}$
`fc2.bias`	$\begin{bmatrix} 0.0000 & -0.7808 & 0.0000 & -0.7532 & 0.0000 \end{bmatrix}$
`fc3.weight`	$\begin{bmatrix} 0.0000 & -1.3830 & 0.0000 & -0.3224 & 0.0000 \end{bmatrix}$
`fc3.bias`	$\begin{bmatrix} -2.2170 \end{bmatrix}$

Table 1 displays the computed gradients for each parameter of the multi-layer perceptron (MLP) model. Each parameter name corresponds to a specific part of the model:

- `fc1`, `fc2`, and `fc3` denote the fully connected layers in the MLP model. Here, `fc1` is the first hidden layer, `fc2` is the second hidden layer, and `fc3` is the output layer.
- `.weight` represents the weight matrix associated with each fully connected layer. These weights determine how the input or previous layer's output is transformed to produce the next layer's output.
- `.bias` refers to the bias vector for each layer, which allows the network to make adjustments independently of the weighted input. Adding a bias term provides the model with greater flexibility, enabling it to fit the data more effectively.

The gradient values in the table indicate the direction and magnitude of the adjustments required to minimize the model's loss with respect to each parameter. For example, in `fc1.bias`, non-zero gradient values suggest that adjustments to these biases will directly impact the model's error. Overall, this table provides insight into the parameters actively involved in minimizing the model's error during training, with non-zero gradients highlighting the specific weights and biases that influence the model's performance.

In this example, PyTorch's `autograd` performs automatic differentiation by constructing a dynamic computational graph of operations during the forward pass. During the backward pass, it applies the chain rule to compute gradients of the loss with respect to each parameter. This automatic differentiation enables efficient training of neural networks, as it avoids the need for manual calculation of gradients. With `autograd`, PyTorch provides an intuitive and powerful tool for building and training complex neural networks.

Exercises

1. 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, use `delta_t = 0.01` and simulate for `num_steps=10000` steps. Set the initial angle to $x_{1,0} = \pi - 0.1$ (close to the vertical position) and the initial angular velocity to zero, i.e., $x_{2,0} = 0$. Set the input to be:

```
# Sinusoidal input
amplitude = 0
frequency = 0.5
# Sinusoidal input over time
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 $x_{1,k}$ and the y-axis is $x_{2,k}$. 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:

```
# 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
```

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

2. 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:

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

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, for each observation, identifies the most likely hidden state by 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 the hidden state sequence you estimated in the previous step.

3. Consider a dynamical system characterized by the following system of recursions:

$$\begin{aligned} x_{1,k+1} &= x_{1,k}/2 + x_{2,k}/4 + x_{3,k}/8 + \sin(3k), \\ x_{2,k+1} &= x_{1,k}, \\ x_{3,k+1} &= x_{2,k}, \end{aligned}$$

with the observation equation:

$$y_k = x_{1,k} + x_{2,k} + x_{3,k}.$$

(a) Write the system in matrix form.

(b) Is the system stable? *Hint*: The state matrix is stable if its eigenvalues are inside the unit circle.

(c) Determine whether the LSSM is in controllable or observable canonical form. *Hint*: Refer to Example 6 for guidance on canonical forms.

(d) Simulate the system in Python for 100 time steps, using the initial condition $\mathbf{x}_0 = \mathbf{0}$. Plot the evolution of y_k over time. Verify whether the system is stable.

(e) Write the state and output equations of the system in observable canonical form.

(f) Simulate the transformed system in Python for 100 time steps, using the initial condition $\boldsymbol{\xi}_0 = \mathbf{0}$. Plot the evolution of the output y_k over time.

(g) Compare the simulations of the original and transformed systems. Justify your observations.

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

$$\begin{aligned} x_{k+1} &= x_k/2 + u_k, \\ y_k &= x_k. \end{aligned}$$

Answer the questions below:

(a) Starting with the initial condition $x_0 = 2$, 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 $\{H_0, H_1, H_2, \ldots\}$ and the step function as the input sequence, compute the convolution of the Markov parameters with the input. Express the resulting sequence $\{y_0, y_1, y_2, \ldots\}$ 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 state-space matrices $\widetilde{A}, \widetilde{B}, \widetilde{C}, \widetilde{D}$ and the transformed initial condition ξ_0 for the new state-space model in terms of a, b, c, and d.

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

5. Consider an LSSM model with system matrices (A, B, C, D). Prove the following identities:

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

6. Prove that

$$\frac{\partial \left(UXV\right)}{\partial X} = UV^\intercal$$

7. Prove that

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

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?

