Topic 5A: Recurrent Neural Networks

Victor M. Preciado

Contents

1 Traditional Neural Networks for Time-Series Data

Neural networks have long been utilized for a variety of machine learning tasks, including classification, regression, and function approximation. However, when applied to time-series data, traditional feedforward neural networks exhibit significant limitations in capturing temporal dependencies across sequential data points. In this section, we examine the mathematical structure of standard neural networks, highlights their shortcomings for time-series data, and presents the motivation for recurrent neural networks (RNNs) and their gated extensions.

1.1 Feedforward Neural Networks in State-Space Formulation

A traditional **feedforward neural network (FNN)** consists of multiple interconnected layers of artificial neurons, where each neuron processes inputs from the previous layer using a weighted summation followed by a non-linear activation function. The term FNN refers to a broad category of neural networks represented as a **directed acyclic graph (DAG)**, characterized by connections that do not form cycles. This structure ensures that data flows in a single direction—from input to output—without looping back. A particularly relevant type of FNN is the **multilayer perceptron (MLP)**, where each neuron is connected to all the neurons in the previous layer, allowing for expressive modeling capabilities. In the following sections, we present FNNs (and MLPs) through a state-space formulation that frames the mapping from input to output within a dynamical-systems perspective.

Insert diagram of FNN and MLP.

In what follows, we describe FNNs as a discrete-time dynamical system using a nonlinear state-space formulation. In this context, we assign a virtual time index l to each layer in the FNN and consider that the vector of hidden states evolves through this index according to a recursive state equation. The evolution of the hidden states in a feedforward network with L layers is given by:

$$\mathbf{x}_l = \sigma(A_l\,\mathbf{x}_{l-1} + \mathbf{b}_{l-1}) \quad \text{for } l \in [L] \text{ and } \mathbf{x}_0 = \mathbf{u}, \tag{1}$$

where $\mathbf{x}_l$ is the hidden state vector at the l-th layer, $A_l \in \mathbb{R}^{n_l \times n_{l-1}}$ is the *layer-variant* state matrix specific to the l-th layer, $\mathbf{b}_l \in \mathbb{R}^{n_l}$ is the **bias vector** that can be interpreted as a virtual input vector influencing the state dynamics at each layer, and $\sigma(\cdot)$ is the non-linear activation function applied element-wise. Note that the input to the FNN serves as the initial condition of this dynamical system.

The output of the neural network is defined by the following output equation:

$$\mathbf{y} = C\mathbf{x}_L + \mathbf{d},$$

where C and $\mathbf{d}$ are parameters of the final output layer, and $\mathbf{y}$ represents the output of the neural network corresponding to the initial input $\mathbf{u}$. Hence, the output

represents the observation of the hidden state after L virtual time steps of evolution when the system starts with an initial condition $\mathbf{u}$ and is driven by an input sequence $(\mathbf{b}_l)_{l=0}^{L-1}$.

Diagram of the temporal evolution.

This state-space representation underscores that an FNN behaves as a dynamical system where the state evolves through discrete stages (i.e., layers) for a finite horizon of L layers. Unlike typical linear state-space models (LSSMs), where the state matrix and input matrix are constant over time, the FNN features a layer-variant state matrix A_l that differs across layers. Additionally, the nonlinearity introduced by the activation function $\sigma(\cdot)$ distinguishes the network from purely linear systems, enabling it to capture intricate, non-linear mappings. The FNN induces a parametric mapping between an input $\mathbf{u}$ and an output $\mathbf{y}$ that we denote by:

$$\mathbf{y} = \Phi_{\boldsymbol{\theta}}(\mathbf{u}),$$

where $\Phi_{\boldsymbol{\theta}}(\cdot)$ represents the overall function of the network, parameterized by $\boldsymbol{\theta}$, encompassing all state matrices A_l, bias vectors $\mathbf{b}_l$, and the output layer parameters C and $\mathbf{d}$.

During training, we have a dataset $\mathcal{D} = \{(\mathbf{u}_k, \mathbf{y}_k)\}_{k=1}^{N}$, consisting of N input-output pairs. Using an iterative optimization process, the sequence of state matrices $(A_l)_{l=1}^{L}$ and virtual input vectors $\{\mathbf{b}_l\}_{l=1}^{L}$ are adjusted so that the nonlinear dynamical system described by (1) produces a collection of predicted outputs $\widehat{\mathbf{y}}_k$ by running the dynamics over a finite horizon L when initialized with the input $\mathbf{u}_k$. This iterative process optimizes the parameters to minimize the discrepancy between the predicted outputs and the true outputs in the dataset, enabling the network to learn complex mappings from inputs to outputs effectively.

1.2 Control Theory for FNNs

TBD... LeCun 1988

1.3 Neural ODEs

TBD

Hamiltonian Neural ODEs

TBD

2 FNNs for Time-Series Data

While the state-space dynamical formulation of FNN helps in understanding their mathematical structure, it also underscores their inherent limitations when applied to time-series data. In traditional feedforward architectures, each input at time step k is processed independently:

$$\mathbf{y}_k = \Phi_{\boldsymbol{\theta}}(\mathbf{u}_k),$$

where $\Phi_{\boldsymbol{\theta}}(\cdot)$ represents the overall function of the network, parameterized by $\boldsymbol{\theta}$, encompassing all state matrices A_l, bias vectors $\mathbf{b}_l$, and the output layer parameters C and $\mathbf{d}$. However, this formulation lacks any inherent mechanism for retaining or transferring information across different time steps. Consequently, the model is limited in its ability to capture dependencies between sequential inputs $\mathbf{u}_{k-1}, \mathbf{u}_k, \mathbf{u}_{k+1}$. This shortcoming makes traditional feedforward networks unsuitable for tasks requiring an understanding of long-term dependencies, such as time-series forecasting or anomaly detection.

One straightforward approach to addressing this limitation is the **stack model**, where a sequence of past inputs is concatenated to create a composite input vector:

$$\overline{\mathbf{u}}_k = [\mathbf{u}_k, \mathbf{u}_{k-1}, \ldots, \mathbf{u}_{k-m}],$$

with m representing the window size. This allows the network to learn from a fixed time horizon of past inputs, providing a limited degree of temporal awareness. The output at time k is then:

$$\mathbf{y}_k = \overline{\Phi}_{\boldsymbol{\theta}}(\overline{\mathbf{u}}_k).$$

While this method introduces some temporal context, it comes with significant limitations. The fixed time horizon T restricts the model's ability to capture long-term dependencies, as any information beyond this window is ignored. This approach can also increase computational complexity, as larger window sizes lead to high-dimensional input vectors that may contribute to overfitting.

These limitations underscore the need for more advanced architectures capable of dynamically retaining and managing memory over time. This brings us to Recurrent Neural Networks (RNNs), which introduce feedback connections that allow the model to maintain a memory of past states and capture dependencies across variable-length sequences.

3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) address the temporal limitations of feedforward networks and the stack model by incorporating recurrent connections that enable information retention across time steps. In this section, we describe **Recurrent Neural Networks** (RNNs) as another nonlinear extension of Linear State-Space

Models (LSSMs) for modeling and forecasting time-series data. While LSSMs provide a robust linear framework for dynamical systems, they often fall short in capturing complex, nonlinear dependencies across time. Recurrent Neural Networks build upon this foundation by introducing nonlinear dynamics into the hidden state evolution, enabling the modeling of a wide range of real-world systems where relationships between inputs, outputs, and states are inherently nonlinear. RNNs are especially powerful in scenarios where patterns in the data are governed by long-term dependencies and nonlinear transformations. This capability makes RNNs a preferred choice in time-series forecasting, language processing, and other applications where sequential data structure plays a central role. By incorporating nonlinear activation functions within the state-update equation, RNNs can represent a broader class of dynamical behaviors than their linear counterparts.

Building on the reader's understanding of LSSMs, we will first explore the structure and principles behind the standard RNN model. We will examine how RNNs generalize LSSMs by adapting familiar state-space notation to the RNN setting, thus grounding RNNs within a known framework. This treatment highlights the role of recurrent connections, the recursive nature of the hidden states, and the role of state and output matrices. We then introduce and analyze the challenges associated with training RNNs. These include the issues of vanishing and exploding gradients, which stem from the repeated application of the state matrix and can hinder the model?s ability to learn long-term dependencies. In response to these issues, we delve into popular RNN variants such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), both of which are specifically designed to improve gradient stability over long sequences. The chapter concludes with discussions on advanced RNN applications in time-series forecasting and strategies for enhancing performance, such as sequence normalization and adaptive learning rate techniques. By the end of this chapter, readers will gain a solid understanding of RNN architectures, their underlying principles, and practical techniques for implementing these models in complex time-series settings.

Recurrent Neural Networks (RNNs) extend the framework of Linear State-Space Models (LSSMs) to nonlinear dynamics, enabling the capture of complex temporal dependencies in sequential data. This section introduces the mathematical structure of RNNs, highlighting how they build on the recursive nature of LSSMs to model nonlinear relationships in time-series forecasting. Here, we adopt a notation consistent with the standard state-space model, where we use $\mathbf{x}_k$ for the hidden state, $\mathbf{u}_k$ for the input, and $\mathbf{y}_k$ for the output, and matrices A, B, C, and D to represent the system dynamics.

3.1 Nonlinear Extension of LSSMs

LSSMs represent linear dynamical systems where the evolution of the hidden state is governed by a state matrix A and the influence of input $\mathbf{u}_k$ through a matrix B:

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

where η_k denotes process noise. In contrast, RNNs allow for *nonlinear* transformations of both the hidden states and inputs, represented by a nonlinear activation function f (typically a sigmoid or hyperbolic tangent function). The RNN state update is therefore defined as:

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

where $\mathbf{b}_x \in \mathbb{R}^n$ is a **bias term** for the hidden state that replaces the noise term in the standard LSSM.

The predicted output $\widehat{\mathbf{y}}_k$ in an LSSM is modeled as:

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

where ϵ_k is the measurement noise. In contrast, in an RNNs the predicted output $\widehat{\mathbf{y}}$ at each time step includes a nonlinear transformation, given by:

$$\boxed{\widehat{\mathbf{y}}_k = g\left(C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{b}_y\right),}$$

where $\mathbf{b}_y \in \mathbb{R}^p$ is an **output bias** that replaces the measurement noise in the LSSM, and g is the output activation function (often the identity function for regression tasks).

As was the case in LSSMs, the RNN structure relies on the concept of *hidden states* that evolve over time, capturing information from previous time steps to influence current predictions. The hidden state $\mathbf{x}_k$ encapsulates information from the information set $\mathcal{F}_{k-1}$. The recurrent connection via the state matrix A means that the model's memory of past inputs persists as long as it remains in the hidden state vector. However, due to the nonlinearity of f, the propagation of this information is complex, allowing the RNN to model intricate relationships between time steps.

4 Training RNNs for Time-Series Forecasting

Training recurrent neural networks (RNNs) for time-series forecasting requires unique adaptations to traditional gradient-based optimization techniques due to the sequential data structure and recursive architecture inherent to RNNs. In this section, we introduce Backpropagation Through Time (BPTT), the primary training algorithm tailored for RNNs. Alongside BPTT, we discuss strategies for overcoming challenges

such as vanishing and exploding gradients—issues that frequently arise due to the recursive nature of RNNs. Techniques for gradient stabilization and regularization are covered to ensure both reliable convergence and robust forecasting performance.

In a supervised time-series forecasting setting, the goal of an RNN is to learn a set of parameters that enables it to map an input sequence to a desired output sequence. The parameters we aim to optimize in an RNN for time-series forecasting are:

$$\theta = \{A, B, C, D, \mathbf{b}_x, \mathbf{b}_y, \mathbf{x}_0\}.$$

These parameters are determined by minimizing a suitable loss function, often the mean squared error (MSE) between the predicted output sequence $\{\widehat{\mathbf{y}}_k \colon k \in [L]\}$ and the actual observed sequence $\{\mathbf{y}_k \colon k \in [L]\}$:

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

This loss function quantifies the difference between predicted and actual values across all time steps, and its minimization encourages the RNN to capture meaningful temporal patterns in the data. By systematically reducing this loss, the RNN learns to model dependencies across the time sequence, improving its forecasting accuracy.

4.1 Backpropagation Through Time (BPTT)

Backpropagation Through Time (BPTT) extends the backpropagation algorithm to compute gradients over sequences, enabling effective training for RNNs. Unlike traditional feedforward networks where gradient computation only involves traversing the network once, BPTT must propagate gradients backward through each time step, creating a computational graph that *unrolls* through time. Below, we detail the mathematical foundations of BPTT, the steps involved in calculating gradients, and the unique challenges that arise when applying this method to sequential data. By understanding these principles, we can better address computational considerations and design effective training strategies for time-series forecasting with RNNs.

Unrolled Computational Graph for an RNN

For an RNN, the hidden state update at each time step k is given by:

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

where f is a nonlinear activation function, $\mathbf{x}_k$ represents the hidden state, and $\mathbf{u}_k$ is the input at time k. The corresponding output is modeled as:

$$\mathbf{y}_k = g\left(C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{b}_y\right),$$

where g is the output activation function, and $\mathbf{y}_k$ denotes the predicted output.

Over a sequence of length L, BPTT constructs a computational graph that spans from $k = 1$ to $k = L$. Each node in this graph represents a time step, and each edge captures the recursive dependencies inherent in the state update. This unrolled graph facilitates gradient computation with respect to the model parameters by enabling application of the chain rule across time steps.

Gradient Computation through BPTT

The objective in BPTT is to minimize a cumulative loss function $\mathcal{L}(\boldsymbol{\theta}; \mathbf{u}_{0:L}, \mathbf{y}_{0:L})$, which, for a sequence of length L, can be expressed as:

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

where ℓ is a loss function (e.g., mean squared error) that measures the discrepancy between the actual outputs $\mathbf{y}_k$ and the predicted outputs $\widehat{\mathbf{y}}_k$, and $\boldsymbol{\theta} = \{A, B, C, D, \mathbf{b}_x, \mathbf{b}_y\}$ represents the set of model parameters.

To compute the gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$, we apply the chain rule iteratively, leveraging the recursive nature of the hidden states:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \sum_{k=0}^{L} \frac{\partial \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} \cdot \frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} \cdot \frac{\partial \mathbf{x}_k}{\partial \boldsymbol{\theta}}.$$

Step-by-Step Gradient Calculation in BPTT

1. **Output Gradient**: The derivative of the loss with respect to the output is computed as:
$$\frac{\partial \ell(\mathbf{y}_k, \widehat{\mathbf{y}}_k)}{\partial \widehat{\mathbf{y}}_k} = 2\left(\widehat{\mathbf{y}}_k - \mathbf{y}_k\right).$$

2. **Jacobian of Output with Respect to Hidden State**: The gradient of $\widehat{\mathbf{y}}_k$ with respect to $\mathbf{x}_k$ is:
$$\frac{\partial \widehat{\mathbf{y}}_k}{\partial \mathbf{x}_k} = C \cdot g'(C\mathbf{x}_k + D\mathbf{u}_k + \mathbf{b}_y),$$
where g' represents the derivative of the output activation function.

3. **Recursive Gradient of Hidden States**: Using the chain rule for each time step, we propagate gradients backward through each hidden state:
$$\frac{\partial \mathbf{x}_k}{\partial \boldsymbol{\theta}} = f'(A\mathbf{x}_{k-1} + B\mathbf{u}_k + \mathbf{b}_x) \cdot \left(A \cdot \frac{\partial \mathbf{x}_{k-1}}{\partial \boldsymbol{\theta}} + \frac{\partial \mathbf{x}_k}{\partial \mathbf{u}_k} \cdot \frac{\partial \mathbf{u}_k}{\partial \boldsymbol{\theta}}\right).$$

Challenges in BPTT

BPTT can be computationally intensive, as it requires storing activations and gradients at each time step. Furthermore, BPTT is vulnerable to gradient instability issues, namely, the vanishing and exploding gradient problems. In the next section, we discuss these issues and present techniques to mitigate them.

4.2 Gradient Stability and Regularization Techniques

Training RNNs on long sequences often leads to gradients that either diminish to zero (vanishing gradient problem) or grow uncontrollably (exploding gradient problem), thus impacting the model's ability to learn long-term dependencies.

Vanishing and Exploding Gradients

EXPLAIN USING THE ERGODIC THEOREM AND LYAPUNOV EXPONENT... HANIN 2008

- **Vanishing Gradients**: When the eigenvalues of the state matrix A are less than one in magnitude, repeated matrix multiplications during BPTT cause the gradient to decay exponentially. This decay makes it difficult for the network to learn dependencies across long sequences.
- **Exploding Gradients**: Conversely, when the eigenvalues of A are greater than one in magnitude, the gradient grows exponentially over time steps, leading to unstable updates and potential divergence during training.

Stabilization Techniques

Several regularization techniques are employed to address gradient instability, enabling the network to handle longer sequences more effectively.

- **Gradient Clipping**: Gradient clipping limits the maximum value of gradients during backpropagation, preventing them from exceeding a specified threshold. If the gradient norm surpasses this threshold, it is scaled down as follows:

$$\text{if } \|\nabla \mathcal{L}\| > \tau, \quad \nabla \mathcal{L} \leftarrow \tau \frac{\nabla \mathcal{L}}{\|\nabla \mathcal{L}\|},$$

  where τ is the clipping threshold.

- **Weight Regularization**: Regularizing the weights of the state matrix A encourages smaller eigenvalues, reducing the likelihood of exploding gradients. This can be achieved through ℓ_2-norm regularization:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \|A\|_2^2,$$

where λ is a regularization parameter.

- **Dropout**: Dropout randomly sets a fraction of hidden state units to zero during training, reducing overfitting and improving generalization. In RNNs, dropout can be applied between time steps or within hidden layers, though temporal consistency should be maintained.
- **Learning Rate Scheduling**: Adjusting the learning rate dynamically, either based on validation performance or gradient norms, helps avoid large updates that can destabilize training.
- **Layer Normalization**: Layer normalization normalizes the activations within each layer, thus stabilizing hidden state dynamics across time steps.

Implementation Considerations

Implementing gradient stabilization techniques requires monitoring gradient norms and adjusting hyperparameters based on model performance. Key hyperparameters, such as the learning rate, regularization strength, and dropout probability, are essential in ensuring stable training and capturing complex temporal patterns effectively.

...

...

4.3 Challenges with RNNs and the Need for Gated Extensions

While RNNs can theoretically model long-term dependencies, in practice, they are limited by the vanishing and exploding gradient problems. During backpropagation through time (BPTT), the gradients associated with earlier time steps can either diminish or explode exponentially, making it difficult for the model to learn long-range dependencies effectively.

To overcome these issues, *gated* RNN extensions such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks were developed. These architectures introduce gates that control the flow of information within the network, enabling the retention or discarding of information as needed. For example, an LSTM includes an input gate, a forget gate, and an output gate, which collectively manage the cell state $\mathbf{c}_t$ and hidden state $\mathbf{h}_t$ at each time step:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t,$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \phi(\mathbf{c}_t),$$

where:

- $\mathbf{f}_t$ (forget gate), $\mathbf{i}_t$ (input gate), and $\mathbf{o}_t$ (output gate) are vectors that regulate information flow,
- $\odot$ denotes the Hadamard (element-wise) product,
- $\tilde{\mathbf{c}}_t$ is the candidate cell state.

These mechanisms allow LSTMs and GRUs to retain relevant information for longer durations, mitigating the issues of vanishing and exploding gradients.

Conclusion: The development of RNNs and their gated extensions represents a significant step forward in time-series analysis, enabling the modeling of complex, long-term dependencies that feedforward neural networks cannot handle. However, even these architectures face limitations in parallel processing and handling extremely long sequences efficiently, setting the stage for further advancements such as Transformer architectures, which are covered in subsequent chapters.

...

...

5 Variants of Recurrent Neural Networks

In this section, we discuss three popular RNN variants that address specific challenges in RNN training, particularly long-term dependency issues. Each variant builds on the RNN structure with modifications that allow the network to selectively retain or forget information over time.

5.1 Long Short-Term Memory (LSTM) Networks

5.1.1 Motivation and Structure of LSTMs

We introduce LSTMs as a solution to the vanishing gradient problem and a method for capturing long-term dependencies in time-series data. The section outlines the structure of an LSTM cell, including the input, forget, and output gates, as well as the cell state and hidden state.

5.1.2 Mathematical Formulation of LSTMs

The mathematical equations defining LSTM dynamics are presented, with an emphasis on the role of each gate and the cell state in information retention and flow. This subsection relates the LSTM?s gates and memory cell to concepts in LSSMs and discusses how these structures allow LSTMs to model complex temporal dependencies.

5.2 Gated Recurrent Unit (GRU) Networks

5.2.1 Motivation and Structure of GRUs

We introduce GRUs as a streamlined alternative to LSTMs, designed to reduce computational complexity while retaining the ability to model long-term dependencies. This section covers the main components of a GRU cell, including the reset and update gates.

5.2.2 Mathematical Formulation of GRUs

We present the mathematical equations defining GRU dynamics, detailing the operations within each gate and the update of the hidden state. A comparison to LSTMs is provided to highlight the differences in structure and computational efficiency.

5.3 Other RNN Variants

This section briefly surveys other RNN variants and extensions, such as bidirectional RNNs, which can incorporate information from both past and future states, and attention-based mechanisms, which improve focus on relevant parts of a sequence. Each variant's application to time-series forecasting is discussed at a high level.

6 Applications of RNNs in Time-Series Forecasting

6.1 Forecasting Financial Time-Series

We explore the application of RNNs in predicting stock prices, currency exchange rates, and other financial metrics. This section highlights the benefits of using RNNs for modeling nonlinear dependencies in complex financial datasets.

6.2 Energy Consumption and Demand Forecasting

In this section, we discuss how RNNs are used for predicting energy demand and consumption patterns, covering the advantages of capturing seasonality, trends, and sudden fluctuations in data.

6.3 Healthcare and Biomedical Time-Series

RNN applications in healthcare are covered here, including forecasting patient health metrics, analyzing EEG and ECG data, and predicting disease outbreaks. This section highlights the importance of long-term dependency modeling in biomedical contexts.

7 Challenges and Future Directions in RNN-Based Forecasting

7.1 Scalability and Computational Efficiency

We discuss the challenges of scaling RNNs to larger datasets and longer sequences, focusing on the computational demands of BPTT and potential optimization methods.

7.2 Alternative Architectures and Hybrid Models

The limitations of RNNs in handling long-term dependencies and the emergence of alternative architectures, such as Transformers, are covered here. We discuss hybrid models that combine RNNs with other approaches to improve performance on complex time-series tasks.

We summarize the advantages and limitations of RNNs and their variants in time-series forecasting, reiterating their nonlinear modeling power and challenges in training stability. This chapter sets the foundation for the next chapter on advanced architectures, such as Transformers, that continue to push the boundaries of sequential modeling in machine learning.