Recurrent Neural Networks: A Mathematical Overview

Jayin Khanna
MAT399: UG Seminar Presentation 3 Report
Shiv Nadar University

Abstract

After the advent of Neural networks, there were multiple architectures for various different tasks and applications of Deep Learning. Recurrent Neural Networks is one such architecture derived from neural networks which handle data which has a sequential nature. For instance, RNNs are heavily used in deep learning based time series analyis, spech recognition, language models, image and text processing. Essentially any task that is sequential. This report provides a comprehensive overview of Recurrent neural networks, delving deep into their history, applications, mathematical formulation, forward propagation, backpropagation through time, gradient computation, and the vanishing gradient problem.

1 Introduction

The aim of this report is to understand the need to build new architectures for handling data which has a sequential and time series nature to it. Such models are called sequential models. These models have a special architecture which is different from a traditional neural network.
This report focuses on Recurrent Neural Networks (RNNs), their theoretical foundation, training methodology, and evolution into more robust architectures. The structure of the report is as follows:

- History of Recurrent Neural Networks (RNNs)
- Motivation for Recurrent Neural Networks
- Preliminaries: Introduction to Sequence and Language Modeling
  - Sequence Modeling
  - Language Modeling as a Special Case
  - Markov Models and the Context Window
  - Limitations and Motivation for Neural Models
  - Example of What We Don't Want in an RNN
- Recurrent Neural Network (RNN) Architecture
  - 1. Model Structure
  - 2. Loss Function
- Backpropagation Through Time (BPTT)
  - Local Gradients
  - Gradients w.r.t Parameters
  - Recursive Gradient Flow
- Gradient Accumulation Across Time
  - Unrolling the Gradient for $\mathbf{W}_{hh}$

2 History of Recurrent Neural Networks (RNNs)

RNNs emerged in the 1980s as an extension of feedforward neural networks to handle **sequential and temporal data**.
Traditional neural networks treat inputs as **independent**, making them unsuitable for tasks where **order and context** are crucial.
RNNs introduced the concept of a **hidden state** that is updated over time, enabling the network to **retain memory** of previous inputs.
This design made RNNs especially useful for applications like:

- *Speech recognition*
- *Language modeling*
- *Time-series prediction*

Early RNNs faced major training issues:

- **Vanishing gradients**: gradients shrink exponentially during backpropagation.
- **Exploding gradients**: gradients grow exponentially and destabilize training.

To address these issues, new architectures were developed:

- **Long Short-Term Memory (LSTM)** networks (1990s)
- **Gated Recurrent Units (GRUs)** (2010s)

These models introduced **gating mechanisms** that control information flow, allowing for better:

- Gradient propagation over long sequences
- Stability and learning of long-term dependencies

3 Motivation for Recurrent Neural Networks

Many real-world tasks involve **sequential information** and evolve over time:

- *Natural Language Processing*
- *Video analysis*
- *Sensor and financial data*

Traditional feedforward networks treat each input **independently**, ignoring context and sequence structure. RNNs allow **information to persist** through their hidden state, effectively giving the network a **memory** of previous inputs.

This enables RNNs to:

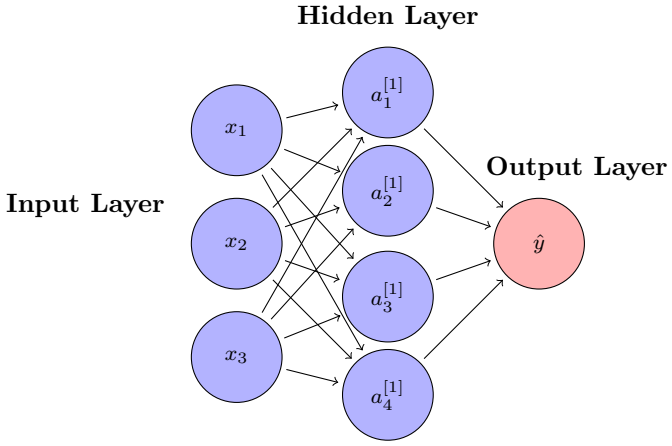
- Understand dependencies between data points across time
- Handle **variable-length sequences**
- Model **context-aware behavior**

The need to learn patterns over time and preserve temporal structure drove the development and widespread adoption of RNN-based models.

To address this, we seek models that:

1. Handle inputs of arbitrary length.
2. Capture dependencies across time—i.e., between earlier and later inputs.
3. Apply the same function (with shared parameters) at each time step to process sequential data consistently.

Recurrent Neural Networks (RNNs) achieve this by introducing a **hidden state (not hidden layer!)** that evolves over time, enabling the network to retain and propagate information through the sequence. This hidden state is updated recursively based on the current input and the previous hidden state using shared parameters across all time steps.



center

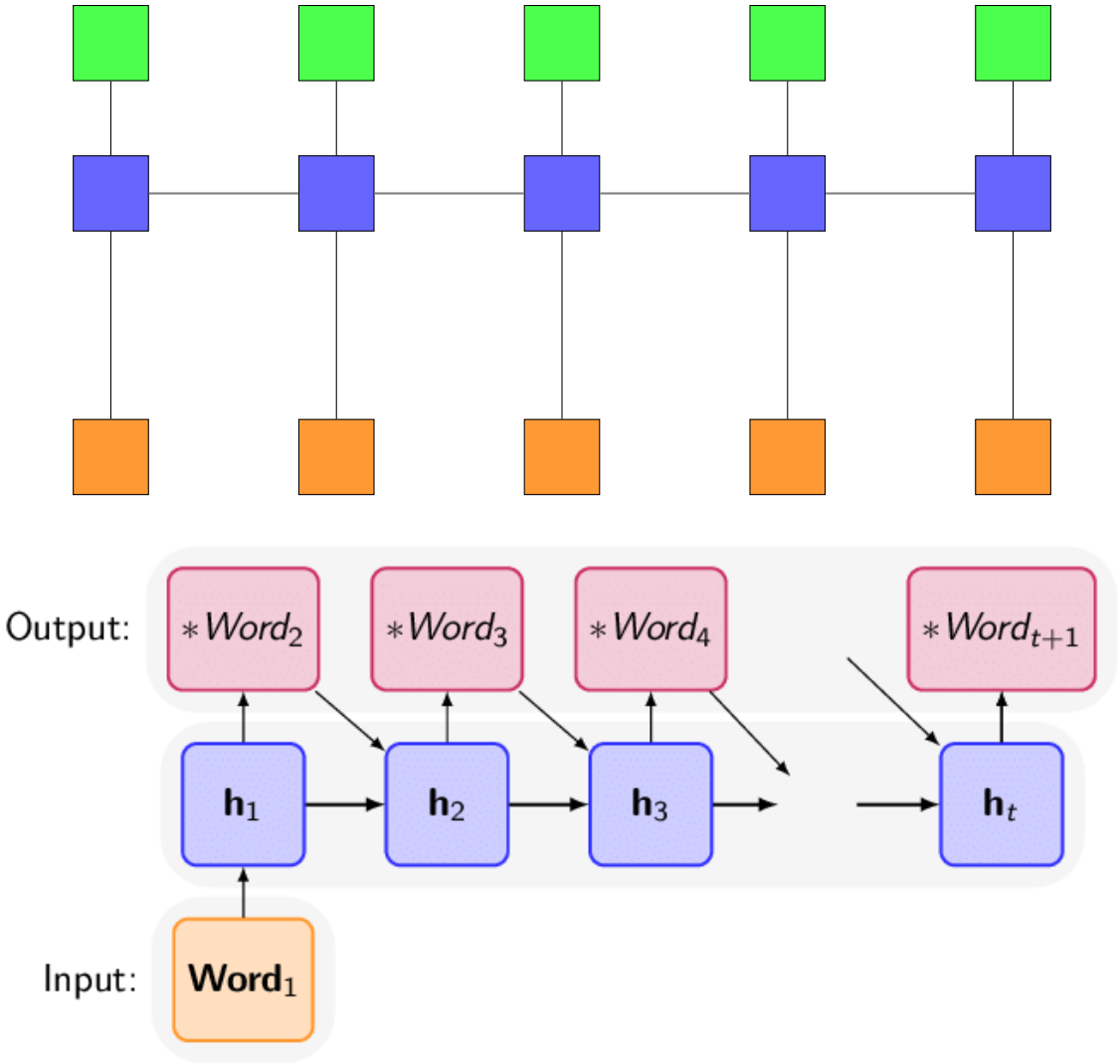


Figure 1:

4 Preliminaries: Introduction to Sequence and Language Modeling

In order to understand, the construction of RNNs, we first consider a basic sequential model. Sequential data arises naturally in numerous domains such as natural language, time series forecasting, speech recognition, and bioinformatics. Modelling such data requires tools that can capture the order and dependencies inherent in sequences. This section introduces the fundamental concepts and probabilistic formulation underlying sequence and language modeling.

4.1 Sequence Modeling

Let $\mathbf{x} = (x_1, x_2, \ldots, x_T)$ denote a sequence of random variables over a finite vocabulary $\mathcal{V}$. The goal of sequence modeling is to learn the joint distribution $P(\mathbf{x}) = P(x_1, x_2, \ldots, x_T)$.

[Autoregressive Factorization] Using the chain rule of probability, the joint distribution can be factorized as:

$$P(x_1, x_2, \ldots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t \mid x_1, \ldots, x_{t-1})$$

Each term $P(x_t \mid x_1, \ldots, x_{t-1})$ represents a conditional distribution over the vocabulary $\mathcal{V}$, making the problem a sequence of multi-class classification tasks.

4.2 Language Modeling as a Special Case

Language modeling refers to estimating the probability distribution over sequences of words (or tokens) in natural language. The objective remains the same: modeling $P(x_1, \ldots, x_T)$ where each $x_t \in \mathcal{V}$ is a word or subword token.

Applications of language models include:

- Computing sentence probabilities.
- Ranking candidate outputs in generation tasks.
- Generating new sequences via sampling.
- Learning contextual embeddings for downstream tasks.

4.3 Markov Models and the Context Window

To reduce model complexity, early statistical models often assumed a fixed-length memory:

[Markov Assumption] A sequence model satisfies the Markov property of order τ if:

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-\tau}, \ldots, x_{t-1})$$

[First-Order Markov Model] When $\tau = 1$, the factorization becomes:

$$P(x_1, \ldots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t \mid x_{t-1})$$

While this assumption reduces computational complexity, it limits the model's ability to capture long-range dependencies, which are crucial in natural language (e.g., subject-verb agreement across clauses).

4.4 Limitations and Motivation for Neural Models

Markov models face several trade-offs:

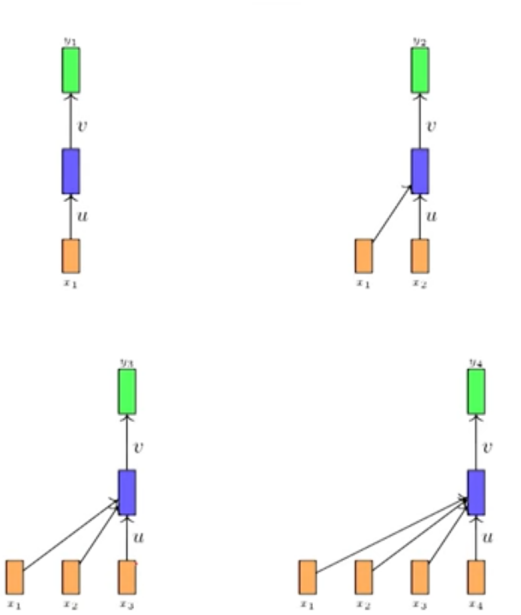
- Higher-order models ($\tau > 1$) better capture context but require exponentially more parameters and data.
- Real-world data often violates the Markov assumption due to long-term dependencies.

To overcome these limitations, neural architectures such as Recurrent Neural Networks (RNNs) were introduced. These models:

- Reuse parameters across time steps.
- Maintain a hidden state to capture history.
- Handle sequences of arbitrary length.

RNNs thus provide a framework for approximating $P(x_t \mid x_1, \ldots, x_{t-1})$ in a more scalable and expressive manner.

4.5 Example of what we don't want in an RNN



- First, the function being computed at each time-step now is different

$$y_1 = f_1(x_1)$$
$$y_2 = f_2(x_1, x_2)$$
$$y_3 = f_3(x_1, x_2, x_3)$$

- The network is now sensitive to the length of the sequence
- For example a sequence of length 10 will require $f_1, \ldots, f_{10}$ whereas a sequence of length 100 will require $f_1, \ldots, f_{100}$

Figure 2: Enter Caption

5 Recurrent Neural Network (RNN) Architecture

In this section we use the ideas and motivations ftom the earlier section (Preliminaries: Introduction to Sequence and Language Modeling) and extend it to the construction of RNNs

5.1 1. Model Structure

Let the input at time step t be $\mathbf{x}^{(t)} \in \mathbb{R}^n$, hidden state $\mathbf{h}^{(t)} \in \mathbb{R}^m$, and output $\mathbf{y}^{(t)} \in \mathbb{R}^k$. The RNN evolves according to the following recurrence:

$$\mathbf{a}^{(t)} = \mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h \tag{1}$$
$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)}) \tag{2}$$
$$\mathbf{o}^{(t)} = \mathbf{W}_{yh}\mathbf{h}^{(t)} + \mathbf{b}_y \tag{3}$$
$$\mathbf{y}^{(t)} = \text{softmax}(\mathbf{o}^{(t)}) \tag{4}$$

Parameters:

- $\mathbf{W}_{hx} \in \mathbb{R}^{m \times n}$: input-to-hidden weights
- $\mathbf{W}_{hh} \in \mathbb{R}^{m \times m}$: hidden-to-hidden (recurrent) weights
- $\mathbf{W}_{yh} \in \mathbb{R}^{k \times m}$: hidden-to-output weights
- $\mathbf{b}_h \in \mathbb{R}^m$, $\mathbf{b}_y \in \mathbb{R}^k$: bias terms

5.2 2. Loss Function

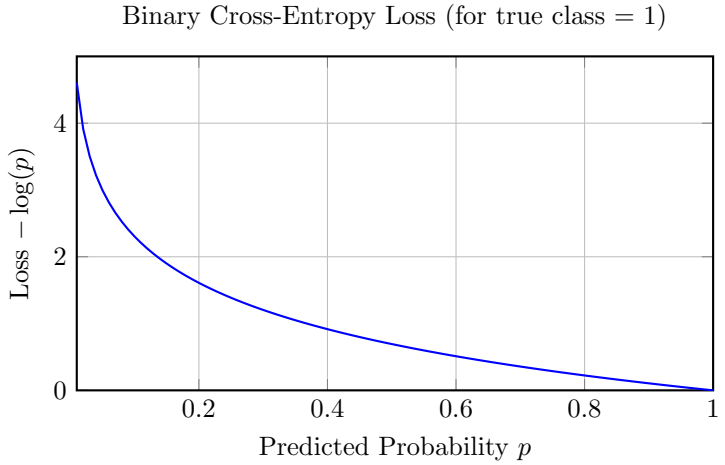
In every deep learning model that is based on *supervised learning* (**Supervised learning**: Data and its labels are given, the classifier/ model must map the inputs to the right labels. In this case, the input words till time n-1 is given to predict the word at place n in the sentence.)

Given a target sequence $\{c^{(t)}\}_{t=1}^{T}$, where each $c^{(t)}$ is the index of the correct class at time t, the total cross-entropy loss is:

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}^{(t)}, \quad \mathcal{L}^{(t)} = -\log y_{c^{(t)}}^{(t)} \tag{5}$$

Rationale: Cross-entropy loss is widely used in classification problems because it measures the dissimilarity between the predicted probability distribution $y^{(t)}$ and the true distribution (which is a one-hot vector with 1 at $c^{(t)}$). Intuitively, it penalizes the model more when it assigns low probability to the correct class. In the context of RNNs, where each output is a probability distribution over the vocabulary at each time step, cross-entropy effectively guides the model to increase the likelihood of the correct next word. This aligns perfectly with the goal of sequence modeling in language tasks: to predict the most probable next token.

Graphical Intuition:



Interpretation: The binary cross-entropy loss $-\log(p)$ increases rapidly as the predicted probability p of the true class (label = 1) decreases. When the model is *confident and correct* (i.e., $p \approx 1$), the loss is near zero. However, when the model is *uncertain* or assigns a low probability to the correct class, the loss increases sharply. This reflects the idea of **surprisal** in information theory—unexpected or surprising outcomes carry more information and thus incur higher penalty. Cross-entropy captures this by heavily penalizing confident but incorrect predictions, encouraging the model to be both accurate and well-calibrated in its uncertainty.

6 Backpropagation Through Time (BPTT)

In order to optimize and train the model to learn the sequences and predict the next value on unseen data, the model, must minimize the loss function on the given training dat first. This minimization occurs through an optimization technique call gradient descent (explained in last report).
Gradient descent requires the information about the gradients after each iteration of forward propagation discussed earlier.
This section discussed how the gradients are calculated in an RNN.

6.1 Local Gradients

We define the local gradient at time t as:

$$\delta^{(t)} := \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(t)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(t)}} \circ \left(1 - \tanh^2(\mathbf{a}^{(t)})\right) \tag{6}$$

6.2 Gradients w.r.t Parameters

The gradients for each parameter matrix are:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hx}} = \sum_{t=1}^{T} \delta^{(t)} {\mathbf{x}^{(t)}}^{\top} \tag{7}$$

$$\frac{\partial \mathrm{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \delta^{(t)} {\mathbf{h}^{(t-1)}}^{\top} \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{yh}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}^{(t)}}{\partial \mathbf{o}^{(t)}} {\mathbf{h}^{(t)}}^{\top} \tag{9}$$

6.3 Recursive Gradient Flow

Due to the recurrence, $\mathbf{h}^{(t)}$ depends on all previous states. Therefore:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(t-1)}} = \left(\mathbf{W}_{hh}^{\top} \delta^{(t)}\right) \circ \left(1 - \tanh^2(\mathbf{a}^{(t-1)})\right) \tag{10}$$

This recursive structure creates a computational graph that is "unrolled" over time to apply standard backpropagation.

7 Gradient Accumulation Across Time

7.1 Unrolling the Gradient for $\mathbf{W}_{hh}$

The influence of $\mathbf{W}_{hh}$ at each step involves contributions from multiple paths:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \left(\left(\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} \right) \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right) \tag{11}$$

7.2 General Gradient Structure

More generally, for any shared parameter W (e.g., $\mathbf{W}_{hh}$), the total gradient is:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W} = \frac{\partial \mathcal{L}^{(t)}}{\partial \mathbf{h}^{(t)}} \sum_{k=1}^{t} \left(\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial W} \right) \tag{12}$$

8 Gradient Descent Algorithm for RNNs

After the forward propagation, gradient calculation, the model is finally trained by minimiztioning the loss function using the gradient descent algorithm for many iterations

Given the gradients computed above, the parameters of the RNN are updated using gradient descent. Let η denote the learning rate. The update rules for the parameter matrices are:

$$\mathbf{W}_{hx} \leftarrow \mathbf{W}_{hx} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hx}} \tag{13}$$

$$\mathbf{W}_{hh} \leftarrow \mathbf{W}_{hh} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} \tag{14}$$

$$\mathbf{W}_{yh} \leftarrow \mathbf{W}_{yh} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}_{yh}} \tag{15}$$

This procedure is repeated iteratively after computing the loss and corresponding gradients over the entire sequence. When training on mini-batches or full datasets, the gradients are averaged across the batch before applying the update. These updates aim to minimize the total loss $\mathcal{L}$ by adjusting the parameters in the direction of steepest descent.

Note that due to the recursive dependency of the hidden states, backpropagation through time (BPTT) is used to compute these gradients efficiently by unrolling the RNN and applying the chain rule over the time steps.

Now, in the next section we discuss a major problem faced while training an RNN.

9 Interpretation and Challenges

After the gradient was calculated and optimized using the gradient descent algorithms, the following problem is faced during training sequential models like RNNs

9.1 Vanishing/Exploding Gradients

The term $\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}}$ may become very small (vanishing gradient) or very large (exploding gradient), due to repeated multiplication of Jacobians. This affects the ability of the RNN to learn long-term dependencies.

9.2 The Vanishing Gradient Problem in RNNs

The **vanishing gradient problem** arises during the training of deep or recurrent neural networks when gradients propagated through many layers or time steps tend to diminish exponentially, leading to extremely small gradient values. This effectively prevents the weights in earlier layers (or earlier time steps in RNNs) from being updated meaningfully, thereby limiting the model's ability to learn long-range dependencies.

Let us recall the gradient of the loss with respect to the RNN parameters W_h, unrolled over time:

$$\frac{\partial \mathcal{L}}{\partial W_h} = \frac{1}{T} \sum_{t=1}^{T} \delta_t \sum_{i=1}^{t} \left(\left(\prod_{j=i+1}^{t} J_j \right) \cdot \frac{\partial f(x_i, h_{i-1})}{\partial W_h} \right), \tag{16}$$

where $J_j = \frac{\partial f(x_j, h_{j-1})}{\partial h_{j-1}}$ is the Jacobian matrix that encodes how the hidden state at time step j changes with respect to the previous state.

Let us consider a simplified case where f is a pointwise nonlinear activation function such as tanh or $\sigma(\cdot)$, and the hidden state update is:

$$h_t = \phi(W_h h_{t-1} + W_x x_t), \tag{17}$$

where ϕ is a smooth activation function and W_h is recurrent weight matrix.

9.2.1 Unrolled Gradient Expression

In this setup, the Jacobian J_j can be written as:

$$J_j = \operatorname{diag}\left(\phi'(a_j)\right) \cdot W_h, \tag{18}$$

where $a_j = W_h h_{j-1} + W_x x_j$ and ϕ' denotes the derivative of the activation function. The product of Jacobians across time steps becomes:

$$\prod_{j=i+1}^{t} J_j = J_t J_{t-1} \cdots J_{i+1}. \tag{19}$$

This is the core component responsible for either vanishing or exploding gradients.

9.2.2 Exponential Decay of Gradient Norms

Suppose the spectral norm of W_h is $\rho = \|W_h\|_2$ and the activation function derivative satisfies $|\phi'(\cdot)| \leq \gamma < 1$ for all inputs. Then the norm of the Jacobian product satisfies:

$$\left\| \prod_{j=i+1}^{t} J_j \right\| \leq (\gamma\rho)^{t-i}. \tag{20}$$

Hence, if $\gamma\rho < 1$, the gradient norm decays exponentially with respect to the number of time steps $(t-i)$:

$$\left\| \frac{\partial h_t}{\partial h_i} \right\| \to 0 \quad \text{as } t - i \to \infty. \tag{21}$$

This implies that the influence of early time steps on the loss function diminishes rapidly as the time gap increases. Consequently, the model struggles to capture long-term dependencies in sequences.

9.2.3 Implications

- **Poor Learning of Long-Term Dependencies:** The vanishing gradient causes the model to "forget" earlier time steps when learning.
- **Slow Convergence:** Gradients with very small magnitudes lead to minuscule weight updates, making training inefficient.
- **Bias Toward Recent Inputs:** The network implicitly focuses more on recent inputs since distant ones exert negligible influence on the current state.

9.2.4 Contrast With Exploding Gradients

In contrast, if $\gamma\rho > 1$, the gradient norm can grow exponentially, resulting in **exploding gradients**. This leads to numerical instability and causes the loss to oscillate or diverge. Techniques such as *gradient clipping* are commonly used to address this.

9.2.5 Remedies

To address vanishing gradients:

- Use specialized architectures like **LSTM** or **GRU**, which introduce gating mechanisms that enable better gradient flow.
- Employ **ReLU**-like activations (though they bring other challenges).
- Apply **gradient clipping** to stabilize training.
- Use **shorter sequences** or **truncated BPTT** to limit the length of backpropagation.
- Consider using **residual connections** in deeper RNN stacks.

Model	Sequential Input	Memory	Gradient Issues
MLP	No	No	No
RNN	Yes	Yes	Yes (vanishing/exploding)
LSTM	Yes	Long memory	No
N-gram	Yes (limited)	Fixed-length	No

10 Transition Towards Neural Temporal Point Processes (TPPs)

Having explored the structure and training dynamics of Recurrent Neural Networks (RNNs), especially the challenges such as the vanishing gradient problem, we now move toward a more flexible and expressive class of models: **Neural Temporal Point Processes (TPPs)**.

RNNs are powerful in modeling sequences of data in *discrete time* steps, but they are not naturally equipped to handle sequences where events occur *irregularly over continuous time*. Many real-world phenomena—such as financial transactions, healthcare events, or user interactions—are inherently asynchronous and are better modeled in continuous time.

Neural Temporal Point Processes combine classical *temporal point processes* with the representational power of neural networks to learn complex temporal patterns in event data. This shift aligns with the broader goal of this report: modeling stochastic, time-dependent behaviors with both temporal precision and representational flexibility.

How It Connects to my Study

- The study of RNNs and their gradient behavior has laid the mathematical and computational groundwork.
- Understanding backpropagation through time (BPTT) and gradient flow is essential before analyzing continuous-time gradient dynamics in TPPs.
- TPPs extend this knowledge to a more general setting: we move from modeling hidden states $\{h_t\}$ at discrete steps to modeling event times $\{t_k\}$ and their distributions directly.
- This transition allows us to analyze sequences not only by their content but by the timing of their occurrence—adding an essential temporal dimension to our learning framework.

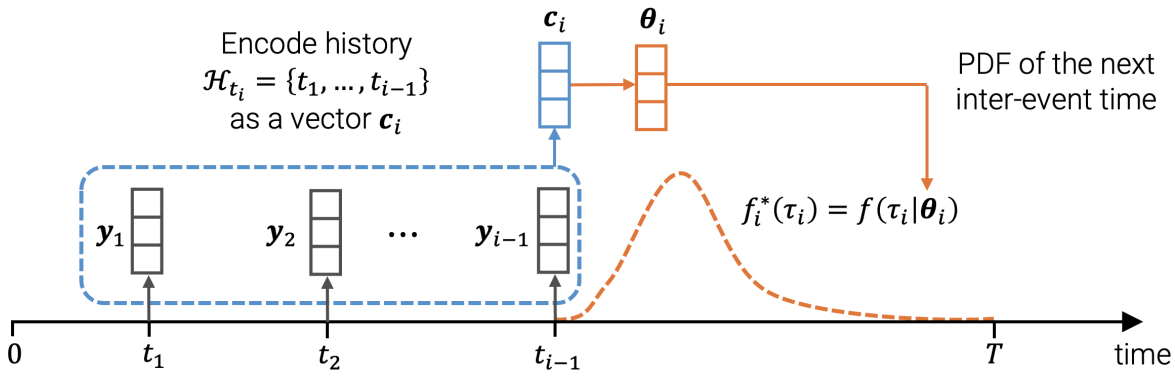


Figure 3: Enter Caption

References

[1] Oleksandr Shchur, Ali Caner Türkmen, Tim Januschowski, Stephan Günnemann *Neural Temporal Point Processes: A Review*. arXiv preprint arXiv:2104.03528, 2021. `https://arxiv.org/abs/2104.03528`.

[2] A. Zhang, M. Li, Z. Lipton, and A. J. Smola. *Dive into Deep Learning*. Publisher : Cambridge University Press; 1st edition (December 7, 2023), Chapter 5. `https://d2l.ai/`.