Investigating Practical Limits of Storing and Recalling Sequential Inputs in RNNs and factors contributing to it

Nirmal Eswar Vee

Supervisor: Dr Anton Ragni

A report submitted in fulfilment of the requirements for the degree of BSc in Computer Science (AI)

in the

School of Computer Science

May 16, 2025

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Nirmal Eswar Vee

Signature: E.Nirmal

Date: 17/10/2024

Abstract

Storing and recalling long input sequences is critical in domains such as natural language processing and time-series forecasting. Recurrent Neural Networks (RNNs) model sequences of data by carrying a state forward in time and, in theory, offer universal sequence modelling. However, literature claims that they suffer from vanishing or exploding gradients that limit their practical storing and recall of information. Common solutions—gating [24] and attention [48]—mitigate but do not eliminate this limitation, and the fundamental recall capacity of standard ungated RNNs remains unclear. This dissertation revisits the standard RNN architecture—without adding gates or attention—and shows that by restructuring its weight matrices into sparse, block-shifting form with an invertible activation, one obtains an RNN that can exactly reverse each update and perfectly recall sequences up to length 10^5. This directly refutes earlier practical limits of ten steps [6] or a few hundred steps with normalisation [43]. This demonstrates that no new model class is needed: a standard RNN suffices when its recurrence is carefully parameterised. A systematic sweep over sequence length, input class size, state size, activation function, and initialisation strategy quantifies how much information from early inputs survives each update. These findings demonstrate that, with careful architectural design, standard ungated RNNs can preserve recall accuracy over long sequences without complex mechanisms, offering a simpler, more interpretable alternative to existing sequence models.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the project and summarises the paper.

1.1 Background

Storing and recalling early inputs over long sequences is critical in tasks such as natural language processing, time-series prediction, and speech recognition [1]. Recurrent Neural Networks (RNNs)—which process one input at a time and carry a state forward through time—are, in principle, universal function approximators for sequential data [14]. However, literature claims that without specialised architectures, gradients can vanish or explode, limiting how far back in time they can recall [25]. This has led to more computationally expensive architecture with more complex mechanisms (gating and attention) being used over the traditional RNN in recent times. Yet, architectural factors have been shown to mitigate but not completely eliminate this limitation [24, 4, 48]. Even when gradients are stabilised, the need to squash a growing history of states into a fixed-dimensional vector forces a lossy compression. Recent analyses of recurrent networks also pinpoint compression as the core bottleneck in recurrent networks [20]. As each new input is processed, the network must overwrite older representations to make space, leading to irreversible information loss (see section **??**). This leads to the core question of this dissertation: could a carefully structured, standard, ungated RNN with no attention mechanisms still achieve perfect recall over long sequences if it had enough state space for all the sequence information? Surprisingly, this question remains underexplored despite the theoretical universality of RNNs. In the next section, a proposed approach to this question is outlined.

1.2 Proposed Approach

In the foundational 1994 study by Bengio et al. (1994) [6], it was argued that "RNNs unfolded in time fail to learn inter-temporal influences more than ten time steps apart" [6]. Subsequent work by Schäfer and Zimmermann (2006) [43] showed that normalised RNNs increase this limit to a hundred steps. Further advances—such as careful weight initialisation [27], gradient clipping [35], orthogonal or unitary recurrent matrices [4], and the introduction of gating [24, 8] (all explained in chapter 2)—have pushed these practical limits into the hundreds or thousands of time steps. This paper proposes structuring the RNN to have a state scaled by the max sequence length and structured parameters that allow all information from all time steps to slot into the state. By applying an inversion of the chosen activation function, the architecture achieves exact recovery of past states—eliminating the gradual information decay inherent in traditional RNNs and enabling reliable recall across arbitrarily long sequences, far surpassing the ten- to hundred-step limits established in previous literature.

Note: **The RNN (state scaled with structured parameters) is not a new architecture – it is a standard RNN that works under the assumptions of a state scaled proportionately to the sequence length and having appropriately structured parameters (see subsection 3.3.4). It is also important to note that the solution described does not require training; it is algorithmic rather than a learnt solution (see section 3.3).**

1.2.1 Contributions

The RNN (state scaled with structured parameters) delivers exact reconstruction up to sequence lengths of 10^5, thereby directly refuting both the ten-step and hundred-step ceilings reported in the literature. By measuring the reconstruction error of both the state and the embeddings at each timestep, the amount of information from early inputs that survives each recurrent update can be precisely quantified. For full results see Chapter 5: Results.

1.3 Aims and Objectives

This dissertation aims to address the gap in understanding in contemporary literature by investigating the practical limits and factors of the RNN (state scaled with structured parameters):

- **Sequence length**
- **Input class size**
- **State dimensionality**
- **Activation invertibility**
- **Weight initialisation strategies**
- **Reconstruction (inversion) depth**
- **Semantic-clustering performance**

A deeper understanding of these factors is essential to fully exploit the universal approximation power of RNNs and to inform the design of future architectures [52].

1.4 Relevance to Degree

As a student in Computer Science with Artificial Intelligence, this project integrates key concepts from machine learning, neural network architectures, and software engineering. It builds on the knowledge gained in modules such as:

- **COM3524 - Bioinspired Computing:** Understanding and designing neural networks.
- **COM21002 - AI Group Project:** Applying agile methodologies and iterative development processes.
- **COM2004 - Data Driven Computing:** Investigating and implementing advanced machine learning techniques.

This project bridges theoretical research with practical implementation. It also offers the chance to deepen my knowledge in the field of neural networks beyond the scope of my degree.

1.5 Overview of the Report

This report is structured as follows:

- **Chapter 2: Literature Survey**
  Provides a detailed review of relevant literature, covering the challenges of storing and recalling long sequences, the vanishing gradient problem, and various RNN architectures (including LSTMs and GRUs).
- **Chapter 3: Analysis**
  Outlines the theoretical framework and proof of concept, explaining the proposed structuring of standard RNNs to achieve perfect recall over long sequences and the rationale behind it.
- **Chapter 4: Implementation**
  Details the implementation of the proof of concept into Python code using the PyTorch framework and describes specific design choices.
- **Chapter 5: Results**
  Presents parameter choices and experimental results with a brief discussion on their impact for each experiment.
- **Chapter 6: Conclusion**
  Summarises the key findings and discusses their implications in more depth than chapter 5. Furthermore, limitations of the study are highlighted, as well as future work that can be done.
- **Appendix: Planning**
  Describes the design process and experimental plan from the first semester of doing the project, including the schedule used for meetings and the timeline initially planned for the project.

Chapter 2

Literature Survey

This chapter serves as a database of knowledge needed to fully understand the rest of the paper.

2.1 Notational conventions

Throughout this dissertation the following notational conventions are used:

- **Scalars** are set in non-bold font: e.g. x, L, D, d.
- **Vectors** are bold lowercase: e.g. $\boldsymbol{x}$, $\boldsymbol{h}_t$, $\boldsymbol{e}_k$.
- **Matrices** are bold uppercase: e.g. $\boldsymbol{W}^h$, $\boldsymbol{W}^x$.
- **Calligraphic symbols** denote sets or spaces: e.g. $\mathcal{X}$, $\mathcal{T}$.
- **Functions and operators** are non-bold:
  - Activation functions: $\sigma(\cdot)$, $\tanh(\cdot)$, etc.
  - Logarithm: $\log(x)$ for scalars; when applied elementwise to a vector, bold may be used, e.g. $\log \boldsymbol{x}$.
  - Probability: the operator $P(\cdot)$ is never bold.
- **Partial derivatives** use non-bold symbols throughout:

$$\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}.$$

- **Subscripts and superscripts** are always non-bold, even when attached to bold symbols:

$$\boldsymbol{h}_{t-1}, \quad \boldsymbol{W}^h, \quad \sigma^{-1}.$$

- **Boldface** is reserved exclusively for quantities that are inherently vector- or matrix-valued.

2.2 Mathematical Foundations and Assumptions

This section aims to provide an explanation of specific math topics, which will serve as a foundation to help with understanding the rest of the report.

2.2.1 Differential Calculus

Partial derivatives: Partial derivatives measure the rate of change of a function with respect to one variable while keeping the others constant. They are useful for functions with multiple variables [3]. For a function $f(\boldsymbol{x}, \boldsymbol{y})$, the partial derivatives with respect to $\boldsymbol{x}$ and $\boldsymbol{y}$ are denoted as:

$$\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}. \tag{2.1}$$

For example, if:

$$f(x, y) = x^2 + y^2, \tag{2.2}$$

Then:

$$\frac{\partial f}{\partial x} = 2x, \quad \frac{\partial f}{\partial y} = 2y. \tag{2.3}$$

The chain rule: The chain rule is used to link relationships of variables together by using their derivatives. It is used to differentiate composite functions and link the rate of change between the variables [12]. For the composite function:

$$h(x) = f\big(g(x)\big), \tag{2.4}$$

$$\frac{dh}{dx} = \frac{df}{dg}\frac{dg}{dx}. \tag{2.5}$$

Where:

- $\frac{dh}{dx}$ is the rate of change of h with respect to x.
- $\frac{df}{dg}$ is the rate of change of f with respect to g.
- $\frac{dg}{dx}$ is the rate of change of g with respect to x.

2.2.2 Vectors and Matrices

- **Vector**: an ordered one-dimensional array

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}. \tag{2.6}$$

- **Matrix**: an ordered two-dimensional array.

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}. \tag{2.7}$$

- **Matrix-vector form**: compactly expresses a linear transformation as $\boldsymbol{A}\,\boldsymbol{v}$.
- **Square matrix**: a matrix with the same number of rows and columns, i.e. $\boldsymbol{A} \in \mathbb{R}^{n\times n}$.
- **Transpose**:
  - For a vector $\boldsymbol{v} \in \mathbb{R}^{D}$, the transpose $\boldsymbol{v}^\top \in \mathbb{R}^{1\times D}$ is a row vector.
  - For a matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, the transpose $\boldsymbol{A}^\top \in \mathbb{R}^{n\times m}$ is defined by $(\boldsymbol{A}^\top)_{ij} = A_{ji}$.
- **Identity matrix**: the $n \times n$ matrix with ones on the diagonal and zeros elsewhere,

$$\boldsymbol{I}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}, \tag{2.8}$$

  which acts as the multiplicative identity: $\boldsymbol{A}\,\boldsymbol{I}_n = \boldsymbol{I}_n\,\boldsymbol{A} = \boldsymbol{A}$.
- **Inverse matrix**: for a square matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, the inverse $\boldsymbol{A}^{-1}$ satisfies

$$\boldsymbol{A}\,\boldsymbol{A}^{-1} = \boldsymbol{I} \quad \Longleftrightarrow \quad \det(\boldsymbol{A}) \neq 0. \tag{2.9}$$

- **Complex conjugate**: for a complex scalar,

$$z = a + b\,i, \tag{2.10}$$

  Where:
  - a is a real number
  - b is a real number
  - i is a complex number, $i^2 = -1$

  The complex conjugate is: $\overline{z} = a - bi$.
- **Conjugate transpose (Hermitian transpose)**: for $\boldsymbol{U} \in \mathbb{C}^{m\times n}$, the conjugate transpose $\boldsymbol{U}^* \in \mathbb{C}^{n\times m}$ is defined by

$$\boldsymbol{U}^* = \overline{\boldsymbol{U}}^\top, \quad (\boldsymbol{U}^*)_{ij} = \overline{U_{ji}}. \tag{2.11}$$

- **One-hot encoded vectors:** Binary vectors where only one element is set to 1 (hot) and the rest are set to 0 (cold)[9]. It is commonly used to represent different categories in machine learning models. For example, if a variable had 3 different classes, they could be represented as one-hot encoded vectors like this:

$$\boldsymbol{class}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{class}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{class}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{2.12}$$

  The vector length should match the number of classes.

- **Tensor**: a multi-dimensional array that generalises vectors (order 1) and matrices (order 2). For example, a 2×2×2 tensor can be thought of as two 2×2 "slices" stacked together:

$$\mathcal{T} = [\, \underbrace{\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}}_{\mathcal{T}_{:,:,1}}, \underbrace{\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}}_{\mathcal{T}_{:,:,2}} \,] \in \mathbb{R}^{2\times2\times2}. \tag{2.13}$$

  Tensors are the basic data structure in PyTorch for holding multivariable data [17].

2.2.3 Euclidean Norm and Distance

Euclidean distance is used when computing distance-based measures to see how similar vectors are to one another.

- **Euclidean norm:** for a vector $\boldsymbol{v} \in \mathbb{R}^D$,

$$\|\boldsymbol{v}\|_2 = \sqrt{\sum_{i=1}^{D} v_i^2}\,.$$

- **Euclidean distance:** for two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^D$,

$$d(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{u} - \boldsymbol{v}\|_2 = \sqrt{\sum_{i=1}^{D} (u_i - v_i)^2}\,.$$

- **Magnitude of a vector (Euclidean norm)**: represents the vector as a scalar quantity. The length of $\boldsymbol{v}$ measured in Euclidean space,

$$\|\boldsymbol{v}\|_2 = \sqrt{\sum_{i=1}^{D} v_i^2}. \tag{2.14}$$

- **Magnitude of a matrix (Frobenius norm)**: the root-sum-of-squares of all entries of $\boldsymbol{A}$,

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2}, \tag{2.15}$$

  where $\boldsymbol{A} \in \mathbb{R}^{m\times n}$.

2.2.4 Probabilities and Log-Probabilities

- **Probability**: the likelihood of an event x, denoted

$$P(x), \quad 0 \le P(x) \le 1. \tag{2.16}$$

- **Joint probability**: for a sequence $\{x_t\}_{t=1}^{L}$,

$$P(x_1, x_2, \ldots, x_L). \tag{2.17}$$

- **Conditional probability**: the likelihood of x_t given all previous events,

$$P\big(x_t \mid x_{<t}\big). \tag{2.18}$$

- **Log-Probability:** For a probability $P(x)$, the log-probability is given by:

$$\log(P(x)). \tag{2.19}$$

  This converts products into sums:

$$\log P(x_1, \ldots, x_L) = \sum_{t=1}^{L} \log P\big(x_t \mid x_{<t}\big). \tag{2.20}$$

2.2.5 Statistical Measures

Summaries of measurements that are used throughout the dissertation to measure the performance of model results.

- **Mean** (μ): the average of N values $\{x_i\}$,

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i. \tag{2.21}$$

- **Variance** (σ^2): the average squared deviation from the mean,

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2. \tag{2.22}$$

- **Standard deviation** (σ): the square root of the variance,

$$\sigma = \sqrt{\sigma^2}. \tag{2.23}$$

- **Mean Squared Error (MSE)**: given true values $\{y_i\}$ and estimates $\{\hat{y}_i\}$,

$$\mathrm{MSE} = \frac{1}{N} \sum_{i=1}^{N} \big(\hat{y}_i - y_i\big)^2. \tag{2.24}$$

These measures provide insight into the average performance (mean), variability (variance, standard deviation), and reconstruction error (MSE) of models and are standard tools in statistical analysis and machine learning [17].

2.3 Neural Networks (NNs)

Artifical Neural Networks, which are more commonly known as ”Neural Networks” (NN) were inspired by the structure of the human brain. NNs have the ability to produce outputs for inputs they haven’t encountered during training. This ability allows them to be able to approximate complex functional relationships [21], which makes them very useful in fields such as speech and text processing. There are many types of neural networks; see [22].

2.3.1 Architecture

Neural networks (NN) consist of layers of interconnected nodes, or ”neurones”, that process the input. For each node, there are parameters that modify the input. These parameters include weights and biases; each node only has one output. Each layer generates a vector of intermediate outputs.

$$\mathbf{z} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b} \tag{2.25}$$

where:

- $\boldsymbol{z}$ is the set of intermediate outputs,
- $\boldsymbol{W}$ is the weight matrix for the layer,
- $\boldsymbol{x}$ is the input vector,
- $\boldsymbol{b}$ is the bias vector.

From this equation, we can see that the layers transform multidimensional input data into multidimensional output data. The dimensions of the input and output data don’t strictly need to be the same [17].

2.3.2 Activation Functions

An activation function in NNs is a function applied to the output of each node in a layer, determining whether that node should be ’activated’ or not. If a node is ’activated,’ it contributes to the output of the network as a whole [31]. The activation of a node is based on the weighted sum of its inputs. The weighted sum ($\boldsymbol{z}$) can be expressed as:

$$\boldsymbol{z} = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b \tag{2.26}$$

where:

- $x_1, x_2, \ldots, x_n$ are the inputs to the neuron,
- $w_1, w_2, \ldots, w_n$ are the weights associated with these inputs,
- b is the bias term.

According to [16], common activation functions include:

- **Sigmoid** ($\sigma(z) = 1/(1 + e^{-z})$): maps its input to the interval $(0, 1)$.

- **Tanh** ($\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$): maps its input to $(-1, 1)$.
- **ReLU (Rectified Linear Unit)** ($\mathrm{ReLU}(z) = \max(0, z)$): passes positive inputs unchanged, zeroes out negatives.
- **Leaky ReLU** ($\mathrm{LeakyReLU}(z) = \max(\alpha z,\, z)$ with small $\alpha > 0$): allows a small, nonzero gradient when $z < 0$.
- **ELU (Exponential Linear Unit)** ($\mathrm{ELU}(z) = \begin{cases} z, & z > 0, \\ \alpha\,(e^z - 1), & z \le 0, \end{cases}$): smooths the negative side via an exponential curve.

Activation functions introduce non-linearity into the model so that they can represent non-linear random functional mappings between inputs and outputs [44].

2.3.3 Feedforward Neural Networks (FNN)

Feedforward neural networks (FNNs) are neural networks where data only flows in one direction [13]. The equation for a FNN is:

$$\boldsymbol{a} = f(\boldsymbol{z}) = f\big(\boldsymbol{W}\,\boldsymbol{x} + \boldsymbol{b}\big) \tag{2.27}$$

where:

- $\boldsymbol{a}$ is the output
- $\boldsymbol{W}$ is the weight matrix,
- $\boldsymbol{x}$ is the input vector,
- $\boldsymbol{b}$ is the bias vector,
- f is the activation function,
- $\boldsymbol{z}$ is the set of intermediate outputs.

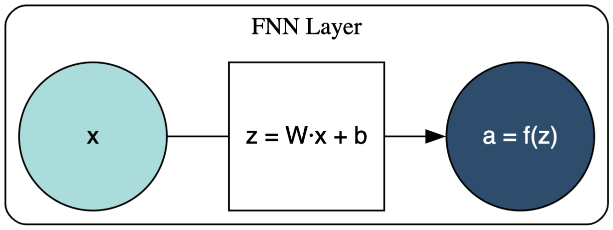


Figure 2.1: A layer of an FNN visualised

The next section will look at how these networks are trained.

2.4 Training

The main goal of training NNs is to learn the optimal set of parameters that minimise the loss function. This allows them to identify patterns and connections in the data. This makes NNs effective for tasks such as classification, regression, and feature extraction [17].

2.4.1 Loss Functions:

A loss function $L(\boldsymbol{\theta})$ measures the loss. The loss is the difference between the output at the final layer ($\boldsymbol{z}$) and the expected value [29]. Here, 'difference' refers to any method of quantifying the discrepancy between the predicted and expected outputs, rather than implying that it's only ever simple arithmetic subtraction. The gradient of the loss function, $\nabla L(\boldsymbol{\theta})$, is a vector of partial derivatives that points in the direction of the steepest increase in $L(\boldsymbol{\theta})$.

$$\nabla L(\boldsymbol{\theta}) = \left[\frac{\partial L}{\partial \boldsymbol{\theta}_1}, \frac{\partial L}{\partial \boldsymbol{\theta}_2}, \ldots, \frac{\partial L}{\partial \boldsymbol{\theta}_n}\right]^\top \tag{2.28}$$

Where:

- $\boldsymbol{\theta} = [\theta_1, \theta_2, \ldots, \theta_n]^\top$ are the parameters.

There are many different types of loss functions used in deep learning for different tasks; see [49].

2.4.2 Gradient Descent

Gradient descent is an optimisation algorithm that iteratively updates parameters in the direction that most reduces the loss [26]. The update rule is

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha\, \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t), \tag{2.29}$$

where:

- $\boldsymbol{\theta}_t$ is the parameter vector at iteration t,
- α is the (scalar) learning rate,
- $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t)$ is the gradient of the scalar loss L with respect to the vector $\boldsymbol{\theta}$.

The learning rate α plays a critical role in gradient descent. If α is too large, the algorithm may overshoot the minimum and fail to converge. If α is too small, convergence will be slow. There exist many different algorithms that dynamically adjust the learning rate, like Adam and RMSprop [17].

2.4.3 Backpropagation

Backpropagation is a well-known algorithm used widely in the neural network field that calculates the gradient of the loss function with respect to each parameter using the chain rule. It propagates errors backwards through the network to minimise the loss by updating the weights via gradient descent [40].

The process:

1. **Forward Pass:** Compute the predicted output $\boldsymbol{y}$ by passing the input $\boldsymbol{x}$ through the network. Then calculate the loss $L(\boldsymbol{\theta})$.

2. **Backward Pass:** Propagate the error backward through the network to compute the gradients:

   At the output layer:
   $$\boldsymbol{\delta}^{(L)} = \frac{\partial L}{\partial \boldsymbol{y}} \cdot \sigma'\big(\boldsymbol{z}^{(L)}\big) \tag{2.30}$$
   where:

   - $\boldsymbol{z}^{(L)}$ is the pre-activation vector at the output layer.
   - σ' is the derivative of the activation function.

   At middle layers:
   $$\boldsymbol{\delta}^{(l)} = \big(\boldsymbol{W}^{(l+1)}\big)^{\top} \boldsymbol{\delta}^{(l+1)} \odot \sigma'\big(\boldsymbol{z}^{(l)}\big) \tag{2.31}$$
   where:

   - $\boldsymbol{\delta}^{(l)}$ is the error vector for layer l.
   - $\boldsymbol{W}^{(l+1)}$ is the weight matrix connecting layer l to layer $l+1$.
   - $\odot$ is the element-wise multiplication operator.

2.5 Evaluation Metrics

Training is typically stopped when validation loss or accuracy no longer improves significantly. Model performance is then assessed using metrics such as precision, recall, and F1-score for classification tasks [18], and clustering metrics like the silhouette score. These metrics will reappear in Chapters 4 and 5.

2.5.1 Common Training and Evaluation Metrics

- **Precision (P):** measures the proportion of correctly predicted positive instances out of all predicted positive instances:
  $$P = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.32}$$
  Where:

  - TP = True positive
  - FP = False positive
  - TN = True Negative
  - FN = False Negative

- **Recall (R):** measures the proportion of correctly predicted positive instances out of all actual positive instances:
  $$R = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.33}$$

- **F1-Score (F_1):** is the harmonic mean of precision and recall:

$$F_1 = 2\,\frac{P\,R}{P+R} \tag{2.34}$$

  Where:

  - P = Precision
  - R = Recall

- **Silhouette Score:** Clustering is the grouping of data points so that those in the same group are more similar to each other than to those in other groups. The silhouette score quantifies this quality: for each sample i,

$$s(i) \;=\; \frac{b(i) \;-\; a(i)}{\max\{\,a(i),\, b(i)\}}, \tag{2.35}$$

  where $a(i)$ is the average distance from i to all other points in its own cluster, and $b(i)$ is the minimum average distance from i to points in any other cluster. Scores range from -1 to $+1$:

  - $s \approx +1$: point is well matched to its own cluster and poorly matched to neighbouring clusters (ideal).
  - $s \approx 0$: point lies between clusters (ambiguous assignment).
  - $s \approx -1$: point may have been assigned to the wrong cluster.

Other scores can be used to reflect performance depending on the specific implementation.

2.6 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a type of neural network that are used to discover patterns in sequential data where positional or temporal context matters. In contrast to FNNs, which process inputs independently and don't have the structure to capture patterns in the data relating to time, RNNs use cycles to relay information back to themselves. Because of this, RNNs can model sequences where temporal context matters, as each state is influenced by previous states [41].

In this section, we discuss the architecture of RNNs, the process of unrolling, and the challenges they face, particularly with how they store and recall inputs.

2.6.1 Architecture

RNNs are able to model temporal dependencies by having a state that evolves over time. This evolution is controlled by the following state transition equation:

$$\boldsymbol{h}_t = f\big(\boldsymbol{W}_h\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_x\,\boldsymbol{x}_t + \boldsymbol{b}_h + \boldsymbol{b}_x\big) \tag{2.36}$$

Where:

- $\boldsymbol{h}_t$ is the state at time t,

- $\boldsymbol{W}_h$ is the weight matrix that connects $\boldsymbol{h}_{t-1}$ to $\boldsymbol{h}_t$,
- $\boldsymbol{W}_x$ is the weight matrix for the input $\boldsymbol{x}_t$,
- f is the element-wise activation function,
- $\boldsymbol{b}_h$ is the bias vector for the state,
- $\boldsymbol{b}_x$ is the bias vector for the input.

Unlike FNNs, RNNs use both feedforward and recurrent layers. Recurrent layers form a feedback loop $\boldsymbol{W}_h\boldsymbol{h}_{t-1}$, allowing the network to incorporate information from both past and current inputs.

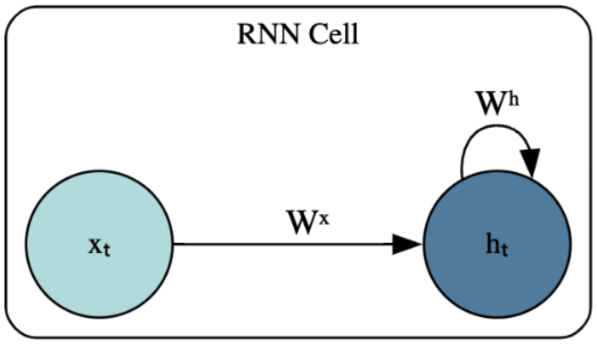


Figure 2.2: Feedback loop visualised

The processed output of the RNN at each time step is computed from the current state using the following equation:

$$\boldsymbol{y}_t = \boldsymbol{W}_y\,\boldsymbol{h}_t + \boldsymbol{b}_y \tag{2.37}$$

Where:

- $\boldsymbol{y}_t$ is the output at time t,
- $\boldsymbol{W}_y$ is the weight matrix mapping $\boldsymbol{h}_t$ to the output space,
- $\boldsymbol{b}_y$ is the bias vector for the output.

This combination of recurrent and feedforward layers allows RNNs to maintain and process information across time steps, making them particularly suited for tasks involving sequential data where positional and time context matters [7]. Furthermore, making them a good choice for tasks involving storing and recalling inputs in sequential data.

2.6.2 Unrolling

RNNs reuse the same parameters across all their time steps to contribute to the final output. Unrolling helps visualise this process by representing the recurrent feedback loop $\boldsymbol{W}_h\boldsymbol{h}_{t-1}$ explicitly for each time step [30]. In an unrolled RNN, the computations for each time step can be visualised as sequential steps in a chain.

For instance:

- At $t = 1$, the network computes the state $\boldsymbol{h}_1$ based on the input x_1 and an initial state $\boldsymbol{h}_0$.
- At $t = 2$, the same parameters $(\boldsymbol{W}_h, \boldsymbol{W}_x, \boldsymbol{b}_h, \boldsymbol{b}_x)$ are reused to compute $\boldsymbol{h}_2$, based on x_2 and $\boldsymbol{h}_1$.
- This process continues for all subsequent time steps.

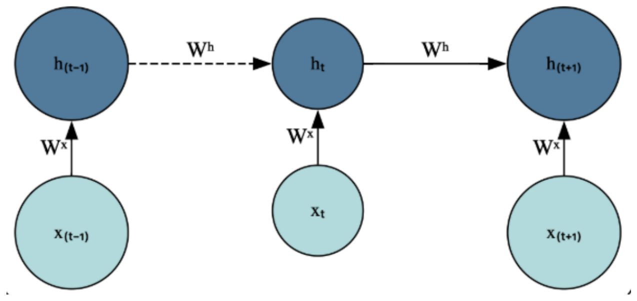


Figure 2.3: Figure 2.2 unrolled over time

Unrolling changes the cyclic nature of an RNN into a series of operations to get the output, making it easier to understand how the network processes input over time [7]. Unrolling not only provides a clear representation of how the network processes sequential data but also lays the foundation for training RNNs using Backpropagation Through Time (BPTT), as gradients can be computed over the unrolled sequence.

Backpropagation Through Time (BPTT)

BPTT is an extension of backpropagation that adjusts the RNN's shared parameters across all time steps by calculating gradients for the entire sequence. The algorithm enables RNNs to learn the underlying temporal context of the sequence by optimising the shared parameters over the entire sequence [50].

During BPTT:

- The loss $L(\boldsymbol{\theta})$ is computed over the entire sequence, typically as a sum of losses at each time step:
$$L = \sum_{t=1}^{T} L_t \tag{2.38}$$
where:
  - T is the total number of time steps.
- Gradients are propagated backwards through the unrolled sequence, starting from the final time step and moving to the initial time step.
- At each time step t, the gradient of the state $\boldsymbol{h}_t$ depends on the gradients from both the current output and the next time step:
$$\boldsymbol{\delta}_{h_t} = \boldsymbol{\delta}_{y_t} \boldsymbol{W}_h + \boldsymbol{\delta}_{h_{t+1}} \boldsymbol{W}_h \tag{2.39}$$

where:

- $\boldsymbol{\delta}_{h_t}$ is the error gradient with respect to the state at time t,
- $\boldsymbol{\delta}_{y_t}$ is the error gradient with respect to the output $\boldsymbol{y}_t$,
- $\boldsymbol{\delta}_{h_{t+1}}$ is the error gradient propagated from the next time step.

While BPTT enables RNNs to learn the underlying temporal context of the sequence, it also introduces challenges such as the vanishing gradient problem (see Section 2.8.1).

2.7 Storing and Recalling Sequential Inputs in RNNs

This section discusses the theory and practicality documented in literature of how and how well RNNs store and recall inputs in sequences.

2.7.1 Theoretical universality for modelling sequences

Recurrent neural networks (RNNs) maintain a continuous, fixed state size that, in principle, can recall all past inputs. Theoretical results show that RNNs are universal approximators of sequence-to-sequence mappings [14, 45], but practical implementations use finite state dimensions and fixed-precision arithmetic, which introduces a compression bottleneck and inevitable information loss [28]. These limitations will be further discussed in the next section (section 2.8).

2.7.2 Practical Limits of Recall in Literature

Early work by Bengio et al. (1994) linked this loss of information from compression to vanishing/exploding gradients, demonstrating that standard RNNs fail to learn contexts within the sequence beyond roughly ten time steps [6]. Schäfer and Zimmermann (2006) showed that normalising the recurrent weights can extend this limit to around one hundred steps [43]. Further techniques (see section 2.10) in more recent times have pushed practical recall limits into the hundreds or low thousands of steps, but none have fully eliminated the need to overwrite older inputs. The next section investigates the limitations on recall in RNNs.

2.8 Vanishing and Exploding Gradient

This section examines the well-documented limitations (vanishing and exploding gradients) of traditional RNNs in literature and how they hinder the ability of RNNs to store and recall inputs as sequence length grows.

2.8.1 The Vanishing Gradient Problem

The vanishing gradient problem is when contributions to the gradients of the loss function from earlier time steps get very small as they are back-propagated through a neural network. In RNNs, whilst the overall gradient is not small, contributions to the gradient from far away can become negligible when BPTT is used.[25] The gradient of the log probability can be expressed as:

$$\frac{\partial \log P(x_1, \ldots, x_L)}{\partial \boldsymbol{\theta}} = \sum_{t=1}^{L} \frac{\partial \log P(x_t \mid x_{<t})}{\partial \boldsymbol{\theta}}. \tag{2.40}$$

Where:

- $P(x_1, \ldots, x_L)$ is the joint probability of the input sequence,
- x_t is the input at time step t,
- $x_{<t}$ represents all inputs before time step t,
- $\boldsymbol{\theta}$ are the model parameters.

The chain rule is applied as gradients are computed through a sequence of historical states:

$$\frac{\partial \log P(x_t \mid x_{<t})}{\partial \boldsymbol{\theta}} = \sum_{i=1}^{t} \frac{\partial \log P(x_t \mid x_{<t})}{\partial \boldsymbol{h}_i} \cdot \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{\theta}}, \tag{2.41}$$

- $\boldsymbol{h}_i$ is the state at time step i.

If the derivatives of the activation functions are less than one, the cumulative product of these gradients can lead to values approaching zero:

$$\prod_{i=1}^{t} \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{z}_i} < 1 \quad \Rightarrow \quad \frac{\partial \log P(x_1, \ldots, x_L)}{\partial \boldsymbol{\theta}} \to 0, \tag{2.42}$$

- $\frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{z}_i}$ is the derivative of the activation at step i.

This significantly limits the network's ability to preserve sufficient information about inputs from earlier time steps. This directly impacts the model's capacity to recall specific historical states. The vanishing gradient problem is therefore an obstacle to storing and recalling of inputs as sequence length grows.

2.8.2 The Exploding Gradient Problem

Conversely, contributions to the gradient of the loss function can grow excessively large, leading to the exploding gradient problem. If the derivatives of the activation function are greater than one, the cumulative product of these gradients grows exponentially. [23]

$$\prod_{i=1}^{t} \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{z}_i} > 1 \quad \Rightarrow \quad \frac{\partial \log P(x_1, \ldots, x_L)}{\partial \boldsymbol{\theta}} \to \infty. \tag{2.43}$$

Similar to the vanishing gradient problem, exploding gradients hinder the storing and recalling of inputs as sequence length grows, by introducing numerical instability during training. This instability prevents the model from consistently learning patterns or storing historical information, as parameters may diverge or oscillate [36].

2.9 Other Factors that Affect the Storing and Recalling of Inputs

This section outlines limitations that aren't as well documented in literature as the vanishing and exploding gradient problems is when discussing the recall limits of RNN models.

2.9.1 Factors Underlying Recall Failure

The literature identifies several interrelated causes of degraded exact recall in RNNs:

- **Compression bottleneck:** a fixed-size state must compress an ever-growing input history, forcing lossy overwriting of past information [5].
- **Nonlinear activations:** "Saturating" activation functions (such as sigmoid or tanh) "plateau" for large-magnitude inputs—once you're in the high or low tail, changing the input hardly moves the output and the derivative falls toward zero [25]. This flattening effect compresses small differences and makes gradients vanish even faster, which in turn makes it harder for the network to recover information from earlier time steps [35].
- **Finite precision:** floating-point rounding errors accumulate over long sequences, introducing numerical drift even when gradient issues are addressed. Numerical drift refers to the gradual divergence of a computed quantity from its exact theoretical value.
- **Sequence length:** as T grows, a fixed-dimensional state must compress an ever-increasing input history, leading to irreversible loss of earlier information and a bias toward recent inputs [6].
- **Number of input classes:** larger class sizes require the state to encode more distinct patterns, increasing the risk of detail loss when the representation is not sufficiently expressive.
- **Weight initialisation:** poor initial weight distributions can cause early saturation or unstable updates that limit effective storing and recall of inputs [36].

By examining these factors—alongside vanishing and exploding gradients—this dissertation aims to illuminate the broader set of conditions that influence the practical limits of storing and recalling inputs in RNNs.

2.10 Practical Optimisations for RNNs

In this section, contemporary practical optimisations that have been introduced to mitigate some of the problems that face RNNs are detailed.

2.10.1 Normalisation of Recurrent Weights

Schäfer and Zimmermann (2006) demonstrate that limiting the recurrent weight matrix to have normalised rows mitigates numerical instabilities in the state and extends perfect recall limits from around 10 to roughly 100 time steps without additional gating mechanisms [43].

Here, “normalised rows” means that each row vector of the recurrent weight matrix $\boldsymbol{W}_h$ is scaled to unit Euclidean norm (see section 2.2). If $\boldsymbol{w}_i$ denotes the ith row $\boldsymbol{W}_h$, then after normalisation

$$\boldsymbol{w}_i \leftarrow \frac{\boldsymbol{w}_i}{\|\boldsymbol{w}_i\|_2} \tag{2.44}$$

for all i. This ensures that $\|\boldsymbol{w}_i\|_2 = 1$ by scaling each row of $\boldsymbol{W}_h$ to unit length, the recurrent update keeps the magnitude of any vector it’s applied to, so gradients are prevented from exploding or vanishing too quickly.

2.10.2 Weight Initialisation

Properly chosen initial weights can prevent gradients from vanishing or exploding at the start of training. Le et al. (2015) show that initialising the recurrent weight matrix to the identity (and biases to zero) allows ReLU-based RNNs to preserve information over many time steps and recall long sequences to a higher degree of accuracy [27].

2.10.3 Gradient Clipping

To guard against exploding gradients, Pascanu et al. (2013) propose clipping the norm of the gradient vector during backpropagation to a fixed threshold. This intervention reduces any numerical instabilities during training and enables RNNs to handle longer sequences [35].

2.10.4 Orthogonal and Unitary Recurrent Matrices

Constraining the recurrent weight matrix to be orthogonal or unitary preserves the norm of error signals as they propagate through time. Arjovsky et al. (2016) demonstrate that unitary recurrent matrices can eliminate both vanishing and exploding gradients, improving the trainability of deep or long-sequence RNNs [4].

A real square matrix $\boldsymbol{W} \in \mathbb{R}^{d\times d}$ is called orthogonal if its transpose equals its inverse:

$$\boldsymbol{W}^\top \boldsymbol{W} = \boldsymbol{I}. \tag{2.45}$$

A complex square matrix $\boldsymbol{U} \in \mathbb{C}^{d\times d}$ is unitary if its conjugate transpose equals its inverse:

$$\boldsymbol{U}^* \boldsymbol{U} = \boldsymbol{I}, \tag{2.46}$$

where $\boldsymbol{U}^*$ denotes the conjugate-transpose of $\boldsymbol{U}$ (see Section 2.2).

2.11 Variants of RNNs

There are many different variants of the RNN model that have been developed in response to the limitations of the traditional RNN using some of the optimisations discussed above. In this section, I will introduce some of these models.

2.11.1 Elman Recurrent Neural Network

An Elman Recurrent Neural Network (ERNN) is an extension of the standard RNN architecture by incorporating context units into the layers. These context units store the previous time step's state and provide feedback to the network, enabling it to retain and reuse past information more effectively. The state ($\boldsymbol{h}_t$) in an ERNN is updated at each time step using the same equation as with a traditional RNN (2.36). The same goes for how the processed output ($\boldsymbol{y}_t$) is computed (2.37). The feedback loop created by the context units stores information from previous time steps, which makes the ERNN better at modelling temporal dependencies and storing more recent inputs. However, ERNNs can still suffer from challenges like vanishing gradients when handling long sequences.[1]

2.11.2 Jordan Recurrent Neural Network

A Jordan Recurrent Neural Network (JRNN) extends the standard RNN architecture by introducing context units that receive feedback from the output layer instead of the state. Context units store the previous output, enabling the network to better retain and process sequential information.

The state $\boldsymbol{h}_t$ in a JRNN is updated at each time step as:

$$\boldsymbol{h}_t = \sigma\big(\boldsymbol{W}_x \, \boldsymbol{x}_t + \boldsymbol{W}_c \, \boldsymbol{c}_{t-1} + \boldsymbol{b}_h\big) \tag{2.47}$$

where:

- $\boldsymbol{h}_t$ is the state at time t,
- $\boldsymbol{x}_t$ is the input at time t,
- $\boldsymbol{c}_{t-1}$ is the context vector storing the previous output,
- $\boldsymbol{W}_x$ and $\boldsymbol{W}_c$ are the weight matrices for input and context connections,
- $\boldsymbol{b}_h$ is the bias vector,
- σ is a non-linear activation function.

The context units are updated by:

$$\boldsymbol{c}_t = \sigma_c\big(\boldsymbol{y}_{t-1}\big) \tag{2.48}$$

where:

- $\boldsymbol{c}_t$ is the context vector at time t,
- $\boldsymbol{y}_{t-1}$ is the output vector from the previous time step,
- σ_c is the activation function applied to the previous output.

The processed output $\boldsymbol{y}_t$ is then computed as in standard RNNs (see (2.37)). The feedback mechanism in JRNNs integrates output feedback into the model, enabling it to retain recent output information and better capture sequential patterns. However, JRNNs still face challenges such as vanishing gradients, which limit their ability to store and recall inputs over long sequences [51].

2.11.3 Bidirectional RNNs

A bidirectional recurrent neural network (BRNN) has two layers of recurrent feedback loops that process the input sequence in opposite time directions: one in the forward direction (positive time) and the other in the backward direction (negative time). At each time step, the forward layer processes the input from the start of the sequence to the end, using context from the past. The backward layer processes the input from the end of the sequence to the start using context from the future. The outputs from the forward and backward layers are combined at each time step to produce the final output; this lets the BRNN use information from both past and future contexts simultaneously.

BRNNs are particularly useful in speech and text processing tasks where understanding context from both past and future is important. To train a BRNN, the network is unrolled over time, transforming it into a feedforward structure. This then allows the use of a modified Backpropagation Through Time (BPTT) algorithm on the network because backwards and forwards states are being processed simultaneously [42]. However, even the BRNN still suffers from factors such as the vanishing gradient problem, which hinders it's ability to store and recall inputs as sequence length grows .

2.12 Gated and Attention Based Models

While the previously mentioned RNN variants address some limitations of traditional RNNs, they still struggle with challenges like vanishing gradients, which hinder their ability to store and recall inputs from longer sequences. This section explores solutions that have claimed to be designed to overcome these limitations.

2.12.1 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are claimed to be designed to overcome the vanishing gradient problem by having memory cells and gated architecture. The memory cell ($\boldsymbol{c}_t$)is designed to preserve information across time steps, helping the model recall specific inputs when needed. By regulating the flow of information into, out of, and within the memory cell, the gated architecture allows for longer sequences of inputs to be stored and recalled to a higher degree than traditional RNNs. This is because it ensures that only relevant data is retained and passed on [33]. The state update in an LSTM is controlled by three gates:

- **Forget Gate:** Determines what information to discard from the cell state.

$$\boldsymbol{f}_t = \sigma\big(\boldsymbol{W}_f\,[\boldsymbol{x}_t,\ \boldsymbol{h}_{t-1}] + \boldsymbol{b}_f\big) \tag{2.49}$$

- **Input Gate:** Controls how much new information is added to the cell state. It consists of two parts: the gate activation and the candidate state update.

$$\boldsymbol{i}_t = \sigma\big(\boldsymbol{W}_i\,[\boldsymbol{x}_t,\ \boldsymbol{h}_{t-1}] + \boldsymbol{b}_i\big), \tag{2.50}$$

$$\tilde{\boldsymbol{c}}_t = \tanh\big(\boldsymbol{W}_c\,[\boldsymbol{x}_t,\ \boldsymbol{h}_{t-1}] + \boldsymbol{b}_c\big). \tag{2.51}$$

- **Output Gate:** Regulates the flow of information from the memory cell to the output state.

$$\boldsymbol{o}_t = \sigma\big(\boldsymbol{W}_o\,[\boldsymbol{x}_t,\ \boldsymbol{h}_{t-1}] + \boldsymbol{b}_o\big) \tag{2.52}$$

The final output state is computed as:

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) \tag{2.53}$$

Where:

- $\boldsymbol{x}_t$ is the input vector at time t.
- $\boldsymbol{h}_{t-1}$ is the previous state vector.
- $\boldsymbol{c}_t$ is the cell memory vector at time t.
- $\boldsymbol{f}_t$, $\boldsymbol{i}_t$, $\boldsymbol{o}_t$ are the forget, input, and output gate activation vectors, respectively.
- $\tilde{\boldsymbol{c}}_t$ is the candidate cell state vector.
- $\boldsymbol{W}_f, \boldsymbol{W}_i, \boldsymbol{W}_c, \boldsymbol{W}_o$ are the weight matrices for the forget, input, candidate, and output gates.
- $\boldsymbol{b}_f, \boldsymbol{b}_i, \boldsymbol{b}_c, \boldsymbol{b}_o$ are the corresponding bias vectors.
- $\sigma(\cdot)$ is the sigmoid activation function.
- $\tanh(\cdot)$ is the hyperbolic tangent activation function.
- $\odot$ denotes element-wise multiplication.

The combined effect of these gates enables LSTMs to selectively remember, update, or forget information.

LSTMs still have limitations; this includes a relatively higher computational cost due to their more advanced architecture. Furthermore, while LSTMs mitigate the vanishing gradient problem, they are not entirely immune to it when modelling long sequences [23].

2.12.2 Gated Recurrent Units (GRU)

Gated Recurrent Units (GRUs) are a simplified variant of LSTM networks, designed to manage information flow through fewer gates. GRUs were introduced to address some of the complexity of LSTMs while providing comparable performance in many sequence modelling tasks [11]. The update equations for GRUs replace the separate forget and input gates in LSTMs with a single update gate. This reduces computational complexity. The state ($\boldsymbol{h}_t$) in a GRU is updated using the following equations:

$$\boldsymbol{z}_t = \sigma(\boldsymbol{W}_z\boldsymbol{x}_t + \boldsymbol{U}_z\boldsymbol{h}_{t-1} + \boldsymbol{b}_z) \tag{2.54}$$

$$\boldsymbol{r}_t = \sigma(\boldsymbol{W}_r\boldsymbol{x}_t + \boldsymbol{U}_r\boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \tag{2.55}$$

$$\tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}_h\boldsymbol{x}_t + \boldsymbol{r}_t \odot (\boldsymbol{U}_h\boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \tag{2.56}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t \tag{2.57}$$

where:

- $\boldsymbol{z}_t$ is the **update gate**, which determines how much of the previous state $\boldsymbol{h}_{t-1}$ should be retained,
- $\boldsymbol{r}_t$ is the **reset gate**, which controls how much of the previous state contributes to the current candidate state,
- $\tilde{\boldsymbol{h}}_t$ is the candidate state, computed using the reset gate $\boldsymbol{r}_t$,
- $\odot$ denotes the element-wise product,
- σ is the sigmoid activation function, and tanh applies a hyperbolic tangent activation.

Despite their efficiency, GRUs still have limitations. Similarly to LSTMs, they can still struggle with long sequences due to their sequential nature, and their performance heavily depends on the dataset and task at hand. Empirical evaluations, such as those by Chung et al. [11], have shown that GRUs can perform comparably to LSTMs in many sequence modelling tasks, making them a strong alternative when computational resources are limited.

2.12.3 Attention Based Models

Other strategies have been developed to address long-term sequence modelling issues:

- **Attention Mechanisms:** Attention mechanisms claim to enhance the performance of neural network models by allocating varying degrees of importance to different parts of the input, allowing the model to concentrate on the most relevant information. This helps to significantly improve the storing and recalling of inputs in long sequences. [48]
- **Transformers:** Transformers extend attention mechanisms by using multi-head attention and positional encoding, allowing them to effectively capture and retain earlier time step input influences. The parallelisable architecture enables them to process entire sequences simultaneously, making them efficient.
  - Multi-head attention is a mechanism that allows a model to attend to different parts of the input sequence simultaneously by computing multiple independent attention distributions.
  - Positional encoding is a technique used to inject information about the order of elements in a sequence into a model, enabling it to capture positional relationships that are otherwise missed by attention mechanisms.

  [48].

  While transformers are computationally intensive due to the complexity of self-attention, they claim to outperform the other solutions discussed in scalability and storing and recalling inputs in longer sequences.

  ***Note:* LSTMs and GRUs are still variants of the Recurrent Neural Network, whilst attention based models such as Transformers have claimed to be a departure from the RNN**

2.12.4 Summary and Comparison of Existing Solutions

LSTMs and GRUs have claimed to be effective at mitigating vanishing gradients and storing and recalling inputs, particularly in tasks involving sequential data. However, they remain computationally expensive and can still struggle with extremely long sequences, as they still don't offer a solution to the numerical stability that arises from linear compression. Recent innovations like attention mechanisms and transformers claim to overcome numerical instability and vanishing/exploding gradients, claiming to offer superior performance in tasks requiring the modelling of very long sequences. Despite their advantages, these methods often demand more computational resources and may not be suitable for all applications. Table 2.1 provides a comparison of these solutions.

Model	**Handles Vanishing Gradients better than traditional RNN**	**Computational Cost**	**Handles Very Long Sequences**
LSTM	Yes	High	Limited
GRU	Yes	Moderate	Limited
Attention Mechanisms	Yes	High	Excellent
Transformers	Yes	Very High	Excellent

Table 2.1: Comparison of Input Storage and Recall Solutions

2.13 Input Storage and Recall Testing

This section explains how models are evaluated on their ability to store and later recall specific inputs from sequences. The following benchmarks and metrics have been adapted from memorisation tasks to directly assess the fidelity of input storage and recall, which is central to this dissertation.

2.13.1 Benchmarks for Input Storage and Recall

Evaluating a model's capacity to accurately store and recall inputs involves tasks that test its ability to maintain detailed information over varying sequence lengths. Common benchmarks include:

- **Copy Tasks:** The model is trained to reproduce an input sequence after a delay. This task tests the model's ability to store and retrieve information from earlier time steps.[2]
- **Sequence Prediction Tasks:** The model is required to predict the next element in a sequence or retrieve a specific element from a historical state.[46]
- **Algorithmic Tasks:** Tasks like sorting or reversing sequences measure a model's ability to maintain precise state information across time steps.[38]

These benchmarks are specifically designed to evaluate the practical limits of storing and recalling sequential inputs rather than the more general concept of memorisation.

2.13.2 Evaluation Metrics

To quantify a model's performance in storing and recalling inputs, several metrics are used:

- **Mean Squared Error (MSE):** Assesses the difference between the model's reconstructed outputs and the original inputs, indicating how closely the recalled sequence matches the stored sequence [47].

- **Categorical Accuracy:** Measures the proportion of correctly recalled elements in classification-based tasks, useful for assessing exact input retrieval [32].
- **Cross-Entropy Loss:** Evaluates the divergence between predicted probabilities and the true labels in categorical recall tasks, penalising incorrect reconstructions [32].
- **Memory Capacity:** Quantifies the amount of information a model can store and recall as a function of sequence length, providing a direct measure of its input retention ability [15].
- **Error Rates by Sequence Length:** Analyzing how error rates evolve with increasing sequence lengths reveals the practical limitations of the model's storage and recall mechanisms [39].

2.13.3 Transition to My Solution

This section has outlined the benchmarks and evaluation metrics used to assess the practical limits of storing and recalling sequential inputs. In the following chapter, the focus will shift to the proposed solutions. I will detail how to structure a standard RNN to be able to overcome numerical instability caused by its linear compression as well as how to circumnavigate vanishing/exploding gradients. In the process, showcasing how to structure the RNN to overcome its limitations.

Chapter 3

Analysis

This chapter introduces the proposed solution and a discussion on it.

3.1 Introduction

Chapter 2 highlighted how RNNs have been seen to fail to retain information over long sequences due to issues like vanishing gradients and overwriting of past states. It was also seen that more complex architectures can mitigate but not eliminate these problems at the added expense of increased complexity. This leaves open the question of whether a carefully structured RNN can still handle long-term storage and even state reconstruction if information is perfectly stored in the state. Chapter 3 addresses this by proposing an RNN with state scaling and structured parameters, whose mathematical framework shows how it preserves historical inputs in a blockwise fashion and analyses its numerical stability. In doing so, insight is provided into the practical limits of standard RNNs and illustrates conditions under which exact sequence recall is possible.

***Note:* Importantly, the RNN (state-scaled with structured parameters) introduces no new gating mechanisms or memory units—its recurrence is identical to a standard RNN's. Instead, by restructuring the state vector into contiguous blocks and pairing it with an invertible activation, it achieves exact sequence recall.**

3.2 Programme of Research

This chapter is structured into three main sections:

1. **Proof of Concept:** Demonstrating that an RNN (state-scaled with structured parameters) can recall input sequences of arbitrary length by storing all historical information in its state.
2. **Numerical Stability and Scaling:** Addressing how the model avoids the numerical underflow issues (common when multiplying thousands of probabilities) that are problematic in fields such as speech recognition. The section also discusses how scaling and the clamping of the activation functions help maintain stability.

3. **Compression and Overwriting:** Discussing why simple linear transformations are insufficient for compressing sequential information—information can be quickly overwritten by naive approaches. Although the proof of concept (step 1) overcomes numerical instability (step 2), effective compression (step 3) remains an open challenge.

3.3 Proof of Concept

This Proof of Concept shows that the RNN (state-scaled with structured parameters) can encode the entire sequence in its final state and that with an invertible activation function, the state and input sequence can be exactly reconstructed from this final state.

3.3.1 Problem Setup

Consider discrete sequences where each input is a one-hot vector. Let $K = 3$ be the number of classes, encoded as:

$$\text{Class 1} \implies \boldsymbol{x} = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \text{Class 2} \implies \boldsymbol{x} = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \text{Class 3} \implies \boldsymbol{x} = \begin{bmatrix}0\\0\\1\end{bmatrix}.$$

Here $\boldsymbol{x}^{(k)}$ denotes the one-hot vector for class k. Let $L_{\max} = 4$ be the input length. For the proof of concept, the class labels are $\{2, 1, 3, 2\}$, yielding

$$\boldsymbol{x}_1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \boldsymbol{x}_2 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \boldsymbol{x}_3 = \begin{bmatrix}0\\0\\1\end{bmatrix}, \quad \boldsymbol{x}_4 = \begin{bmatrix}0\\1\\0\end{bmatrix}.$$

3.3.2 Notation and Dimensions

Inputs $\boldsymbol{x}_t \in \{0,1\}^K$ are one-hot vectors over $1 \leq k \leq K$. They are embedded via

$$\boldsymbol{e}_t = \boldsymbol{W}^x \boldsymbol{x}_t \in \mathbb{R}^D,$$

and the RNN state $\boldsymbol{h}_t \in \mathbb{R}^D$ lives in the same D-dimensional space. Thus K is the number of classes, and D the embedding/state size.

3.3.3 Typical RNN Structure and Encoding

At each step the RNN updates its state:

$$\boldsymbol{h}_t = \sigma\big(\boldsymbol{W}^h \boldsymbol{h}_{t-1} + \boldsymbol{W}^x \boldsymbol{x}_t + \boldsymbol{b}_h + \boldsymbol{b}_x\big), \tag{3.1}$$

where:

- $\boldsymbol{h}_t$ is the state vector at time t,
- $\boldsymbol{x}_t$ is the input vector at time t,

- $\boldsymbol{W}^h$ and $\boldsymbol{W}^x$ are the state and input weight matrices,
- $\boldsymbol{b}_h$ and $\boldsymbol{b}_x$ are the bias vectors,
- $\sigma(\cdot)$ is the activation function.

The weight matrices for $k = 3$ classes would be:

$$\boldsymbol{W}^{\mathtt{h}} = \begin{bmatrix} w^{\mathtt{h}}_{1,1} & w^{\mathtt{h}}_{1,2} & w^{\mathtt{h}}_{1,3} \\ w^{\mathtt{h}}_{2,1} & w^{\mathtt{h}}_{2,2} & w^{\mathtt{h}}_{2,3} \\ w^{\mathtt{h}}_{3,1} & w^{\mathtt{h}}_{3,2} & w^{\mathtt{h}}_{3,3} \end{bmatrix}, \quad \boldsymbol{W}^{\mathtt{x}} = \begin{bmatrix} w^{\mathtt{x}}_{1,1} & w^{\mathtt{x}}_{1,2} & w^{\mathtt{x}}_{1,3} \\ w^{\mathtt{x}}_{2,1} & w^{\mathtt{x}}_{2,2} & w^{\mathtt{x}}_{2,3} \\ w^{\mathtt{x}}_{3,1} & w^{\mathtt{x}}_{3,2} & w^{\mathtt{x}}_{3,3} \end{bmatrix}. \tag{3.2}$$

The bias vectors are:

$$\boldsymbol{b}^{\mathtt{h}} = \begin{bmatrix} b^{\mathtt{h}}_1 \\ b^{\mathtt{h}}_2 \\ b^{\mathtt{h}}_3 \end{bmatrix}, \quad \boldsymbol{b}^{\mathtt{x}} = \begin{bmatrix} b^{\mathtt{x}}_1 \\ b^{\mathtt{x}}_2 \\ b^{\mathtt{x}}_3 \end{bmatrix}. \tag{3.3}$$

Encoding:

Assume the initial state is empty:

$$\boldsymbol{h}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{3.4}$$

Consider the previously defined input sequence $\{2, 1, 3, 2\}$. The state is computed at each time step as follows:

Step 1: Input $\boldsymbol{x}^{(1)} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

$$\boldsymbol{h}_1 = \sigma(\boldsymbol{W}^{\mathtt{h}}\boldsymbol{h}_0 + \boldsymbol{W}^{\mathtt{x}}\boldsymbol{x}^{(1)} + \boldsymbol{b}), \tag{3.5}$$

where:

$$\boldsymbol{W}^{\mathtt{h}}\boldsymbol{h}_0 = \begin{bmatrix} w^{\mathtt{h}}_{1,1} & w^{\mathtt{h}}_{1,2} & w^{\mathtt{h}}_{1,3} \\ w^{\mathtt{h}}_{2,1} & w^{\mathtt{h}}_{2,2} & w^{\mathtt{h}}_{2,3} \\ w^{\mathtt{h}}_{3,1} & w^{\mathtt{h}}_{3,2} & w^{\mathtt{h}}_{3,3} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{3.6}$$

$$\boldsymbol{W}^{\mathtt{x}}\boldsymbol{x}^{(1)} = \begin{bmatrix} w^{\mathtt{x}}_{1,1} & w^{\mathtt{x}}_{1,2} & w^{\mathtt{x}}_{1,3} \\ w^{\mathtt{x}}_{2,1} & w^{\mathtt{x}}_{2,2} & w^{\mathtt{x}}_{2,3} \\ w^{\mathtt{x}}_{3,1} & w^{\mathtt{x}}_{3,2} & w^{\mathtt{x}}_{3,3} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} w^{\mathtt{x}}_{1,2} \\ w^{\mathtt{x}}_{2,2} \\ w^{\mathtt{x}}_{3,2} \end{bmatrix}. \tag{3.7}$$

Adding the bias term:

$$\boldsymbol{h}_1 = \boldsymbol{\sigma}\left(\begin{bmatrix} w^{\mathtt{x}}_{1,2} \\ w^{\mathtt{x}}_{2,2} \\ w^{\mathtt{x}}_{3,2} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right). \tag{3.8}$$

Step 2: Input $\boldsymbol{x}^{(2)} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

$$\boldsymbol{h}_2 = \sigma(\boldsymbol{W}^{\text{h}}\boldsymbol{h}_1 + \boldsymbol{W}^{\text{x}}\boldsymbol{x}^{(2)} + \boldsymbol{b}), \tag{3.9}$$

where:

$$\boldsymbol{W}^{\text{h}}\boldsymbol{h}_1 = \begin{bmatrix} w^{\mathtt{h}}_{1,1} & w^{\mathtt{h}}_{1,2} & w^{\mathtt{h}}_{1,3} \\ w^{\mathtt{h}}_{2,1} & w^{\mathtt{h}}_{2,2} & w^{\mathtt{h}}_{2,3} \\ w^{\mathtt{h}}_{3,1} & w^{\mathtt{h}}_{3,2} & w^{\mathtt{h}}_{3,3} \end{bmatrix} \begin{bmatrix} \sigma(w^{\mathtt{x}}_{1,2} + b_1) \\ \sigma(w^{\mathtt{x}}_{2,2} + b_2) \\ \sigma(w^{\mathtt{x}}_{3,2} + b_3) \end{bmatrix}. \tag{3.10}$$

Expanding row-by-row:

$$\boldsymbol{W}^{\text{h}}\boldsymbol{h}_1 = \begin{bmatrix} w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,2} + b_1) + w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,2} + b_2) + w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,2} + b_3) \\ w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,2} + b_1) + w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,2} + b_2) + w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,2} + b_3) \\ w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,2} + b_1) + w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,2} + b_2) + w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,2} + b_3) \end{bmatrix}. \tag{3.11}$$

$$\boldsymbol{W}^{\text{x}}\boldsymbol{x}^{(2)} = \begin{bmatrix} w^{\mathtt{x}}_{1,1} \\ w^{\mathtt{x}}_{2,1} \\ w^{\mathtt{x}}_{3,1} \end{bmatrix}. \tag{3.12}$$

Adding the bias term:

$$\boldsymbol{h}_2 = \boldsymbol{\sigma}\left(\begin{bmatrix} w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,2} + b_1) + w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,2} + b_2) + w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,2} + b_3) \\ w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,2} + b_1) + w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,2} + b_2) + w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,2} + b_3) \\ w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,2} + b_1) + w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,2} + b_2) + w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,2} + b_3) \end{bmatrix} + \begin{bmatrix} w^{\mathtt{x}}_{1,1} \\ w^{\mathtt{x}}_{2,1} \\ w^{\mathtt{x}}_{3,1} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right). \tag{3.13}$$

Step 3: Input $\boldsymbol{x}^{(3)} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

$$\boldsymbol{h}_3 = \sigma(\boldsymbol{W}^{\text{h}}\boldsymbol{h}_2 + \boldsymbol{W}^{\text{x}}\boldsymbol{x}^{(3)} + \boldsymbol{b}), \tag{3.14}$$

where:

$$\boldsymbol{W}^{\text{h}}\boldsymbol{h}_2 = \begin{bmatrix} w^{\mathtt{h}}_{1,1} & w^{\mathtt{h}}_{1,2} & w^{\mathtt{h}}_{1,3} \\ w^{\mathtt{h}}_{2,1} & w^{\mathtt{h}}_{2,2} & w^{\mathtt{h}}_{2,3} \\ w^{\mathtt{h}}_{3,1} & w^{\mathtt{h}}_{3,2} & w^{\mathtt{h}}_{3,3} \end{bmatrix} \begin{bmatrix} \sigma(w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,2} + b_1) + w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,2} + b_2) + w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,2} + b_3)) \\ \sigma(w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,2} + b_1) + w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,2} + b_2) + w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,2} + b_3)) \\ \sigma(w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,2} + b_1) + w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,2} + b_2) + w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,2} + b_3)) \end{bmatrix}. \tag{3.15}$$

Expanding row-by-row:

$$\boldsymbol{W}^{\mathrm{h}}\boldsymbol{h}_2 = \begin{bmatrix} w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3)) \\ w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3)) \\ w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3)) \end{bmatrix}. \tag{3.16}$$

$$\boldsymbol{W}^{\mathrm{x}}\boldsymbol{x}^{(3)} = \begin{bmatrix} w^{\mathtt{x}}_{1,3} \\ w^{\mathtt{x}}_{2,3} \\ w^{\mathtt{x}}_{3,3} \end{bmatrix}. \tag{3.17}$$

Adding the bias term:

$$\boldsymbol{h}_3 = \boldsymbol{\sigma}\left(\begin{bmatrix} w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3)) \\ w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3)) \\ w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3))+ \\ w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,2}+b_1)+w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,2}+b_2)+w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,2}+b_3)) \end{bmatrix} .+ \begin{bmatrix} w^{\mathtt{x}}_{1,3} \\ w^{\mathtt{x}}_{2,3} \\ w^{\mathtt{x}}_{3,3} \end{bmatrix} .+ \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right). \tag{3.18}$$

Step 4: Input $\boldsymbol{x}^{(4)} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

$$\boldsymbol{h}_4 = \sigma(\boldsymbol{W}^{\mathrm{h}}\boldsymbol{h}_3 + \boldsymbol{W}^{\mathrm{x}}\boldsymbol{x}^{(4)} + \boldsymbol{b}), \tag{3.19}$$

where:

$$\boldsymbol{W}^{\mathrm{h}}\boldsymbol{h}_3 = \begin{bmatrix} w^{\mathtt{h}}_{1,1} & w^{\mathtt{h}}_{1,2} & w^{\mathtt{h}}_{1,3} \\ w^{\mathtt{h}}_{2,1} & w^{\mathtt{h}}_{2,2} & w^{\mathtt{h}}_{2,3} \\ w^{\mathtt{h}}_{3,1} & w^{\mathtt{h}}_{3,2} & w^{\mathtt{h}}_{3,3} \end{bmatrix} \begin{bmatrix} \sigma(w^{\mathtt{h}}_{1,1}\sigma(w^{\mathtt{x}}_{1,3}+b_1)+w^{\mathtt{h}}_{1,2}\sigma(w^{\mathtt{x}}_{2,3}+b_2)+w^{\mathtt{h}}_{1,3}\sigma(w^{\mathtt{x}}_{3,3}+b_3)) \\ \sigma(w^{\mathtt{h}}_{2,1}\sigma(w^{\mathtt{x}}_{1,3}+b_1)+w^{\mathtt{h}}_{2,2}\sigma(w^{\mathtt{x}}_{2,3}+b_2)+w^{\mathtt{h}}_{2,3}\sigma(w^{\mathtt{x}}_{3,3}+b_3)) \\ \sigma(w^{\mathtt{h}}_{3,1}\sigma(w^{\mathtt{x}}_{1,3}+b_1)+w^{\mathtt{h}}_{3,2}\sigma(w^{\mathtt{x}}_{2,3}+b_2)+w^{\mathtt{h}}_{3,3}\sigma(w^{\mathtt{x}}_{3,3}+b_3)) \end{bmatrix}. \tag{3.20}$$

Expanding row-by-row:

$$\boldsymbol{W}^{\mathrm{h}}\boldsymbol{h}_3 = \begin{bmatrix} w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3)) \\ w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3)) \\ w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3)) \end{bmatrix}. \tag{3.21}$$

$$\boldsymbol{W}^{\mathrm{x}}\boldsymbol{x}^{(4)} = \begin{bmatrix} w^{\mathrm{x}}_{1,2} \\ w^{\mathrm{x}}_{2,2} \\ w^{\mathrm{x}}_{3,2} \end{bmatrix}. \tag{3.22}$$

Adding the bias term:

$$\boldsymbol{h}_4 = \sigma\left(\begin{bmatrix} w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3)) \\ w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3)) \\ w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3))+ \\ w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,3}+b_1)+w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,3}+b_2)+w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,3}+b_3)) \end{bmatrix} . + \begin{bmatrix} w^{\mathrm{x}}_{1,2} \\ w^{\mathrm{x}}_{2,2} \\ w^{\mathrm{x}}_{3,2} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right). \tag{3.23}$$

Here it can be seen that with the fixed state size $\boldsymbol{h}_t$, not all the information from previous states and inputs is stored in it. One of the reasons this happens can be seen in the encoding, when the previous state $\boldsymbol{h}_{t-1}$ is compressed with the weight matrix $\boldsymbol{W}^{\mathrm{h}}$, and the product $\boldsymbol{W}^{\mathrm{h}}\boldsymbol{h}_3$ is used to contribute towards $\boldsymbol{h}_t$.

3.3.4 RNN (state-scaled with structured parameters) Structure

Although we introduce new notation $\widehat{\boldsymbol{h}}_t$, the RNN (state-scaled with structured parameters) remains a standard RNN:

$$\widehat{\boldsymbol{h}}_t = \sigma(\widehat{\boldsymbol{W}}^{\mathrm{h}}\widehat{\boldsymbol{h}}_{t-1} + \widehat{\boldsymbol{W}}^{\mathrm{x}}\boldsymbol{x}_t + \widehat{\boldsymbol{b}}^{\mathrm{h}} + \widehat{\boldsymbol{b}}^{\mathrm{x}}), \tag{3.24}$$

Where:

- $\widehat{\boldsymbol{h}}_t$ is the modified state at time step t,

  - After t steps, $\widehat{\boldsymbol{h}}_t$ holds

$$\begin{bmatrix} \boldsymbol{h}_t \\ \sigma(\boldsymbol{h}_{t-1}) \\ \sigma^2(\boldsymbol{h}_{t-2}) \\ \vdots \\ \sigma^t(\boldsymbol{h}_0) \end{bmatrix}$$

  - Where each successive older state is wrapped in one more $\boldsymbol{\sigma}$.
  - The state dimension is expanded from $\boldsymbol{D}$ to $\boldsymbol{D} \times L_{\max}$, split into $L_{\max}$ blocks of size $\boldsymbol{D}$. At each step, the newest state occupies the first block, and all other blocks shift down by one, so past states are preserved rather than overwritten.

- σ is the element-wise non-linear, invertible activation function.
- $\widehat{\boldsymbol{h}}_{t-1}$ is the previous modified state from time step $t-1$,
- $\boldsymbol{x}_t$ is the input at time step t,
- $\widehat{\boldsymbol{W}}^{\mathtt{h}}$ is the modified state transition weight matrix of size $\boldsymbol{D}L_{\max} \times \boldsymbol{D}L_{\max}$:

  –

$$\widehat{\boldsymbol{W}}^{\mathtt{h}} = \begin{bmatrix} w^{\mathtt{h}}_{1,1} & w^{\mathtt{h}}_{1,2} & w^{\mathtt{h}}_{1,3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ w^{\mathtt{h}}_{2,1} & w^{\mathtt{h}}_{2,2} & w^{\mathtt{h}}_{2,3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ w^{\mathtt{h}}_{3,1} & w^{\mathtt{h}}_{3,2} & w^{\mathtt{h}}_{3,3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}_{\boldsymbol{D}L_{\max} \times \boldsymbol{D}L_{\max}} \tag{3.25}$$

  - Where the top-left 3×3 block is the original $\boldsymbol{W}^h$, and the identity band $\boldsymbol{I}_{3(L_{\max}-1)}$ shifts the old $\boldsymbol{D}$-blocks down.

- $\widehat{\boldsymbol{W}}^{\mathtt{x}}$ is the modified input matrix, of size $\boldsymbol{D}L_{\max} \times K$:

–
$$\widehat{\boldsymbol{W}}^{\mathrm{x}} = \begin{bmatrix} w^{\mathtt{x}}_{1,1} & w^{\mathtt{x}}_{1,2} & w^{\mathtt{x}}_{1,3} \\ w^{\mathtt{x}}_{2,1} & w^{\mathtt{x}}_{2,2} & w^{\mathtt{x}}_{2,3} \\ w^{\mathtt{x}}_{3,1} & w^{\mathtt{x}}_{3,2} & w^{\mathtt{x}}_{3,3} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}_{DL_{\mathtt{max}} \times K} . \tag{3.26}$$

- So only the top $\boldsymbol{D} \times K$ block carries the new embedding, and the rest is zero, with enough space for the rest of the embeddings in the sequence.

- The bias vectors $\widehat{\boldsymbol{b}}^{\mathrm{h}}$, $\widehat{\boldsymbol{b}}^{\mathrm{x}}$ each have size $\boldsymbol{D}L_{\max}$:

–
$$\widehat{\boldsymbol{b}}^{\mathrm{x}} = \begin{bmatrix} b^{\mathtt{x}}_{1} \\ b^{\mathtt{x}}_{2} \\ b^{\mathtt{x}}_{3} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}_{DL_{\mathtt{max}}} , \quad \widehat{\boldsymbol{b}}^{\mathrm{h}} = \begin{bmatrix} b^{\mathtt{h}}_{1} \\ b^{\mathtt{h}}_{2} \\ b^{\mathtt{h}}_{3} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}_{DL_{\mathtt{max}}} . \tag{3.27}$$

- **$\widehat{\boldsymbol{b}}^{\mathrm{x}}$ and $\widehat{\boldsymbol{b}}^{\mathrm{h}}$:** Designed as sparse vectors that scale the current bias and shift the other content down, with enough space for all bias information:

The hats over the variables represent modified or specific versions of the standard RNN components. These modifications are essential to enabling the RNN (state-scaled with structured parameters) to attempt exact sequence storage and recall. These sparse matrix and vector designs ensure that all information can be stored within the RNN (state-scaled with structured parameters) without any of it being lost, as shown in the typical RNN structure during encoding.

Encoding Sequences Using the RNN (state-scaled with structured parameters)

Using the sequence defined in the problem setup $\{2, 1, 3, 2\}$. Given $\widehat{\boldsymbol{h}}_0 = \mathbf{0}$, applying the modified equation at each time step:

– **1st Input, Class** 2**, 1st State**

$$\widehat{\boldsymbol{h}}_1 = \begin{bmatrix} \sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) \\ \sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) \\ \sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3) \\ \sigma(0) \\ \sigma(0) \\ \sigma(0) \\ \sigma(0) \\ \sigma(0) \\ \sigma(0) \\ \sigma(0) \\ \sigma(0) \\ \sigma(0) \end{bmatrix} = \begin{bmatrix} \boldsymbol{h}_1 \\ \boldsymbol{\sigma}(\mathbf{0}) \\ \boldsymbol{\sigma}(\mathbf{0}) \\ \boldsymbol{\sigma}(\mathbf{0}) \end{bmatrix} \tag{3.28}$$

– **2nd Input, Class** 1**, 2nd State**

$$\widehat{\boldsymbol{h}}_2 = \begin{bmatrix} \sigma(w^{\mathrm{x}}_{1,1} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) \\ \sigma(w^{\mathrm{x}}_{2,1} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) \\ \sigma(w^{\mathrm{x}}_{3,1} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3) \\ \sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)) \\ \sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)) \\ \sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)) \\ \sigma(\sigma(0)) \\ \sigma(\sigma(0)) \\ \sigma(\sigma(0)) \\ \sigma(\sigma(0)) \\ \sigma(\sigma(0)) \\ \sigma(\sigma(0)) \end{bmatrix} = \begin{bmatrix} \boldsymbol{h}_2 \\ \boldsymbol{\sigma}(\boldsymbol{h}_1) \\ \boldsymbol{\sigma}(\boldsymbol{\sigma}(\mathbf{0})) \\ \boldsymbol{\sigma}(\boldsymbol{\sigma}(\mathbf{0})) \end{bmatrix} \tag{3.29}$$

– **3rd Input, Class 3, 3rd State**

$$\widehat{\boldsymbol{h}}_3 = \begin{bmatrix} \sigma(w^{\mathrm{x}}_{1,3} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) \\ \sigma(w^{\mathrm{x}}_{2,3} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) \\ \sigma(w^{\mathrm{x}}_{3,3} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3) \\ \sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,1} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,1} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,1} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)) \\ \sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,1} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,1} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,1} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)) \\ \sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,1} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,1} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,1} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)) \\ \sigma(\sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3))) \\ \sigma(\sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3))) \\ \sigma(\sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3))) \\ \sigma(\sigma(\sigma(0))) \\ \sigma(\sigma(\sigma(0))) \\ \sigma(\sigma(\sigma(0))) \end{bmatrix} = \begin{bmatrix} \boldsymbol{h}_3 \\ \boldsymbol{\sigma}(\boldsymbol{h}_2) \\ \boldsymbol{\sigma}(\boldsymbol{\sigma}(\boldsymbol{h}_1)) \\ \boldsymbol{\sigma}(\boldsymbol{\sigma}(\boldsymbol{\sigma}(\mathbf{0}))) \end{bmatrix} \tag{3.30}$$

– **4th input, class 2, 4th state**

$$\widehat{\boldsymbol{h}}_4 = \begin{bmatrix} \sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) \\ \sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) \\ \sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3) \\ \sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,3} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,3} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,3} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)) \\ \sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,3} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,3} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,3} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)) \\ \sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,3} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,3} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,3} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)) \\ \sigma(\sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,1} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,1} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,1} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3))) \\ \sigma(\sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,1} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,1} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,1} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3))) \\ \sigma(\sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,1} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,1} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,1} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3))) \\ \sigma(\sigma(\sigma(w^{\mathrm{h}}_{1,1}\sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{1,2}\sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{1,3}\sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)))) \\ \sigma(\sigma(\sigma(w^{\mathrm{h}}_{2,1}\sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{2,2}\sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{2,3}\sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)))) \\ \sigma(\sigma(\sigma(w^{\mathrm{h}}_{3,1}\sigma(w^{\mathrm{x}}_{1,2} + b^{\mathrm{h}}_1 + b^{\mathrm{x}}_1) + w^{\mathrm{h}}_{3,2}\sigma(w^{\mathrm{x}}_{2,2} + b^{\mathrm{h}}_2 + b^{\mathrm{x}}_2) + w^{\mathrm{h}}_{3,3}\sigma(w^{\mathrm{x}}_{3,2} + b^{\mathrm{h}}_3 + b^{\mathrm{x}}_3)))) \end{bmatrix} = \begin{bmatrix} \boldsymbol{h}_4 \\ \boldsymbol{\sigma}(\boldsymbol{h}_3) \\ \boldsymbol{\sigma}(\boldsymbol{\sigma}(\boldsymbol{h}_2)) \\ \boldsymbol{\sigma}(\boldsymbol{\sigma}(\boldsymbol{\sigma}(\boldsymbol{h}_1))) \end{bmatrix} \tag{3.31}$$

As can be seen with the information highlighted in blue, the final state encodes the entire input sequence but in reverse order due to how the states are appended. In typical RNN encoding, information is overridden, which doesn't happen with the RNN (state-scaled with structured parameters) version. This is because in the RNN (state-scaled with structured parameters), there is enough state space for all of the information from historical states to exist in the final state.

3.3.5 Getting the Input from Word Embeddings

The inputs correspond to their grouped weight matrices, represented as word embeddings. Word embeddings are used, as they give a clearer mapping between inputs and weight matrix, as well as helping to generalise the approach beyond the current setup. Each input $\boldsymbol{x}_t$ is mapped to a specific embedding $\boldsymbol{e}^{(k)}$, where k is the class label.

The embeddings are defined as follows:

$$\boldsymbol{e}^{(1)} = \begin{bmatrix} w^{\mathrm{x}}_{1,1} \\ w^{\mathrm{x}}_{2,1} \\ w^{\mathrm{x}}_{3,1} \end{bmatrix} = \begin{bmatrix} w^{\mathrm{x}}_{1,1} & w^{\mathrm{x}}_{1,2} & w^{\mathrm{x}}_{1,3} \\ w^{\mathrm{x}}_{2,1} & w^{\mathrm{x}}_{2,2} & w^{\mathrm{x}}_{2,3} \\ w^{\mathrm{x}}_{3,1} & w^{\mathrm{x}}_{3,2} & w^{\mathrm{x}}_{3,3} \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{3.32}$$

$$\boldsymbol{e}^{(2)} = \begin{bmatrix} w^{\mathrm{x}}_{1,2} \\ w^{\mathrm{x}}_{2,2} \\ w^{\mathrm{x}}_{3,2} \end{bmatrix} = \begin{bmatrix} w^{\mathrm{x}}_{1,1} & w^{\mathrm{x}}_{1,2} & w^{\mathrm{x}}_{1,3} \\ w^{\mathrm{x}}_{2,1} & w^{\mathrm{x}}_{2,2} & w^{\mathrm{x}}_{2,3} \\ w^{\mathrm{x}}_{3,1} & w^{\mathrm{x}}_{3,2} & w^{\mathrm{x}}_{3,3} \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{3.33}$$

$$\boldsymbol{e}^{(3)} = \begin{bmatrix} w^{\mathrm{x}}_{1,3} \\ w^{\mathrm{x}}_{2,3} \\ w^{\mathrm{x}}_{3,3} \end{bmatrix} = \begin{bmatrix} w^{\mathrm{x}}_{1,1} & w^{\mathrm{x}}_{1,2} & w^{\mathrm{x}}_{1,3} \\ w^{\mathrm{x}}_{2,1} & w^{\mathrm{x}}_{2,2} & w^{\mathrm{x}}_{2,3} \\ w^{\mathrm{x}}_{3,1} & w^{\mathrm{x}}_{3,2} & w^{\mathrm{x}}_{3,3} \end{bmatrix} * \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.34}$$

3.3.6 Exact Sequence Reconstruction (Decoding)

Each input embedding $\boldsymbol{e}_t = \boldsymbol{W}^x\,\boldsymbol{x}_t$ is stored (in reverse) in contiguous $\boldsymbol{D}$-sized blocks of the final state:

$$\widehat{\boldsymbol{h}}_T = \begin{bmatrix} \boldsymbol{h}_T \\ \sigma(\boldsymbol{h}_{T-1}) \\ \sigma^2(\boldsymbol{h}_{T-2}) \\ \vdots \\ \sigma^T(\boldsymbol{h}_0) \end{bmatrix}. \tag{3.35}$$

Since the $\boldsymbol{m}$th block $\widehat{\boldsymbol{h}}_{T,[(m-1)D+1:mD]} = \sigma^m(\boldsymbol{h}_{T-m})$ (with $m = T - k + 1$) contains all the nonlinearity, recovering $\boldsymbol{e}_k$ reduces to

$$\boldsymbol{e}_k = \underbrace{\sigma^{-1} \circ \sigma^{-1} \circ \cdots \circ \sigma^{-1}}_{m \text{ times}}\big(\widehat{\boldsymbol{h}}_{T,[(m-1)D+1:mD]}\big) \;-\; \boldsymbol{b}^h \;-\; \boldsymbol{b}^x, \tag{3.36}$$

and the discrete class $\boldsymbol{x}_k$ is then found by

$$\boldsymbol{x}_k = \arg\min_j \big\| \boldsymbol{e}_k - (\boldsymbol{W}^x)_{:,j} \big\|_2. \tag{3.37}$$

Decoding the input sequence from the final hidden state $\widehat{\boldsymbol{h}}_4 \in \mathbb{R}^{12}$ proceeds in four steps, using block size $\boldsymbol{D} = 3$:

1. **Recover $\boldsymbol{x}_4$.** Given the first block $\widehat{\boldsymbol{h}}_{4,1:3}$ and the second block $\widehat{\boldsymbol{h}}_{4,4:6}$, solve

$$\widehat{\boldsymbol{h}}_{4,1:3} = \sigma\big(\boldsymbol{W}^h\,\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,4:6}) + \boldsymbol{W}^x\,\boldsymbol{x}_4 + \boldsymbol{b}^h + \boldsymbol{b}^x\big) \tag{3.38}$$

   for $\boldsymbol{W}^x\,\boldsymbol{x}_4$. Since σ is invertible,

$$\boldsymbol{W}^x\,\boldsymbol{x}_4 = \sigma^{-1}(\widehat{\boldsymbol{h}}_{4,1:3}) - \boldsymbol{W}^h\,\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,4:6}) - \boldsymbol{b}^h - \boldsymbol{b}^x. \tag{3.39}$$

2. **Recover $\boldsymbol{x}_3$.** Using the second block $\widehat{\boldsymbol{h}}_{4,4:6}$ and the third block $\widehat{\boldsymbol{h}}_{4,7:9}$, solve

$$\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,4:6}) = \sigma\Big(\boldsymbol{W}^h\,\sigma^{-1}\big(\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,7:9})\big) + \boldsymbol{W}^x\,\boldsymbol{x}_3 + \boldsymbol{b}^h + \boldsymbol{b}^x\Big) \tag{3.40}$$

to get

$$\boldsymbol{W}^x\,\boldsymbol{x}_3 = \sigma^{-1}\big(\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,4:6})\big) - \boldsymbol{W}^h\,\sigma^{-1}\big(\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,7:9})\big) - \boldsymbol{b}^h - \boldsymbol{b}^x. \tag{3.41}$$

3. **Recover $\boldsymbol{x}_2$.** From the third block $\widehat{\boldsymbol{h}}_{4,7:9}$ and the fourth block $\widehat{\boldsymbol{h}}_{4,10:12}$, solve

$$\sigma^{-1}\big(\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,7:9})\big) = \sigma\Big(\boldsymbol{W}^h\,\sigma^{-1}\big(\sigma^{-1}(\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,10:12}))\big) + \boldsymbol{W}^x\,\boldsymbol{x}_2 + \boldsymbol{b}^h + \boldsymbol{b}^x\Big) \tag{3.42}$$

giving

$$\boldsymbol{W}^x\,\boldsymbol{x}_2 = \sigma^{-1}\big(\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,7:9})\big) - \boldsymbol{W}^h\,\sigma^{-1}\big(\sigma^{-1}(\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,10:12}))\big) - \boldsymbol{b}^h - \boldsymbol{b}^x. \tag{3.43}$$

4. **Recover $\boldsymbol{x}_1$.** Finally, using only the fourth block $\widehat{\boldsymbol{h}}_{4,10:12}$,

$$\sigma^{-1}\big(\sigma^{-1}(\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,10:12}))\big) = \sigma\big(\boldsymbol{W}^h\,\boldsymbol{h}_0 + \boldsymbol{W}^x\,\boldsymbol{x}_1 + \boldsymbol{b}^h + \boldsymbol{b}^x\big) \tag{3.44}$$

so that

$$\boldsymbol{W}^x\,\boldsymbol{x}_1 = \sigma^{-1}\big(\sigma^{-1}(\sigma^{-1}(\widehat{\boldsymbol{h}}_{4,10:12}))\big) - \boldsymbol{W}^h\,\boldsymbol{h}_0 - \boldsymbol{b}^h - \boldsymbol{b}^x. \tag{3.45}$$

$$\boldsymbol{W}^h = \begin{bmatrix} w^h_{1,1} & w^h_{1,2} & w^h_{1,3} \\ w^h_{2,1} & w^h_{2,2} & w^h_{2,3} \\ w^h_{3,1} & w^h_{3,2} & w^h_{3,3} \end{bmatrix}, \quad \boldsymbol{W}^x = \begin{bmatrix} w^x_{1,1} & w^x_{1,2} & w^x_{1,3} \\ w^x_{2,1} & w^x_{2,2} & w^x_{2,3} \\ w^x_{3,1} & w^x_{3,2} & w^x_{3,3} \end{bmatrix}, \quad \boldsymbol{b}^h = \begin{bmatrix} b^h_1 \\ b^h_2 \\ b^h_3 \end{bmatrix}, \quad \boldsymbol{b}^x = \begin{bmatrix} b^x_1 \\ b^x_2 \\ b^x_3 \end{bmatrix}. \tag{3.46}$$

3.3.7 Evaluation of Reconstruction Accuracy

Mean Squared Error

The Mean Squared Error (MSE) between the original and reconstructed embeddings is defined as

$$\text{MSE} = \frac{1}{\boldsymbol{B}\,\boldsymbol{T}\,\boldsymbol{D}} \sum_{b=1}^{\boldsymbol{B}} \sum_{t=1}^{\boldsymbol{T}} \big\|\,\boldsymbol{E}_{\text{enc}}[b,t,:] - \boldsymbol{E}_{\text{dec}}[b,t,:]\big\|^2, \tag{3.47}$$

Where:

$$\boldsymbol{E}_{\text{enc}}[b,t,:] = \boldsymbol{x}_t[b]^\mathsf{T}\,\boldsymbol{W}^x, \quad \boldsymbol{E}_{\text{dec}}[b,t,:]$$

is obtained by applying $\boldsymbol{\sigma}^{-1}$ and $(\boldsymbol{W}^h)^{-1}$ to the corresponding block of the final state. Here:

- $\boldsymbol{B}$ is the batch size,
- $\boldsymbol{T}$ is the sequence length,
- $\boldsymbol{D}$ is the embedding dimension,
- $\boldsymbol{x}_t[b] \in \{0,1\}^{\boldsymbol{K}}$ is the one-hot input vector,
- $\boldsymbol{W}^x \in \mathbb{R}^{\boldsymbol{D}\times\boldsymbol{K}}$ is the input-to-hidden weight matrix.

Interpreting the MSE

The MSE is normalised by $K \times \boldsymbol{D}$ to allow fair comparison across different class counts and embedding sizes. A near-zero MSE indicates that the RNN (state-scaled with structured parameters) has perfectly stored and recovered every input embedding; a larger MSE indicates information loss or distortion. By varying sequence length L, class count K, state size $\boldsymbol{D}$, activation function, and weight initialisation, the experiments determine under which conditions the architecture maintains perfect recall and when errors emerge.

Reconstruction Accuracy

Reconstruction accuracy is defined as the percentage of inputs correctly recovered:

$$\text{Accuracy} = \frac{1}{\boldsymbol{B}\boldsymbol{T}} \sum_{b=1}^{\boldsymbol{B}} \sum_{t=1}^{\boldsymbol{T}} \mathbf{1}\big(\hat{x}_t[b] = x_t[b]\big) \times 100\%, \tag{3.48}$$

where $\hat{\boldsymbol{x}}_{\boldsymbol{t}}[\boldsymbol{b}]$ is computed by $\hat{x}_t[b] = \arg\min_j \|\boldsymbol{e}_t - (\boldsymbol{W}^x)_{:,j}\|_2$, and $x_t[b]$ is the true class index. A 100% accuracy confirms that every discrete symbol was perfectly reconstructed.

3.3.8 Observations

Here are some key points to consider:

1. **Matrix Structure Alternatives:** The proof of concept uses a specific approach to stacking the modified matrices, where historical information is shifted and stored in blocks. While this implementation demonstrates the theoretical possibility of perfect recall, there could be alternative ways to structure these matrices to achieve the same goal. The main focus of our experiments is to determine at what sequence length this particular proof of concept implementation begins to degrade due to numerical issues and the specific factors that contribute to perfect recall, rather than comparing different matrix stacking strategies.

2. **Embedding Representations:** In this setup, discrete classes (e.g., $x(1), x(2), \ldots, x(K)$) are converted into dense vectors called embeddings before being processed by the RNN. These embeddings typically have a smaller dimension ($\boldsymbol{D_x}$) compared to the total number of classes (K), with $\boldsymbol{D}_x$ often ranging from hundreds to thousands. Although the RNN's state dimension ($\boldsymbol{D}_h$) doesn't need to match the embedding dimension ($\boldsymbol{D}_x$), they are often chosen to be the same.

3. **Invertibility of Activation Functions:** Not all activation functions or nonlinearities commonly used in RNNs are invertible, which can limit the ability to exactly reconstruct past states or inputs.

4. **Experiments:** The experiments included in Chapter 5, which analyse reconstruction error under varying conditions (e.g., sequence length, number of classes, and state size), will hope to confirm that the RNN (state-scaled with structured parameters) preserves the entire input sequence in its final state. This empirical evidence hopes to support the theoretical design, demonstrating that the proposed modified architecture in the proof of concept successfully mitigates the overwriting issue inherent in traditional RNNs.

3.4 Discussion: Numerical Stability and Scaling Issues

This section examines the numerical underflow challenges that occur when multiplying many small probabilities, a problem especially common in speech recognition. It is then explained how the RNN (state-scaled with structured parameters) avoids this pitfall by using invertible activation functions and sparse state transition matrices, thereby preserving numerical stability over long sequences.

3.4.1 The Challenge of Multiplying Small Probabilities

In many speech recognition systems, the overall likelihood of a long sequence is computed as the product of many small probabilities. Since each probability is a value between 0 and 1, multiplying thousands of them can result in extremely small numbers, eventually leading to numerical underflow [6]. This problem is particularly severe in systems based on hidden Markov models or similar probabilistic frameworks, where such multiplication is inherent to the model. To overcome this, speech recognition models often switch to log probabilities, converting multiplication into addition to maintain numerical stability [37].

It is important to note that the challenge of multiplying small probabilities is more predominant in the domain of speech recognition due to its reliance on probabilistic modelling of sequential data [37]. In contrast, other fields—such as computer vision, natural language processing, or even many deep learning architectures—do not involve the explicit repeated multiplication of probabilities. Instead, these systems rely on alternative numerical operations (e.g., convolutions or attention mechanisms) that do not inherently suffer from the same underflow issues [17].

3.4.2 Our RNN (state-scaled with structured parameters) Approach

In contrast, the RNN (state-scaled with structured parameters) avoids the pitfalls associated with multiplying small probability values. Rather than relying on this multiplicative process, it uses invertible activation functions and sparse state transition matrices that shift information blockwise. This design ensures that the state does not undergo the cumulative compression that occurs in standard RNNs—where repeated multiplication can lead to overwriting of state information. By allocating distinct blocks to each input, the modified architecture preserves numerical stability over long sequences. This design choice demonstrates that the typical numerical instability (step 2) is effectively mitigated in the RNN (state-scaled with structured parameters), even though alternative scaling methods remain necessary in domains where probability multiplication is unavoidable. Consequently, while scaling techniques like log probabilities are essential in speech recognition, they are generally not required in other domains, further emphasising the robustness of our approach.

3.4.3 Clamping Activation Functions

Although invertible activations (e.g.tanh/atanh) guarantee exact reversal in theory, in practice their outputs can approach machine-precision limits and incur numerical error when iterated many times. To prevent this, the RNN (state-scaled with structured parameters) architecture clamps each activation's output to a safe interval $[-\boldsymbol{c}, \boldsymbol{c}]$ with $\boldsymbol{c} < 1$. Concretely, after computing

$$\boldsymbol{u}_t = \boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{b}, \tag{3.49}$$

the update becomes

$$\boldsymbol{h}_t = \text{Clamp}\big(\mathbf{tanh}(\boldsymbol{u}_t), -\boldsymbol{c}, \boldsymbol{c}\big), \tag{3.50}$$

where

$$\text{Clamp}(\tanh(\boldsymbol{u}_t), -c, +c) = \min\big(\max(\tanh(\boldsymbol{u}_t), -c), +c\big).$$

By selecting $c = 1 - \epsilon$ (e.g.$\epsilon = 10^{-6}$), activation outputs are prevented from reaching ± 1, which would return an undefined value for the inversion.

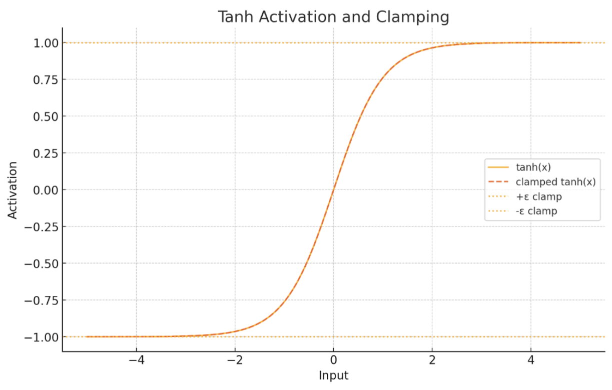


Figure 3.1: Tanh being clamped to avoid undefined values.

3.5 Discussion: Compression and Overwriting

This section examines how linear transformations in standard RNNs can overwrite historical information, limiting their ability to compress sequential data effectively. It then describes the RNN's (state-scaled with structured parameters) blockwise storage approach, highlights the trade-offs between redundancy and compression, and discusses open research questions related to achieving lossless compression without increasing the state size.

3.5.1 Overwriting in Standard RNNs

A common challenge in standard RNNs is that the repeated application of the same linear transformation and nonlinearity tends to compress—or even overwrite—earlier state information. Over many time steps, information from previous inputs is gradually "squashed" into a smaller representation, making it difficult for RNNs to act as effective compressors of sequential data. Consequently, vital details may be lost over long dependencies (see subsection 3.3.3).

3.5.2 Blockwise Storage in the RNN (state-scaled with structured parameters)

The RNN (state-scaled with structured parameters) preserves each input by assigning it a dedicated block within the state vector (see equation (3.31)). Sparse state-transition matrices shift blocks forward at each time step rather than mixing them into a single vector. This blockwise mechanism prevents cumulative overwriting and ensures that each input remains recoverable in its own slot.

3.5.3 Redundancy Versus True Compression

While blockwise storage guarantees exact recall, it incurs *linear* memory growth in the sequence length L. In many sequence-modelling applications—such as language modelling or time-series forecasting—the objective is to compress long inputs into a compact representation without discarding critical information. By contrast, naïve linear compression often leads to vanishing gradients and lost information [6]. The RNN's (state-scaled with structured parameters) design trades memory efficiency for recall accuracy: redundancy is introduced to avoid the pitfalls of standard RNN compression.

3.5.4 Memory Efficiency in the Extended RNN Implementation

Scaling experiments to extremely long sequences (e.g. up to several million tokens) on high-performance computing (HPC) systems necessitates careful memory management. In a traditional RNN, the forward pass stores the state at each time step. For a sequence of length L, this requires retaining L intermediate states, which can demand on the order of gigabytes of RAM and thus become impractical.

Proposed Memory-Efficient Method:

To mitigate this, the RNN's (state-scaled with structured parameters) forward pass was re-implemented to retain only the final state. At each time step, the state vector is updated in place, and intermediate outputs are discarded. Although direct access to every intermediate state is lost, the RNN's (state-scaled with structured parameters) blockwise encoding ensures that the final state encapsulates all the inputs but in reverse order.

3.5.5 Future Work on Smart Compression

While the in-place RNN (state-scaled with structured parameters) reduces memory usage, its linear growth with L remains a limitation. Future research will explore smart compression strategies that preserve invertibility with sublinear memory cost, including:

– **Intra-block factorisation:** applying low-rank approximations or quantisation within each block post-forward pass, so that similar blocks share parameters or can be merged [34].

– **Adaptive block allocation:** creating new blocks only when incoming inputs exceed a threshold (e.g. based on reconstruction error), allowing for storage to be skipped for redundant tokens [19].

– **Hierarchical block hierarchies:** organising blocks at multiple temporal scales—small blocks for fine details and larger blocks for coarse context—to balance memory footprint against information retention [10].

Investigating these avenues will require new metrics that jointly assess memory usage, recall fidelity, and computational overhead, showing the way toward truly lossless yet efficient sequence compression.

3.6 Theoretical Guarantee and Considerations

This section provides a theoretical guarantee for the proof of concept as well as a discussion on any considerations that need to be noted.

3.6.1 Theoretical Guarantees

[Exact Sequence Recall]
Let $\{x_1, \ldots, x_T\}$ be an arbitrary input sequence of real numbers, and let the RNN (state-scaled with structured parameters) satisfy:

1. $\sigma : \mathbb{R} \to (-c, c)$ is an invertible activation (e.g. tanh) clamped to $(-c, c)$ with $0 < c < 1$.
2. All weight matrices $\boldsymbol{W}_h$ and $\boldsymbol{W}_x$ are invertible.
3. The state size $= \boldsymbol{D} \times L_{max}$.

Then there exists an explicit inverse mapping that recovers each x_t uniquely from the final state $\boldsymbol{h}_T$.

3.6.2 Ethical, Legal, and Professional Considerations

No special ethical or legal concerns are expected to arise from this project. The research will be conducted in accordance with BCS professional principles.

Chapter 4

Implementation

This chapter describes the code implementation of the proof-of-concept, including the experiments that are run. This code was developed in conjunction with the help of my supervisor; clear credit acknowledgements are provided in the code files.

4.1 PyTorch Overview

PyTorch is an open-source library for tensor computations and deep learning. It offers dynamic computation graphs and built-in automatic differentiation, making model testing and debugging straightforward. Key components used in this project include:

– **Tensors**: multi-dimensional arrays that can be used with NumPy-style operations (tensors are described in more detail in section 2.2).

– **nn.Module**: Base class for neural networks; custom layers and models use a forward() method that mimics the forward pass.

– **Buffers**: Tensors registered with register_buffer(), used here to store RNN (state-scaled with structured parameters) state without preserving intermediate states for memory optimisation.

PyTorch was chosen as it allows users to define custom recurrent layers (such as the RNN (state-scaled with structured parameters)) as well as manage persistent buffers.

4.2 The Classes

– **Main**: Coordinates the entire workflow—instantiates DataGenerator, MemoryModel and Evaluator, drives data generation, encoding, decoding and error computation.

– **IRNN (RNN state-scaled with structured parameters)**: Implements the extended buffer of size $[L \times \boldsymbol{D}]$ per layer, shifts and prepends each new state to enable exact reconstruction of past states via inverse-tanh operations.

***Note:* The IRNN is the class used to build the RNN (state-scaled with structured parameters)**

- **DataGenerator**: Produces batches of random integer sequences and their one-hot encodings of shape (B, L, K) for the experiments.
- **MemoryModel**: Wraps a standard PyTorch RNN and its IRNN extension; provides the encode() function to obtain standard RNN states and buffer for the RNN (state-scaled with structured parameters) states. The function decode() is used to reconstruct the sequence of states.
- **Evaluator**: Recovers input embeddings from reconstructed states, computes per-sequence and per-embedding mean squared errors, and contains the experiment methods.

4.2.1 Class Diagram

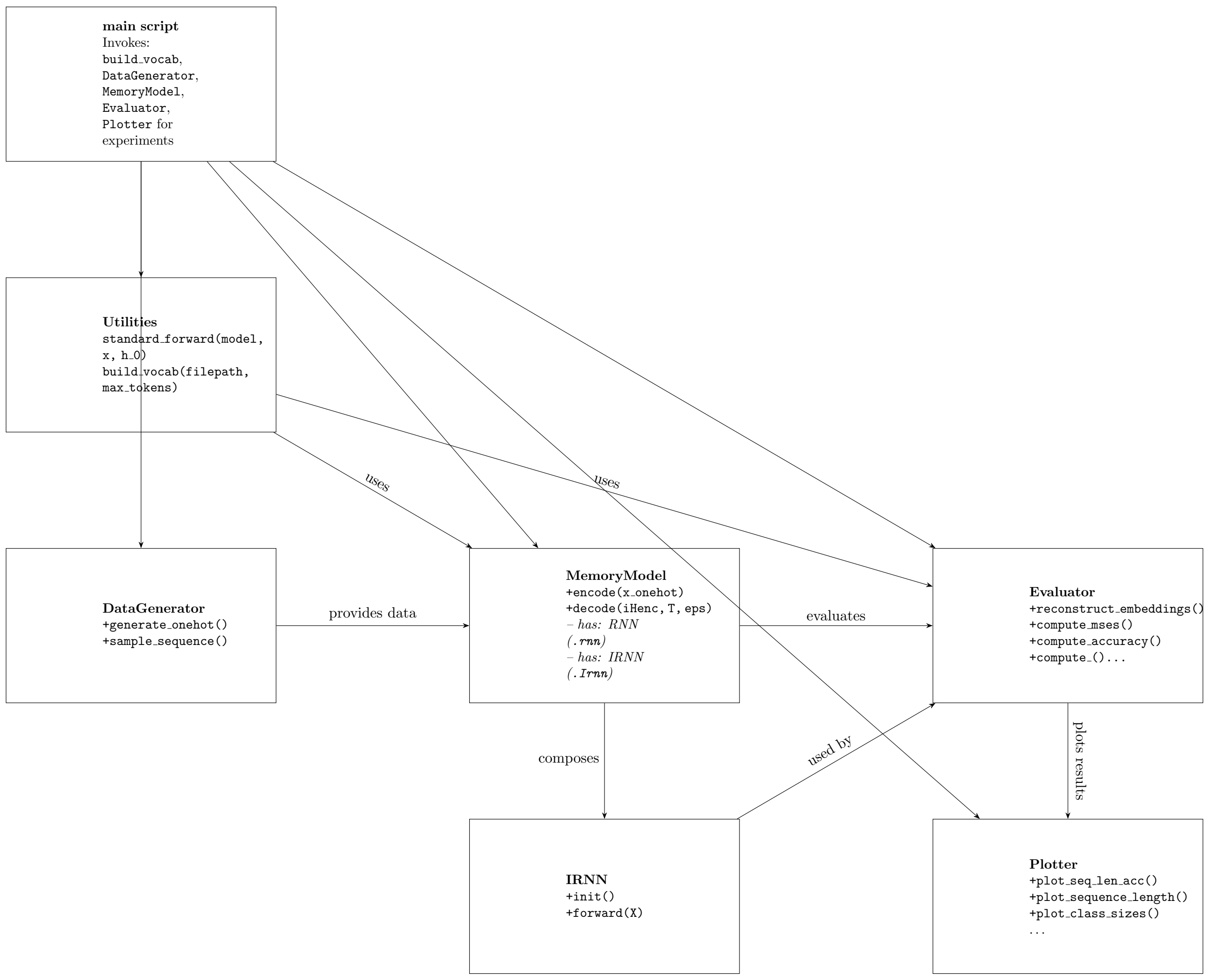


Figure 4.1: Class diagram that matches the code

4.3 Proof-of-Concept Implementation

This section walks through the key stages of turning the RNN (state-scaled with structured parameters) proof of concept into working PyTorch code: how the RNN (state-scaled with structured parameters) is initialised, how encoding and decoding are performed, how embeddings are recovered, and how metrics and experiments are organised.

4.3.1 RNN (state-scaled with structured parameters) Initialisation and Buffer Management

The IRNN class wraps a standard `torch.nn.RNN` and registers its parameters as buffers. Two rolling buffers, $\boldsymbol{H}_{t-1}$ and $\boldsymbol{H}_t$, each of shape [layers $\times$ B $\times$ $(L_{\max} \cdot D)$], maintain the entire history without storing per-step activations on the autograd tape:

```python
class IRNN(torch.nn.Module):
    def __init__(self, base_rnn, L_max, B,
                 activation=torch.tanh,
                 activation_inv=torch.atanh):
        super().__init__()
        self.max_length = L_max
        self.batch_size = B
        self.hidden_size = base_rnn.hidden_size
        self.num_layers = base_rnn.num_layers
        self.act        = activation
        self.act_inv    = activation_inv

        # two flat buffers: [layers, B, L_max*D]
        self.register_buffer('Ht1',
            torch.zeros(self.num_layers, B, L_max * self.hidden_size))
        self.register_buffer('Ht',
            torch.zeros_like(self.Ht1))

        # copy base RNN weights & biases as buffers
        for l in range(self.num_layers):
            w_ih = getattr(base_rnn, f'weight_ih_l{l}').T
            w_hh = getattr(base_rnn, f'weight_hh_l{l}').T
            b_ih = getattr(base_rnn, f'bias_ih_l{l}')
            b_hh = getattr(base_rnn, f'bias_hh_l{l}')
            self.register_buffer(f'weight_ih_{l}', w_ih)
            self.register_buffer(f'weight_hh_{l}', w_hh)
            self.register_buffer(f'bias_ih_{l}', b_ih)
            self.register_buffer(f'bias_hh_{l}', b_hh)
```

Listing 4.1: IRNN __init__ and buffer setup

Because only the two buffers $\boldsymbol{H}_{t-1}$, $\boldsymbol{H}_t$ are stored, memory grows as $\mathcal{O}(B\, L_{\max}\, D)$, not $\mathcal{O}(B\, T\, D)$ during a forward pass of length T.

4.3.2 Encoding: Standard RNN & IRNN Forward Pass

Encoding uses both the standard RNN forward function (to produce reference states $\boldsymbol{H}_{\text{enc}}$) and the IRNN forward function (to fill its flat buffer $\boldsymbol{iH}_{\text{enc}}$):

```python
def standard_forward(model, x_onehot, h0=None):
    # returns (Henc: [ BTD ], hT)
    ...
    return torch.stack(outputs, dim=1), h_t

# in MemoryModel.encode:
Henc, _   = standard_forward(self.rnn, x_onehot)
iHenc     = self.Irnn(x_onehot) # returns [ layersB ( L_maxD )]
```

4.3.3 Decoding: Unrolling the IRNN Buffer

Decoding reconstructs the per-time-step states $\boldsymbol{iH}_{\text{dec}} \in \mathbb{R}^{B\times T\times D}$ by repeatedly peeling off the first $\boldsymbol{D}$ slots and applying the inverse activation:

```python
def decode(self, iHenc, T=None, eps=1e-6):
    if T is None: T = self.Irnn.max_length
    B, D = iHenc.shape[1], self.rnn.hidden_size
    buf  = iHenc[-1]             # shape: [B, L_maxD]
    iHdec = torch.zeros(B, T, D)
    for t in range(T):
        iHdec[:, T-1-t, :] = buf[:, :D]
        # clamp to (-1+eps,1-eps) before inverse
        buf = self.Irnn.act_inv(buf[:, D:].clamp(-1+eps,1-eps))
    return iHdec
```

Listing 4.2: MemoryModel.decode

4.3.4 Embedding Recovery

Once per-step states are reconstructed, embeddings are recovered by inverting $\boldsymbol{h}_t = \tanh(\boldsymbol{x}_t W^x + \boldsymbol{h}_{t-1} W^h + b^x + b^h)$:

```python
def reconstruct_embeddings(self, x_onehot, iHdec):
    B, L, D = iHdec.shape
    iEdec   = torch.zeros(B, L, D)
    Eenc    = torch.zeros_like(iEdec)
    prev_h  = torch.zeros(B, D)
    for t in range(L):
        h_t     = iHdec[:, t]
        # invert tanh then subtract recurrent contributions
        e_t     = self.model.Irnn.act_inv(h_t) \
                - prev_h @ self.model.rnn.weight_hh_l0.T \
                - self.model.rnn.bias_hh_l0 \
               - self.model.rnn.bias_ih_l0
        iEdec[:, t] = e_t
        Eenc[:, t] = x_onehot[:, t] @ self.model.rnn.weight_ih_l0.T
        prev_h     = h_t
    return iEdec, Eenc
```

Listing 4.3: Evaluator.reconstruct_embeddings

4.4 Metrics: MSE and Accuracy

Two primary metrics quantify reconstruction quality:

– **State MSE:** MSE($\boldsymbol{H}_{\text{enc}}, \boldsymbol{iH}_{\text{dec}}$), computed as

```
    state_mse = F.mse_loss(Henc, iHdec).item()
```

Listing 4.4: Evaluator.compute_mse

– **Accuracy:** nearest-neighbour classification in embedding space using euclidean distance,

```
  # compute accuracy of embedding reconstruction using Euclidean distance
def compute_accuracy(self, iEdec, x, model=None, per_sequence=False):
  with torch.no_grad():
      if model is None:
          model = self.model
      B, L = iEdec.size(0), iEdec.size(1)
      P = []
      # weights: [K, D]
      W = model.Irnn.weight_ih
      if W.dim() == 3:
          W = W[0]

      for t in range(L):
          # [ B1D ] vs [ BKD ]
          d = torch.cdist(
              iEdec[:, t].unsqueeze(1),
              W.unsqueeze(0).expand(B, -1, -1),
              p=2
          )
          P.append(torch.argmin(d, dim=2))
      # P: list of [B1]     cat     [ BL ]
      P = torch.cat(P, dim=1)
      # compare to x ([BL])     mean over t
      acc_per_seq = (P == x).float().mean(dim=1) * 100 # [B]
      if per_sequence:
          return acc_per_seq
      else:
          return acc_per_seq.mean().item()
```

Listing 4.5: Evaluator.compute_accuracy

4.5 Experimental Suite

The `Evaluator` class organises a variety of experiments, each of which sweeps a single hyperparameter (the parameter that is being tested):

`compute_mse_vs_sequence_length` Sequence lengths $\{10, 50, \ldots, 10^4\}$.
`compute_error_vs_inversion_depth` Inversion depths $\{0, 10, \ldots, T-1\}$.
`compute_mse_vs_class_sizes` Number of input classes K.
`compute_mse_vs_state_sizes` State sizes D.
`compute_mse_vs_activations`† Various activation functions.
`compute_mse_vs_initializations` Different weight-initialisation schemes.
`compute_semantic_clustering` Semantic clustering on real text.

Experiments are run over three different seeds to ensure that the results are reproducible and to also compute the standard deviations and means for plotting purposes. Results are visualised using the `Plotter` class. For example, to plot MSE vs. sequence length:

```
lengths, st_means, st_stds, em_means, em_stds, acc_means, acc_stds = \
    evaluator.compute_mse_vs_sequence_length(batch_size, seeds)
plotter.plot_sequence_length(
    lengths, st_means, em_means,
    st_stds, em_stds,
    B, D, K
)
```

Listing 4.6: Visualising Sequence–Length Results

4.5.1 MSE vs Sequence Length

This experiment measures how state and embedding reconstruction MSEs change as the input sequence length L increases. Answering the key question: How well can we recover the original input for sequence length L?

```
for L in lengths:
        print(f"  ␣testing␣L={L}")
        def _one_seed(seed):
            # reseed
            random.seed(seed)
            np.random.seed(seed)
            torch.manual_seed(seed)
            with torch.no_grad():
                # build + run your batch
                gen   = DataGenerator(K, L, D, B=batch_size)
                model = MemoryModel(K, D, L, B=batch_size)
                x, x1hot    = gen.generate_onehot()          # [B, L], [B, L, K]
                Henc, _     = standard_forward(model.rnn, x1hot)
                iHenc       = model.Irnn(x1hot)
                iHdec       = model.decode(iHenc)
                iEdec, Eenc = self.reconstruct_embeddings(x1hot, iHdec, model=model)
            # compute your three scalars
            state_mse = ((Henc - iHdec)**2).mean().item()
            embed_mse = ((Eenc - iEdec)**2).mean().item()
            acc       = self.compute_accuracy(iEdec, x, model=model)
            print(f"State Mse for {L} is: {state_mse}")
            # cleanup
            del gen, model, x, x1hot, Henc, iHenc, iHdec, iEdec, Eenc
            torch.cuda.empty_cache()
            return state_mse, embed_mse, acc
)
```

Listing 4.7: compute_mse_vs_sequence_length

4.5.2 Error vs Inversion Depth

This experiment evaluates reconstruction error when decoding only up to a given "inversion depth" d, i.e. how far back the IRNN can invert reliably. Answering the key question: Does information from 10 steps ago get recalled better than information from 1000 steps ago?

```
for d in depths:
            print(f"Testing depth: {d}")
            idx = L - 1 - d
            h_t, h_r = Henc [:,idx,:], iHdec[:,idx,:]
            e_t, e_r = Eenc [:,idx,:], iEdec[:,idx,:]

            mse_s = F.mse_loss(h_t, h_r, reduction="mean").item()
            mse_e = F.mse_loss(e_t, e_r, reduction="mean").item()

            # just one distancebased accuracy at that depth:
            dists = torch.cdist(
                e_r.unsqueeze(1),
                W.unsqueeze(0).expand(batch_size,-1,-1),
                p=2
            )[:,0,:]
            preds = dists.argmin(dim=1)
            acc   = (preds == x[:,idx]).float().mean().item()*100

            all_s[d].append(mse_s)
            all_e[d].append(mse_e)
            all_a[d].append(acc)

        del gen, model, x, x1hot, Henc, iHenc, iHdec, iEdec, Eenc
```

Listing 4.8: compute_error_vs_inversion_depth

4.5.3 MSE vs Class Sizes

This sweep varies the number of unique input classes K and measures reconstruction MSE and accuracy. Answering the key question: does increasing the number of distinct classes lead to higher MSE or drop in accuracy?

```python
for K in class_sizes:
        print(f"  ␣testing␣K={K}")
        def _one_seed(seed):
            random.seed(seed)
            np.random.seed(seed)
            torch.manual_seed(seed)
            with torch.no_grad():
                gen   = DataGenerator(K, L, D, B=batch_size)
                model = MemoryModel(K, D, L, B=batch_size)
                x, x1hot   = gen.generate_onehot()
                Henc, _    = standard_forward(model.rnn, x1hot)
                iHenc      = model.Irnn(x1hot)
                iHdec      = model.decode(iHenc)
                iEdec, Eenc = self.reconstruct_embeddings(x1hot, iHdec, model=model)
            state_mse = ((Henc - iHdec)**2).mean().item()
            embed_mse = ((Eenc - iEdec)**2).mean().item()
            acc       = self.compute_accuracy(iEdec, x, model=model)
            # cleanup
            del gen, model, x, x1hot, Henc, iHenc, iHdec, iEdec, Eenc
            if torch.cuda.is_available():
                torch.cuda.empty_cache()
            return state_mse, embed_mse, acc

            # parallelise, store and return lists like the experiments before...
```

Listing 4.9: compute_mse_vs_class_sizes

4.5.4 MSE vs State Sizes

This experiment varies the state dimension D to see its effect on reconstruction quality. Answering the key question: does increasing state size yield lower MSE and improved accuracy, and where does it plateau?

```python
for D in state_sizes:
        print(f"  ␣testing␣D={D}")
        def _one_seed(seed):
            random.seed(seed)
            np.random.seed(seed)
            torch.manual_seed(seed)
            with torch.no_grad():
                gen   = DataGenerator(K, L, D, B=batch_size)
                model = MemoryModel(K, D, L, B=batch_size)
                x, x1hot   = gen.generate_onehot()
                Henc, _    = standard_forward(model.rnn, x1hot)
                iHenc      = model.Irnn(x1hot)
                iHdec      = model.decode(iHenc)
                iEdec, Eenc = self.reconstruct_embeddings(x1hot, iHdec, model=model)
            # scalars
            state_mse = F.mse_loss(Henc, iHdec).item()
            embed_mse = F.mse_loss(Eenc, iEdec).item()
            acc      = self.compute_accuracy(iEdec, x, model=model)
            # cleanup
            del gen, model, x, x1hot, Henc, iHenc, iHdec, iEdec, Eenc
            if torch.cuda.is_available(): torch.cuda.empty_cache()
            return state_mse, embed_mse, acc

            # parallelise, store and return lists...
```

Listing 4.10: compute_mse_vs_state_sizes

4.5.5 MSE vs Activation Functions

This sweep tests different activation/inversion pairs (e.g. tanh, σ, ELU, ReLU). Answering the key question: which functions degrade performance due to non-invertibility or saturating behaviour?

```
eps = 1e-6

    # inverse functions
    def inv_tanh(y):
        return torch.atanh(y.clamp(min=-1+eps, max=1-eps))
    def inv_sigmoid(y):
        return torch.logit(y, eps=eps)
    def inv_elu(y):
        return torch.where(y > 0, y, torch.log1p(y.clamp(min=-1+eps)))

    # default activation set
    if activations is None:
        activations = {
            "tanh":     (torch.tanh,         inv_tanh),
            "sigmoid":  (torch.sigmoid,      inv_sigmoid),
            "relu":     (torch.relu,         lambda y: y),
            "leaky_relu":(lambda x: F.leaky_relu(x,0.01),
                         lambda y: torch.where(y>0,y,y/0.01)),
            "elu":      (lambda x: F.elu(x,1.0), inv_elu),
        }

    K, L = self.model.K, self.model.L
    names = []
    state_means, state_stds = [], []
    embed_means, embed_stds = [], []
    acc_means,  acc_stds  = [], []

    for name, (act, inv) in activations.items():
        print(f"  testing activation = {name}")
        names.append(name)
        def _one_seed(seed):
            random.seed(seed)
            np.random.seed(seed)
            torch.manual_seed(seed)
            with torch.no_grad():
                # build model + override its IRNN
                gen   = DataGenerator(K, L, self.model.D, B=batch_size)
                model = MemoryModel(K, D, L, B)
                irnn  = IRNN(model.rnn,
                            L_max=L,
                            batch_size=B,
                            activation=act,
                            activation_inv=inv)
                model.Irnn = irnn
                x, x1hot   = gen.generate_onehot()
                Henc, _    = standard_forward(model.rnn, x1hot)
                iHenc      = model.Irnn(x1hot)
                iHdec      = model.decode(iHenc)
                iEdec, Eenc = self.reconstruct_embeddings(x1hot, iHdec, model=model)

            state_mse = F.mse_loss(Henc, iHdec).item()
            embed_mse = F.mse_loss(Eenc, iEdec).item()
            acc      = self.compute_accuracy(iEdec, x, model=model)
            # cleanup
            del gen, model, x, x1hot, Henc, iHenc, iHdec, iEdec, Eenc
            if torch.cuda.is_available(): torch.cuda.empty_cache()
            return state_mse, embed_mse, acc

        # parallelise, store and return lists...
```

Listing 4.11: compute_mse_vs_activations

4.5.6 MSE vs Initialisation Schemes

This experiment compares reconstruction under different weight-initialisation methods (Xavier, Kaiming, etc.). Answering the key question: Does the choice of weight-initialisation scheme significantly affect the RNN's ability to reconstruct sequences?

```
if init_methods is None:
        init_methods = [
            "default",
            "xavier_uniform",
            "xavier_normal",
            "kaiming_uniform",
            "kaiming_normal",
        ]

    K, D, L, B = self.model.K, self.model.D, self.model.L, batch_size

    names = []
    state_means, state_stds = [], []
    embed_means, embed_stds = [], []
    acc_means,  acc_stds  = [], []

    for name in init_methods:
        print(f"  ␣testing␣init␣=␣{name}")
        names.append(name)

        def _one_seed(seed):
            # re-seed everything
            random.seed(seed)
            np.random.seed(seed)
            torch.manual_seed(seed)
            # start from fresh base model
            base = copy.deepcopy(self.model.rnn)

            # apply the desired init
            if name == "xavier_uniform":
                init.xavier_uniform_(base.weight_ih_l0)
                init.xavier_uniform_(base.weight_hh_l0)
            elif name == "xavier_normal":
                init.xavier_normal_(base.weight_ih_l0)
                init.xavier_normal_(base.weight_hh_l0)
            elif name == "kaiming_uniform":
                init.kaiming_uniform_(base.weight_ih_l0, nonlinearity="tanh")
                init.kaiming_uniform_(base.weight_hh_l0, nonlinearity="tanh")
            elif name == "kaiming_normal":
                init.kaiming_normal_(base.weight_ih_l0, nonlinearity="tanh")
                init.kaiming_normal_(base.weight_hh_l0, nonlinearity="tanh")
            #   default   just leaves base as-is

            # build IRNN + full model
            temp_irnn = IRNN(base, L, batch_size=B,
                           activation=self.model.Irnn.act,
                           activation_inv=self.model.Irnn.act_inv)
            temp_model = MemoryModel(K, D, L, B)
            temp_model.rnn = base
            temp_model.Irnn = temp_irnn

            # one random batch
            x, x1hot = self.data_gen.generate_onehot()
            Henc, _ = standard_forward(base, x1hot)
            iHenc   = temp_irnn(x1hot)
            iHdec   = temp_model.decode(iHenc)
```

```
            iEdec, Eenc = self.reconstruct_embeddings(x1hot, iHdec, model=temp_model)

            # metrics
            s = F.mse_loss(Henc, iHdec).item()
            e = F.mse_loss(Eenc, iEdec).item()
            a = self.compute_accuracy(iEdec, x, model=temp_model)

            # clean up GPU memory
            del base, temp_irnn, temp_model, x, x1hot, Henc, iHenc, iHdec, iEdec, Eenc
            torch.cuda.empty_cache()
            return s, e, a

            # parallelise, store and return...
```

Listing 4.12: compute_mse_vs_initializations

4.5.7 Semantic Clustering

This experiment performs clustering on real text embeddings, comparing silhouette scores before and after IRNN reconstruction. Answering the key question: are silhouette scores unchanged before and after reconstruction, indicating intact cluster structure?

```
        batch.append(seq)
        if len(batch) == batch_seqs:
            idxs    = [tokens_to_idx(s) for s in batch]
            x_idx   = torch.tensor(idxs, dtype=torch.long, device=device)
            # Ensure K_text (num_classes for one_hot) is correct
            x_onehot = F.one_hot(x_idx, num_classes=self.model.K).float().to(device)

            Henc, _   = standard_forward(self.model.rnn, x_onehot)
            iHenc     = irnn_batch(x_onehot) # Ensure IRNN is reset or handles state across calls
            iHdec     = self.model.decode(iHenc, T=x_onehot.size(1)) # Pass T explicitly

            # Pass model explicitly if reconstruct_embeddings might use a different one
            iEdec, Eenc = self.reconstruct_embeddings(x_onehot, iHdec, model=self.model)

            B0, L0, D0 = iEdec.shape # L0 here is seq_len
            Etrue = Eenc.reshape(-1, D0).cpu().numpy()
            Erec  = iEdec.reshape(-1, D0).cpu().numpy()

            mbk_true.partial_fit(Etrue)
            mbk_rec.partial_fit(Erec)

            # Improved sampling logic
            if total_samples_collected < sample_size:
                num_from_this_batch = Etrue.shape[0] # Current batch provides B0 * L0 embeddings
                needed = sample_size - total_samples_collected
                to_take = min(num_from_this_batch, needed)

                if to_take > 0:
                    idx_samp = np.random.choice(num_from_this_batch, to_take, replace=False)
                    samples_true_list.append(Etrue[idx_samp])
                    samples_rec_list.append(Erec[idx_samp])
                    total_samples_collected += to_take

            del idxs, x_idx, x_onehot, Henc, iHenc, iHdec, iEdec, Eenc, Etrue, Erec
            irnn_batch.Ht1.zero_() # Reset IRNN state
            irnn_batch.Ht.zero_()
            if torch.cuda.is_available():
                torch.cuda.empty_cache()
            batch = []

    if not samples_true_list or total_samples_collected == 0:
    # Check if any samples were collected
        print("Warning:␣No␣samples␣collected
␣␣␣␣␣␣␣␣for␣silhouette␣score␣calculation.␣Returning␣NaN.")
        return float('nan'), float('nan')

    sample_true = np.vstack(samples_true_list)
    sample_rec = np.vstack(samples_rec_list)

    sil_true = float('nan')
    sil_rec = float('nan')

    # Compute silhouette score for true and reconstructed embeddings
    return sil_true, sil_rec
```

Listing 4.13: compute_semantic_clustering

4.6 Remarks

By separating the IRNN buffers, encoding/decoding logic, embedding inversion, metric computation, and plotting into distinct, reusable sections, the proof-of-concept remains clear, memory-efficient, and easy to navigate to the experiments. Specific design choices were also made to ensure smooth processing speeds which are detailed in the following subsection.

4.6.1 Design Choices

Several key design choices were made in the implementation:

1. Use of PyTorch's built-in RNN as the base model, allowing for efficient computation.
2. Explicit expansion of state vectors rather than using a recursive formulation, making the information storage more transparent.
3. Use of batch processing capabilities. Increased accuracy of results for experiments.

4. An initial attempt was made to parallelise the encode/decode loops on GPU to leverage high throughput and reduce memory transfers. However, because these loops are written as Python for–loops, each time-step created a separate kernel launch, creating latency that outweighed any GPU speed-up. Empirical profiling showed that running the loops on CPU with ThreadPoolExecutor threads was actually faster and more memory-efficient for the batch-size-1 experiments.
5. The seeds were parallelised to save processing time for each experiment
6. After every run of a loop in an experiment, unneeded attributes and tensors would be deleted to save memory, as they would be massive tensors that were still being stored by PyTorch even though they weren't needed for plotting anymore.

The next chapter introduces the results gained from running these experiments.

Chapter 5

Results

5.1 Parameter Selection

For all experiments in this dissertation, the RNN's state dimension $\boldsymbol{D}$ was fixed to 128. A dimension of 128 units is commonly used in RNN research as a compromise between state capacity and computational efficiency [17].

The sequence length L was set to 100 000 time steps to create a memory-intensive workload. This length considerably exceeds the hundreds-to-thousands-of-steps range typical in the literature, yet it is still able to run with the help of the Sheffield Stanage high-performance computing nodes (HPC). For experiments that weren't testing variable sequence length, L was set to 10 000 to balance memory and time constraints whilst using the HPC.

Batch sizes were set to 1 for every experiment except in semantic clustering. Instead, variability was measured through repeated runs under different random seeds. Each experimental configuration was executed with three random seeds to compute the mean and standard deviation, ensuring that error bars reflect variance.

5.2 Empirical Analysis

In this section, I will be showcasing my results and briefly commenting on obvious trends while leaving more in-depth conclusions for Chapter 6.

5.2.1 MSE and Accuracy vs Sequence Length

This experiment measures how state and embedding reconstruction MSEs, as well as reconstruction accuracy, change as the input sequence length $\boldsymbol{L}$ increases.

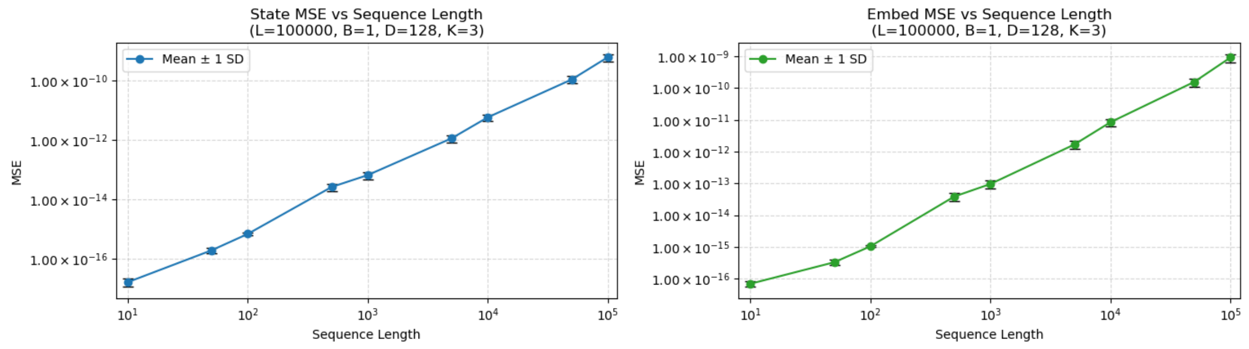


Figure 5.1: Mean ± 1 SD of state and embedding MSE vs sequence length L.

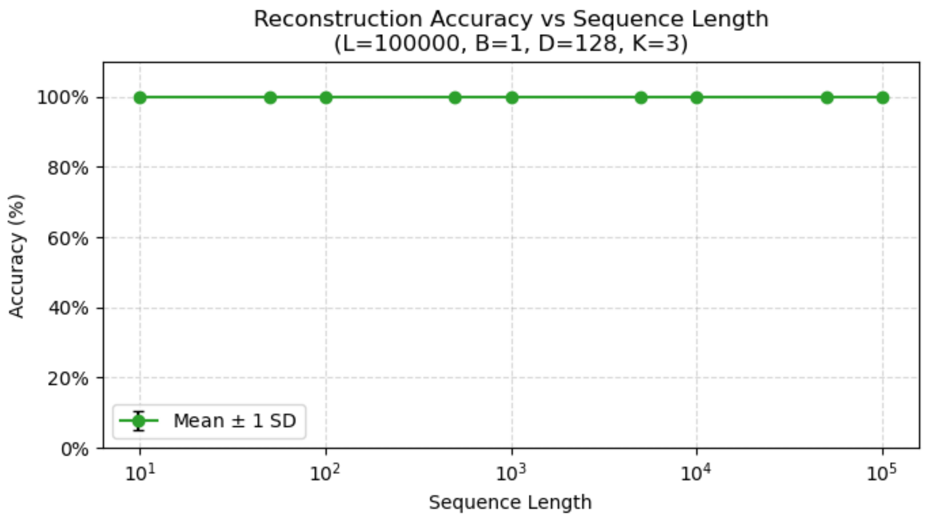


Figure 5.2: Mean ± 1 SD of reconstruction accuracy vs sequence length L.

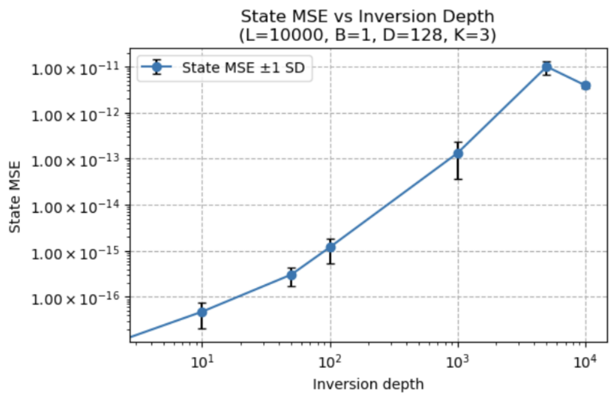
Discussion

Figure 5.1 shows that both the state-level and embedding-level mean squared errors remain extremely low—on the order of 10^{-16} for $L = 10$ rising to 10^{-10}–10^{-9} for $L = 10^5$. The roughly linear increase on the log–log plot reflects the gradual accumulation of floating-point rounding error (numerical drift) as the sequence lengthens, but even at $L = 10^5$ the errors stay far below any practical threshold.

Meanwhile, Figure 5.2 demonstrates perfect (100 %) reconstruction accuracy across all tested lengths, with negligible variance. Together, these results confirm that the proposed structured, state-scaled RNN can encode and decode entire discrete input sequences of up to 10^5 steps exactly—subject only to minor numerical drift—and that neither non-linearity nor sequence length intrinsically limits its recall capacity.

5.2.2 Error vs Inversion Depth

This experiment evaluates reconstruction error when decoding only up to a given "inversion depth" d, i.e. how far back the RNN (state-scaled with structured parameters) can invert reliably.



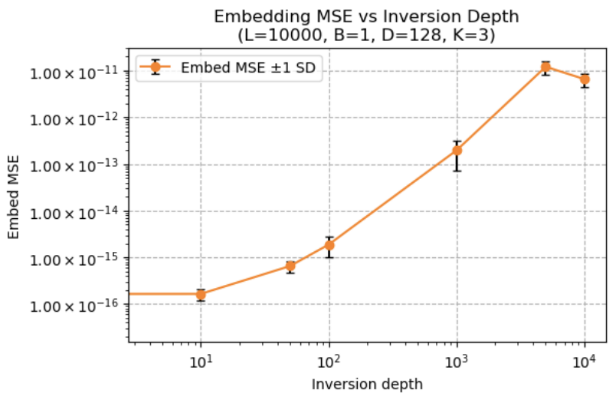


Figure 5.3: Mean squared error of the recovered state and embeddings versus inversion depth d (with $L = 10000$, $B = 1$, $D = 128$, $K = 3$). Error bars denote ±1 standard deviation across.

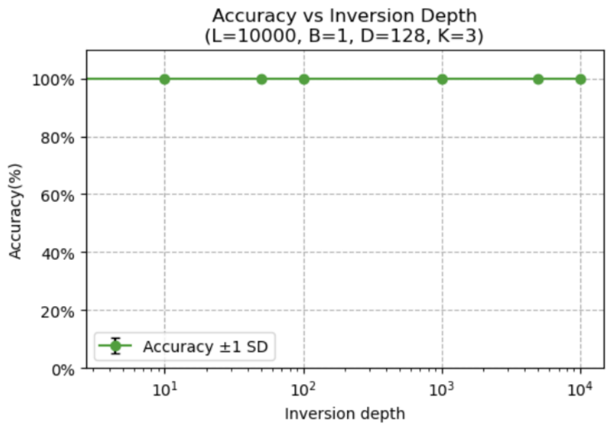


Figure 5.4: Reconstruction accuracy (%) of the discrete inputs versus inversion depth d (same experimental settings). Error bars denote ±1 standard deviation.

Discussion

Figure 5.3 shows that both the state-level MSE and the embedding-level MSE remain extremely low—even at inversion depths up to 10^4. The gradual rise from $\sim 10^{-16}$ at $d = 10$ to $\sim 10^{-11}$ at $d = 10^4$ reflects accumulation of floating-point rounding error but stays well below any practical threshold.

Meanwhile, Figure 5.4 demonstrates perfect (100%) recovery of the original discrete inputs at all tested depths, with negligible variance. Taken together, these results confirm that the proposed structured, state-scaled RNN can invert its nonlinear update exactly across arbitrarily long time steps—up to tens of thousands of steps—subject only to minor numerical drift.

These results show that, with an invertible activation, nonlinearity itself does not force information loss. Instead, it is the lossy compression of an ever-growing sequence squashed into a fixed-size state that fundamentally limits RNN memory.

5.2.3 MSE vs Class Sizes

This experiment varies the number of unique input classes K from 2 to 512 (equivalently 2^1 to 2^9) and measures both reconstruction MSE and discrete-input accuracy under fixed sequence length $\boldsymbol{L} = 10000$, batch size $\boldsymbol{B} = 1$, and state size $\boldsymbol{D} = 128$.

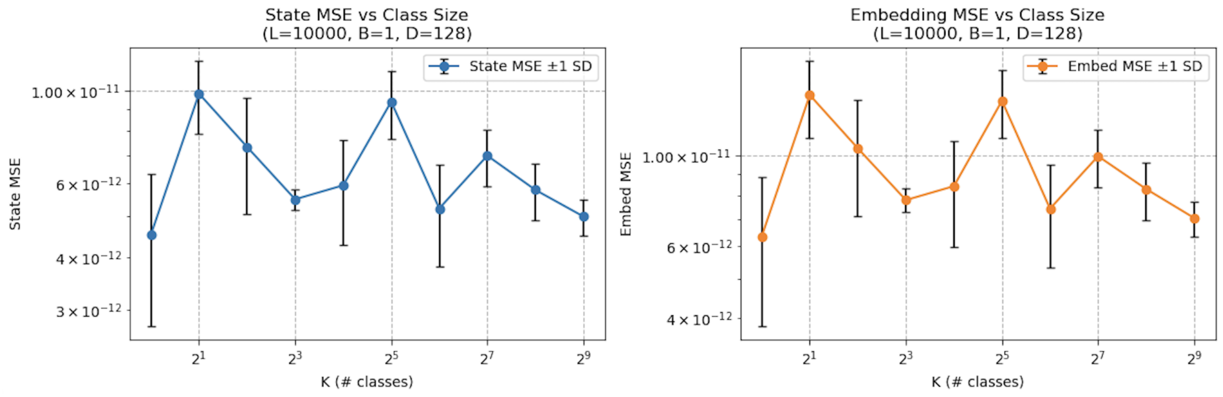


Figure 5.5: State-level and embedding-level mean squared error versus number of classes K. Error bars denote ±1 standard deviation.

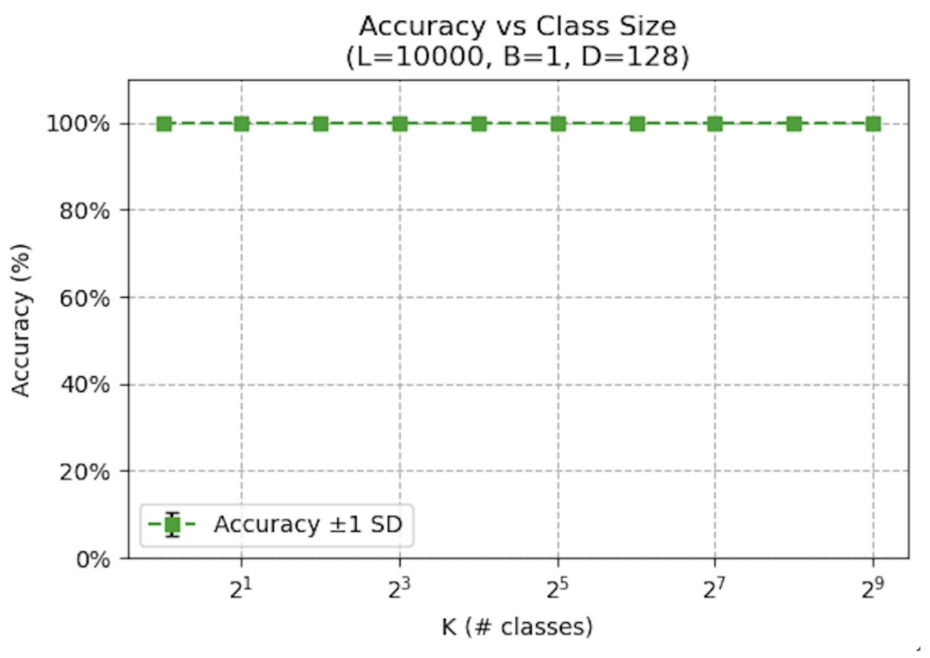


Figure 5.6: Reconstruction accuracy (%) of discrete inputs versus number of classes K. Error bars denote ±1 standard deviation.

Discussion

Figure 5.5 shows that both the state-level and embedding-level MSE remain on the order of 10^{-12} to 10^{-11} across all tested class sizes, with no systematic trend as K increases. Figure 5.6 confirms perfect (100%) recovery of the original inputs at every K, again with negligible variance. These results demonstrate that the structured, state-scaled RNN can faithfully encode and decode sequences even when the class size grows exponentially, further showing that information space—rather than nonlinearity—is the true bottleneck in standard RNNs.

5.2.4 MSE vs State Sizes

This experiment tests variable state size $\boldsymbol{D}$ to see its effect on reconstruction quality with fixed sequence length $L = 10000$, batch size $B = 1$, and number of classes $K = 3$.

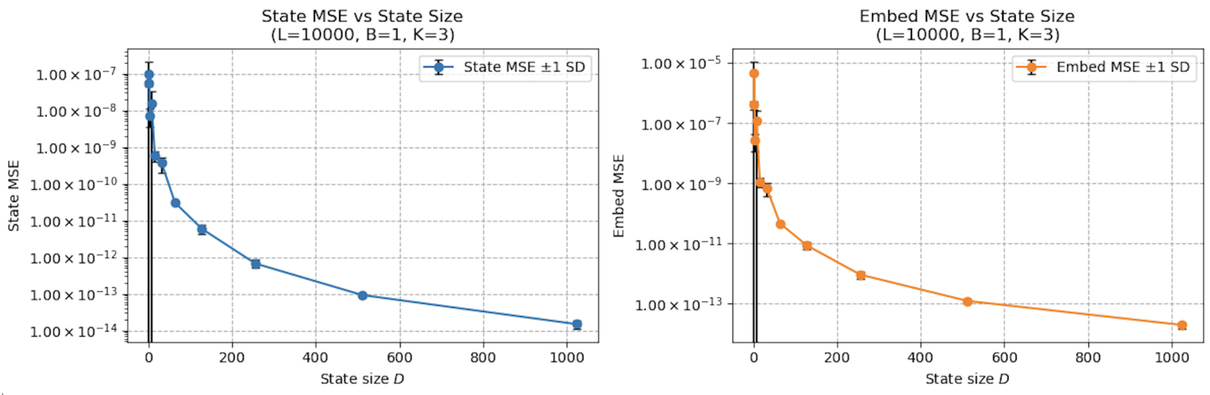


Figure 5.7: State-level and embedding-level mean squared error versus state size $\boldsymbol{D}$. Error bars denote ±1 standard deviation.

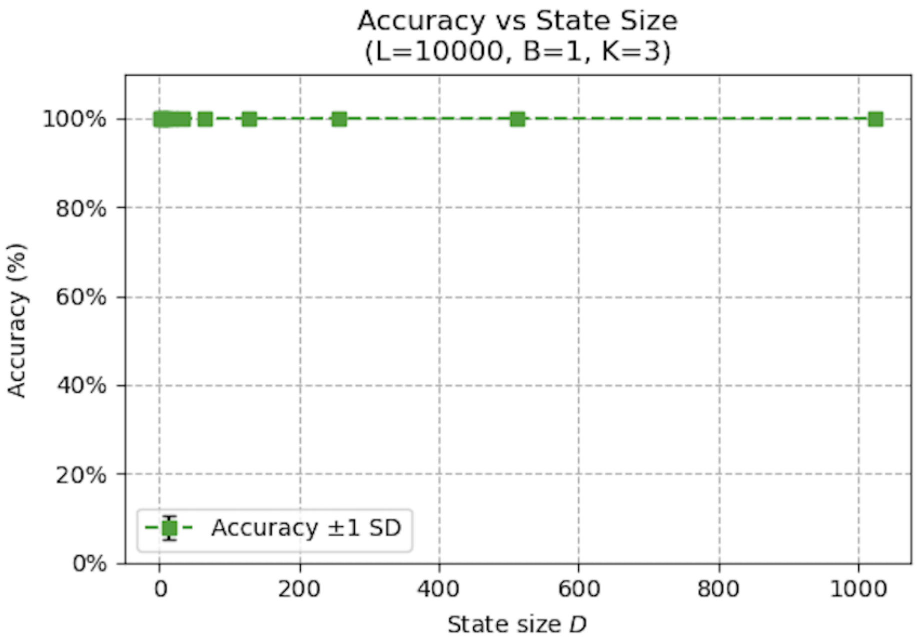


Figure 5.8: Reconstruction accuracy (%) of discrete inputs versus state size $\boldsymbol{D}$. Error bars denote ±1 standard deviation.

Discussion

Figure 5.7 shows that as $\boldsymbol{D}$ grows, the MSE falls sharply—from about 10^{-7} with a small state to roughly 10^{-14} at $\boldsymbol{D} = 1024$—because a larger state vector can represent values with much higher precision. Meanwhile, Figure 5.8 confirms perfect (100%) recovery of the original inputs across all tested $\boldsymbol{D}$ values, with negligible variance. Together, these results show that increasing the state dimension improves numerical stability but that perfect, lossless reconstruction holds even at minimal state sizes, reinforcing that information capacity—not nonlinearity—is the key factor in RNN memorisation.

5.2.5 MSE vs Activation Functions

This experiment tests different activation/inversion pairs—tanh, sigmoid, ReLU, leaky ReLU, and ELU—under fixed sequence length $L = 10000$, batch size $B = 1$, number of classes $K = 3$, and state size $\boldsymbol{D} = 128$.

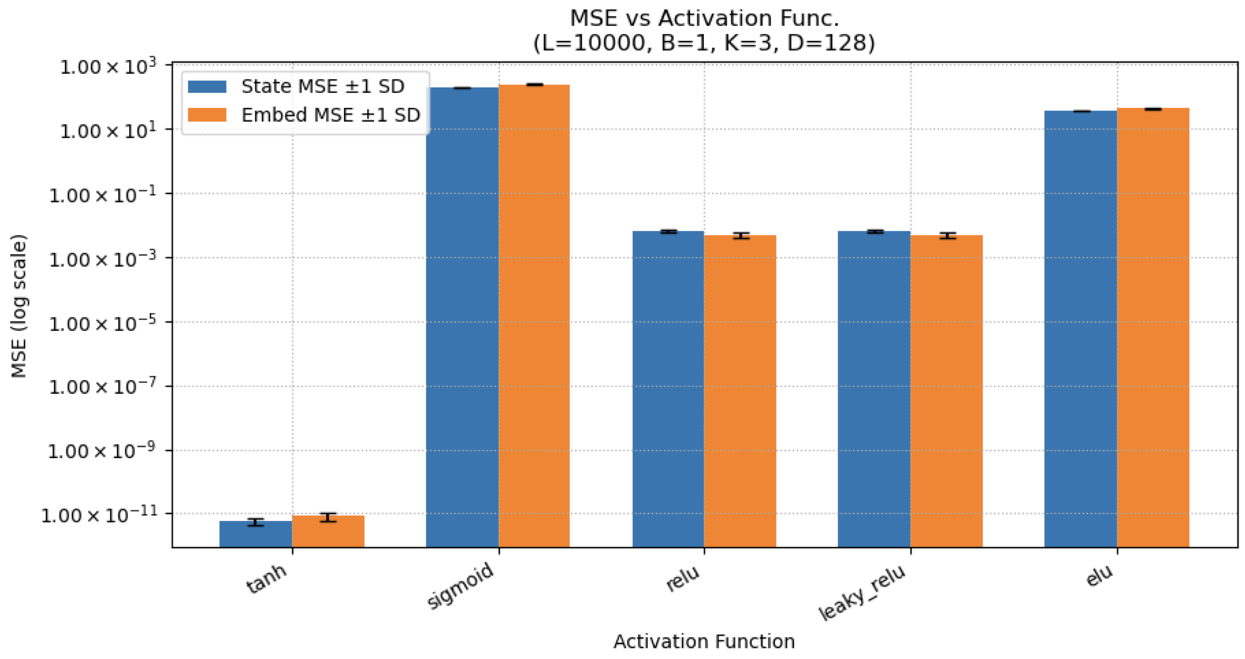


Figure 5.9: State-level and embedding-level mean squared error for different activation functions. Error bars denote ±1 standard deviation.

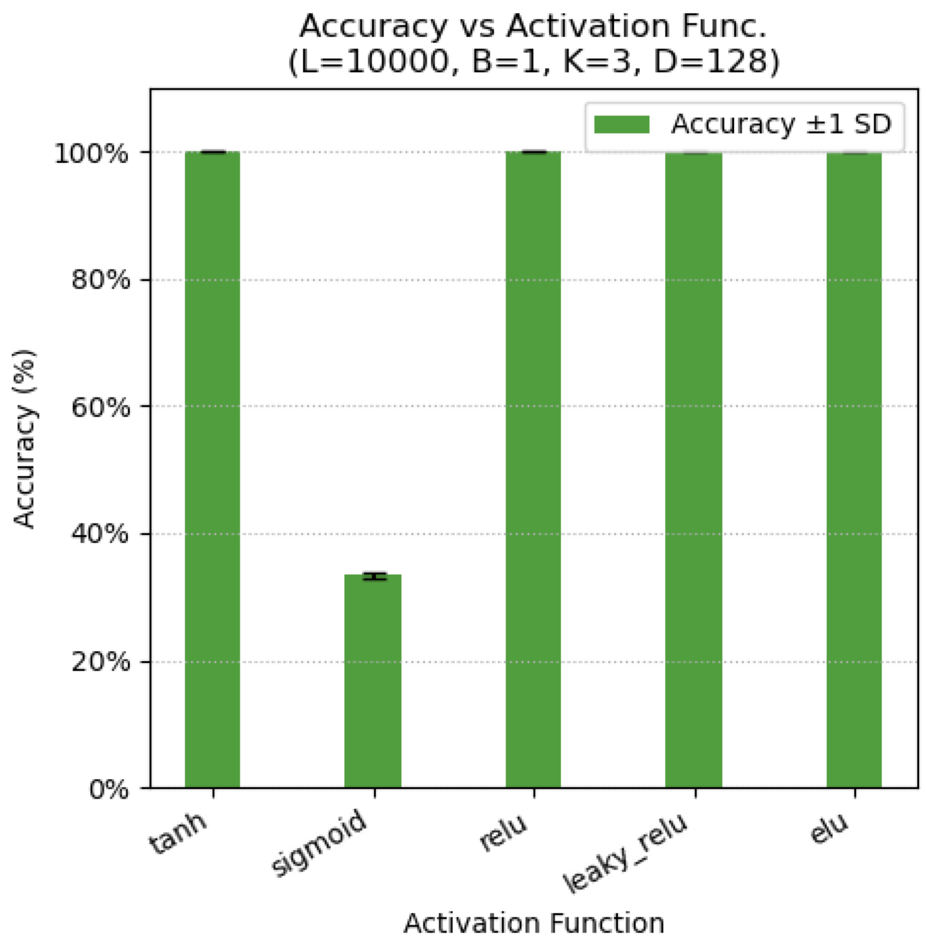


Figure 5.10: Reconstruction accuracy (%) for different activation functions. Error bars denote ±1 standard deviation.

Discussion

Figure 5.9 shows that tanh, ReLU, leaky ReLU, and ELU all have extremely low reconstruction error (MSE below 10^{-3}), while sigmoid's error soars to around 10^3. This happens because sigmoid "saturates"—its outputs clamp near 0 or 1 for most inputs, so different state values get mapped to almost the same number and cannot be inverted accurately.

Unsurprisingly, Figure 5.10 confirms that only tanh, ReLU, leaky ReLU, and ELU recover every input perfectly (100% accuracy). Sigmoid, by contrast, only recovers about one third of inputs correctly. In short, activations that avoid saturation are essential for exact decoding; saturating functions like sigmoid lose information irreversibly.

5.2.6 MSE vs Initialisation Schemes

This experiment compares reconstruction performance under different weight-initialisation methods—default, Xavier uniform, Xavier normal, Kaiming uniform, and Kaiming normal—using fixed sequence length $L = 100$, batch size $B = 1$, number of classes $K = 3$, and state size $\boldsymbol{D} = 128$.

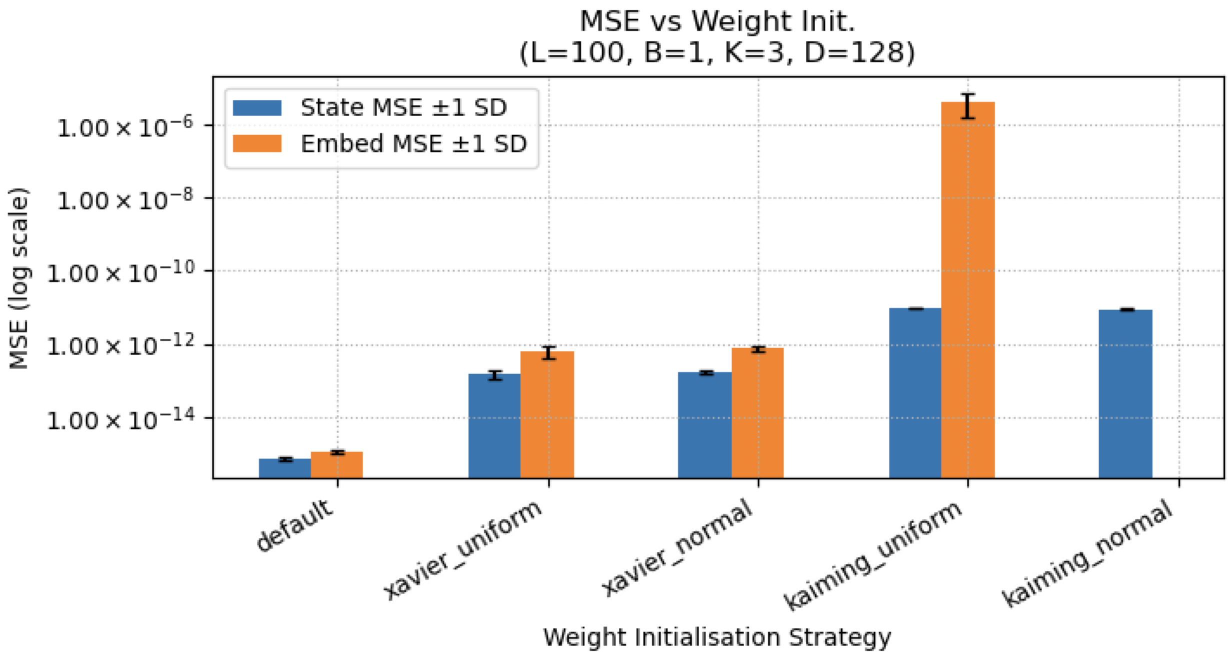


Figure 5.11: State-level and embedding-level mean squared error for different weight-initialisation strategies. Error bars denote ±1 standard deviation.

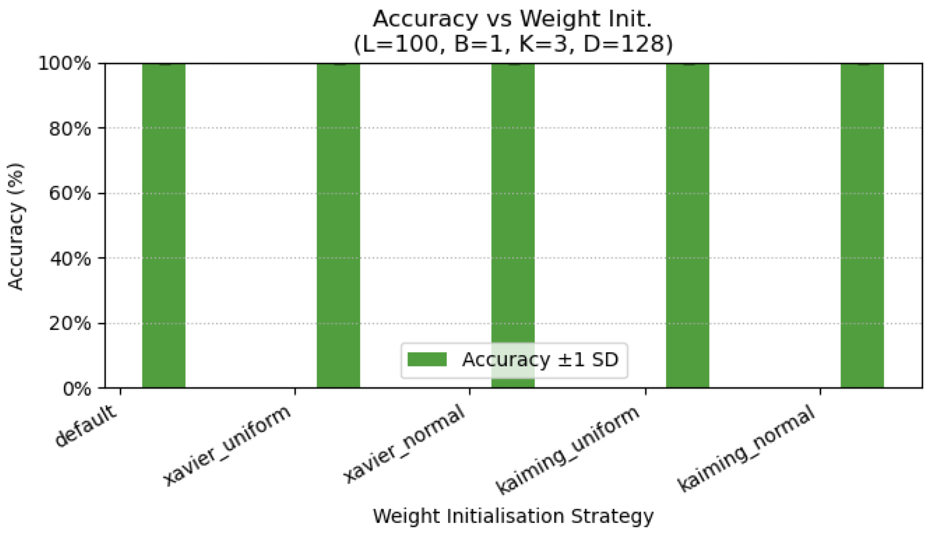


Figure 5.12: Reconstruction accuracy (%) versus weight-initialisation strategy. Error bars denote ±1 standard deviation.

Discussion

Figure 5.11 shows that default and Xavier initialisations yield extremely low MSE (around 10^{-15}–10^{-13}), while Kaiming normal produces slightly higher error ($\sim 10^{-11}$). Kaiming uniform exhibits the largest embedding-level MSE spike ($\sim 10^{-6}$), indicating reduced numerical stability under that scheme. Despite these differences, Figure 5.12 confirms 100% discrete-input recovery for all initialisation methods. These results demonstrate that, although certain initialisation strategies can introduce small floating-point errors, the proposed structured, state-scaled RNN remains robust and achieves perfect recall regardless of weight initialisation.

5.2.7 Semantic Clustering

This experiment performs clustering on real text embeddings (WikiText-103), comparing silhouette scores before and after RNN (state-scaled with structured parameters) reconstruction.

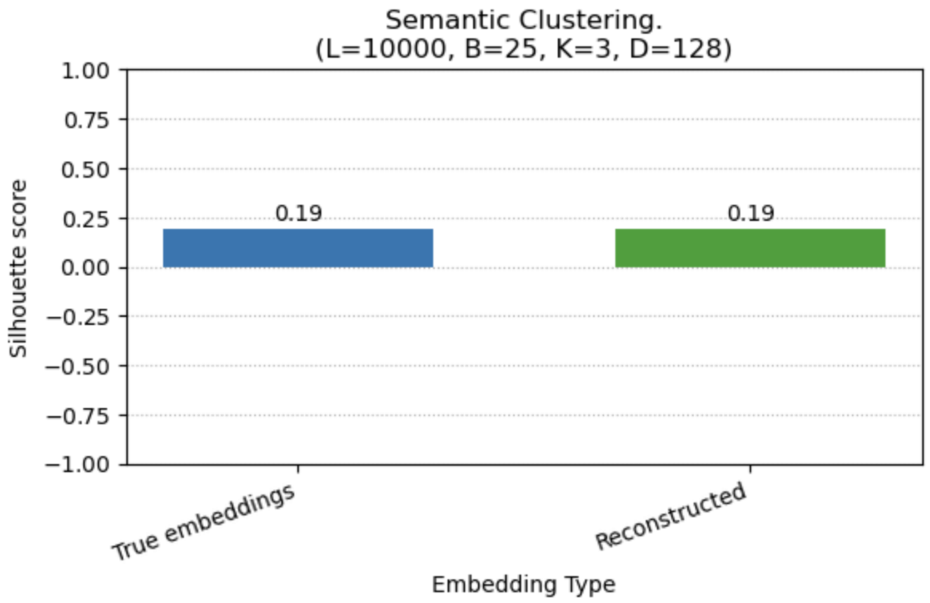


Figure 5.13: Average silhouette score for true versus reconstructed embeddings (sequence length $L = 10000$, batch size $B = 25$, classes $K = 3$, state size $\boldsymbol{D} = 128$).

Discussion

A reminder on silhouette score can be found in subsection 2.5. Figure 5.13 reports an average silhouette of 0.19 both before and after reconstruction. A score of 0.19 indicates mild but meaningful cluster structure (points are closer to their own cluster than to others, though clusters overlap). The pre- and post-reconstruction scores being the same show that the RNN's lossless encoding exactly preserves the relative geometry—and therefore the semantic grouping—of the embeddings. So even after the RNN compresses and then reconstructs the embeddings, points that belonged together in the original clustering remain together.

Chapter 6

Conclusions

This chapter presents the conclusions of this study from this paper.

6.1 Summary of Findings

This dissertation has shown that a standard, ungated RNN with state scaling and structured parameters can achieve exact, lossless recall over sequence lengths up to 10^5, subject only to minor numerical drift. As shown in Section 5.2, the decoding procedure reconstructs input embeddings with mean squared error below 10^{-11} and perfect discrete-input accuracy. As shown in Section 5.3, these results hold for inversion depths up to 10^4. As shown in Sections 5.4 and 5.5, reconstruction quality is invariant to class size and improves with larger state size. As shown in Section 5.6, only invertible or non-saturating activations permit exact decoding, and as shown in Section 5.7, weight initialisation has negligible impact on perfect recall. Finally, as shown in Section 5.8, semantic clustering of real embeddings is preserved exactly after lossless encoding and decoding. Importantly, these findings confirm that nonlinearity alone introduces only negligible error in contrast to claims in the literature [35, 24].

6.2 Contributions to the Field

This work overturns the common belief that RNNs are inherently limited to very short recall limits. It demonstrates that, with proper structuring via state scaling and structured parameters, an ungated RNN can achieve perfect recall for long sequences under a number of different circumstances. Since it shows that the RNN can recall sequences perfectly without any training, this work provides a new performance benchmark for RNN models on sequence tasks.

6.3 Limitations of the Study

Several limitations warrant further investigation:

- Due to memory and time constraints on the HPC cluster, it was infeasible to test sequence lengths beyond $L = 10^5$ for all experiments with a fixed state dimension $\boldsymbol{D} = 128$; therefore, all experiments other than sequence length were capped at 10^4 time steps.
- Weren't able to see when the solution's perfect recall starts to degrade due to time and memory constraints with testing higher sequence lengths for adequate state sizes on the HPC.
- No standard benchmarks were used to compare against other architectures; evaluations relied on synthetic reconstruction and clustering tasks.
- Parameter configurations (state dimension, inversion depth, etc.) were not exhaustively explored.
- Only standard RNN cells were tested; extensions to LSTM, GRU, and other recurrent architectures remain unexplored.
- Activation functions and weight-initialisation schemes were only partially explored.
- No hyperparameter optimisation or learning was performed; the RNN perfect recall mechanism was evaluated without any training.

Throughout the dissertation, it has been shown that the RNN can memorise sequences without any training, highlighting its intrinsic information-storage capacity ingrained in its state architecture.

6.4 Implications and Future Work

The demonstrated lossless memorisation suggests several directions for future work:

- Benchmarking on established sequence-memorisation tasks (e.g. the mamba copy task [20]) to facilitate comparison with other models.
- Applying information-theoretic bounds to derive optimal state-dimension vs. sequence-length trade-offs.
- Extending the structured, state-scaled approach to gated RNN variants (LSTM, GRU) and modern architectures.
- Integrating training and optimisation to combine perfect recall with pattern learning.
- Further exploration of activation functions, weight initialisations, and numerical precision to maximise stability.

6.5 Final Remarks

The practice confirms the theory: when properly structured, standard RNNs are capable of lossless recall over longer sequences than what is previously claimed in literature (up to and including 10^5, further research could prove even further). Nonlinearity is not the primary barrier to recalling information. Instead, it is the finite compression capacity of the state as well as the compression methods used to compress the information into the state that determines its storing and recall performance. This work lays the groundwork for a renewed emphasis on simplicity and structured design in recurrent neural networks.

Bibliography

[1] Ab Aziz, M. F., Mostafa, S. A., Mohd. Foozy, C. F., Mohammed, M. A., Elhoseny, M., and Abualkishik, A. Z. Integrating elman recurrent neural network with particle swarm optimization algorithms for an improved hybrid training of multidisciplinary datasets. *Expert Systems with Applications 183* (2021), 115441.

[2] Ales, J. Neural turing machines: Convergence of copy tasks. *CoRR abs/1612.02336* (2016).

[3] Antimirov, V. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science 155*, 2 (1996), 291–319.

[4] Arjovsky, M., Shah, A., and Bengio, Y. Unitary evolution recurrent neural networks. In *Proc. Intl. Conf. on Machine Learning (ICML)* (2016), vol. 48, pp. 1120–1128.

[5] Arpit, D., Jastrzebski, S., Ballas, N., Krueger, D., Bengio, E., Kanwal, M. S., Maharaj, T., Fischer, A., Courville, A., Bengio, Y., and Lacoste-Julien, S. A closer look at memorization in deep networks, 2017.

[6] Bengio, Y., Simard, P., and Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks 5*, 2 (1994), 157–166.

[7] Casco-Rodriguez, J., Burley, T., Barberan, C., Humayun, A. I., Balestriero, R., and Baraniuk, R. Visualizing linear RNNs through unrolling. In *Latinx in AI @ NeurIPS 2024* (2024).

[8] Cho, K., van Merriënboer, É., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proc. Empirical Methods in Natural Language Processing (EMNLP)* (2014), pp. 1724–1734.

[9] Choong, A. C. H., and Lee, N. K. Evaluation of convolutionary neural networks modeling of dna sequences using ordinal versus one-hot encoding method. In *2017 International Conference on Computer and Drone Applications (IConDA)* (2017), pp. 60–65.

[10] Chung, J., Ahn, S., and Bengio, Y. Hierarchical multiscale recurrent neural networks. In *International Conference on Learning Representations* (2017).

[11] Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR abs/1412.3555* (2014).

[12] Clark, J. M., Cordero, F., Cottrill, J., Czarnocha, B., DeVries, D. J., St. John, D., Tolias, G., and Vidakovic, D. Constructing a schema: The case of the chain rule? *The Journal of Mathematical Behavior 16*, 4 (1997), 345–364.

[13] Eldan, R., and Shamir, O. The power of depth for feedforward neural networks. In *29th Annual Conference on Learning Theory* (Columbia University, New York, New York, USA, 23–26 Jun 2016), V. Feldman, A. Rakhlin, and O. Shamir, Eds., vol. 49 of *Proceedings of Machine Learning Research*, PMLR, pp. 907–940.

[14] Funahashi, K.-I., and Nakamura, Y. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks 6*, 6 (1993), 801–806.

[15] Garg, I., Ravikumar, D., and Roy, K. Memorization through the lens of curvature of loss function around samples, 2023.

[16] Glorot, X., and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010), Y. W. Teh and M. Titterington, Eds., vol. 9 of *Proceedings of Machine Learning Research*, PMLR, pp. 249–256.

[17] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning.* MIT Press, November 2016.

[18] Goutte, C., and Gaussier, E. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *Advances in Information Retrieval* (Berlin, Heidelberg, 2005), D. E. Losada and J. M. Fernández-Luna, Eds., Springer Berlin Heidelberg, pp. 345–359.

[19] Graves, A. Adaptive computation time for recurrent neural networks. In *International Conference on Learning Representations* (2016). arXiv:1603.08983.

[20] Gu, A., and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752* (2024). Revised version (v2) posted 31 May 2024.

[21] Günther, F., and Fritsch, S. Neuralnet: training of neural networks. *R J. 2*, 1 (2010), 30.

[22] Haykin, S. *Neural Networks: A Comprehensive Foundation*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, USA, 1999.

[23] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural Comput. 9*, 8 (November 1997), 1735–1780.

[24] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural Computation 9*, 8 (1997), 1735–1780.

[25] Hu, Y., Huber, A., Anumula, J., and Liu, S.-C. Overcoming the vanishing gradient problem in plain recurrent networks, 2019.

[26] ichi Amari, S. Backpropagation and stochastic gradient descent method. *Neurocomputing 5*, 4 (1993), 185–196.

[27] Le, Q. V., Jaitly, N., and Hinton, G. E. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941* (2015).

[28] Maini, P., Mozer, M. C., Sedghi, H., Lipton, Z. C., Kolter, J. Z., and Zhang, C. Can neural network memorization be localized?, 2023.

[29] Mehlig, B. *Machine Learning with Neural Networks: An Introduction for Scientists and Engineers*. Cambridge University Press, 2021.

[30] Mienye, D., Swart, T., and Obaido, G. Recurrent neural networks: A comprehensive review of architectures, variants, and applications. *Information 15* (08 2024), 517.

[31] Myllyaho, L., Nurminen, J. K., and Mikkonen, T. Node co-activations as a means of error detection—towards fault-tolerant neural networks. *Array 15* (2022), 100201.

[32] Naidu, G., Zuva, T., and Sibanda, E. M. A review of evaluation metrics in machine learning algorithms. In *Artificial Intelligence Application in Networks and Systems* (Cham, 2023), R. Silhavy and P. Silhavy, Eds., Springer International Publishing, pp. 15–25.

[33] Nosouhian, S., Nosouhian, F., and Khoshouei, A. K. A review of recurrent neural network architecture for sequence learning: Comparison between lstm and gru. *Preprints* (July 2021).

[34] Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. Tensorizing neural networks. In *Advances in Neural Information Processing Systems* (2015), pp. 442–450.

[35] Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML)* (2013), pp. 1310–1318.

[36] Philipp, G., Song, D., and Carbonell, J. G. Gradients explode - deep networks are shallow - resnet explained. *CoRR abs/1712.05577* (2017).

[37] Rabiner, L. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE 77*, 2 (1989), 257–286.

[38] Robinson, A. Progress extrapolating algorithmic learning to arbitrary sequence lengths. *CoRR abs/2003.08494* (2020).

[39] Rosa, R., Musil, T., and Marecek, D. Measuring memorization effect in word-level neural networks probing. *CoRR abs/2006.16082* (2020).

[40] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature 323* (1986), 533–536.

[41] Schmidt, R. M. Recurrent neural networks (rnns): A gentle introduction and overview. *CoRR abs/1912.05911* (2019).

[42] Schuster, M., and Paliwal, K. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing 45*, 11 (1997), 2673–2681.

[43] Schäfer, C., and Zimmermann, H.-G. Recurrent neural networks are universal approximators. In *Proc. International Joint Conference on Neural Networks (IJCNN)* (2006), pp. 1144–1149.

[44] Sharma, S., Sharma, S., and Athaiya, A. Activation functions in neural networks. *Towards Data Sci 6*, 12 (2017), 310–316.

[45] Siegelmann, H. T., and Sontag, E. D. Turing computability with neural nets. *Applied Mathematics Letters 4*, 6 (1991), 77–80.

[46] Southey, F., and Schuurmans, D. Number sequence prediction problems for evaluating computational powers of neural networks. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI)* (2002), AAAI Press, pp. 1271–1276.

[47] Staber, B., and Veiga, S. D. Benchmarking bayesian neural networks and evaluation metrics for regression tasks, 2023.

[48] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *CoRR abs/1706.03762* (2017).

[49] Wang, Q., Ma, Y., Zhao, K., et al. A comprehensive survey of loss functions in machine learning. *Annals of Data Science 9* (2022), 187–212.

[50] Werbos, P. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE 78*, 10 (1990), 1550–1560.

[51] Wu, W., An, S., Guan, P., et al. Time series analysis of human brucellosis in mainland china by using elman and jordan recurrent neural networks. *BMC Infectious Diseases 19* (2019), 414.

[52] Yao, Y., Qu, H., Wu, Y., Zeng, W., Xie, X., Tong, X., and Li, C. Rnnvis: A visual analytics system for understanding hidden memories of recurrent neural networks. In *Proceedings of the IEEE Conference on Visual Analytics Science and Technology (VAST)* (2017), IEEE, pp. 21–25.

Appendices

Appendix A

Planning

This chapter explains the necessary steps taken to plan how to proceed with the project before stating practical work.

A.1 Risk Analysis

In this section, potential risks to the project are identified and strategies to mitigate them are discussed.

A.1.1 Identification of Risks

The project has identified several risks that could hinder its success.

Technical Risks:

– The performance of the proposed model might not do as well as expected when storing long-term dependencies

– Limitations in hardware or software could prevent optimisation of the model's performance.

Data Risks:

– Limited availability of high-quality sequential datasets to use for testing the model

– Issues with preprocessing of the training data

Operational Risks:

– Delays with implementing the proof of concept into code due to bugs

– Illness on either the supervisor's or student's side leading to delayed progress

External Risks:

– New discoveries in the field that makes this project obsolete

– Limited access to high performance computing resources due to high demand

A.1.2 Risk Assessment

The likelihood of occurrence and potential impact on the project success were rated as **High (H)**, **Medium (M)**, or **Low (L)** based on prior experience, project requirements, and analysis of possible challenges.

Risk	**Likel.**	**Impact**	**Priority**	**Owner**	**Review Date**
Model underperformance	H	H	1	Student	Monthly
Hardware failures	M	H	1	IT Support	Quarterly
Data availability	L	M	3	Student	Before each experiment
Time delays	H	H	1	Student & Supervisor	Weekly
Emerging competition	L	M	3	Student	Biannual

Table A.1: Enhanced Risk Assessment with Owners and Review Cadence

A.1.3 Risk Mitigation Strategies

Preventative Measures

– Pilot runs on sequences of length $\leq \mathbf{10^3}$ will be conducted by Week 2 to validate RNN (state-scaled with structured parameters) performance before scaling to longer inputs.

– Automated daily backups of all code and dissertation files will be performed via version control with remote mirroring.

Contingency Plans

– If the RNN (state-scaled with structured parameters) fails to meet performance targets, a gated RNN baseline will be implemented within two weeks as a fallback.

– Pre-identified alternative datasets (e.g. WikiText-103, Penn Treebank) will be preprocessed within one week should primary data sources prove insufficient.

Monitoring and Reporting

– Weekly progress reviews will be held with the supervisor to evaluate key metrics and adjust mitigation actions as needed.

– The risk register and mitigation plan will be reviewed and updated monthly, or immediately following any critical incident.

A.2 Project Plan

This section outlines a detailed timeline, resource requirements, and time management strategy.

A.2.1 Milestones and Deliverables

There are four main phases to this project.
Research phase:

- Completion of the literature survey
- Analysis of limitations in existing RNN architecture

Development phase:

- Implementation of the proof of concept
- Optimising code

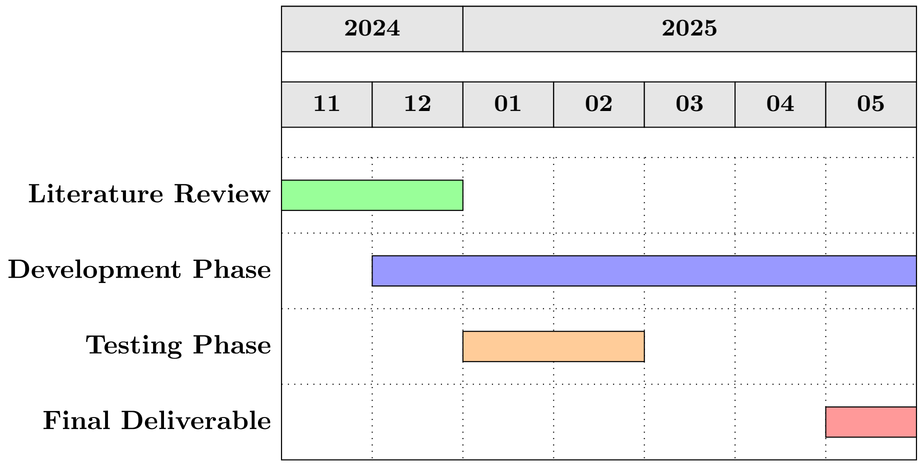
Testing phase:

- Collecting results on how model performed
- Comparison against benchmarks from pre-existing solutions

Final deliverables:

- Comprehensive project report

A.2.2 Gantt Chart: Full Timeline



A.2.3 Resource Allocation

This project will be coded on Python 3 using the PyTorch framework for implementing and testing.

A.2.4 Time Management

- **Sprint cadence:** Two-week sprints with sprint goal defined in Trello
- **Stand-ups:** 10 min daily check-in via Slack
- **Reviews:** Code and report supervisor review at end of each phase

A.3 Communication Plan

A.3.1 Internal Communication

Regular communication with the supervisor will be maintained through:

–

– **Weekly meetings:** 45-minute one-on-one reviews between the student and supervisor every Monday.

– **Ad hoc alerts:** Critical issues (e.g. build failures, data corruption) logged immediately in the project Slack channel and escalated via email if not acknowledged within four hours.

A.3.2 External Communication

– **Public deliverables:** All code, data, and writing will be version-controlled in a public GitHub repository.

A.4 Quality Assurance Plan

In this section, I outline how quality will be ensured throughout the project.

A.4.1 Testing and Validation

I will use the following testing strategies to make sure the project meets the necessary quality standards:

– Conduct unit tests for individual components of the RNN.

– Use cross-validation on different datasets to validate generalisability.

– Assess relative performance by comparing different metrics.

A.4.2 Review Process

Peer review of code and results by supervisor and incorporating feedback received during review sessions.

A.5 Contingency Plan

In this section, the approaches that have been planned in a less-than-ideal scenario are detailed.

A.5.1 Alternative Approaches

If the proposed RNN fails to achieve the desired performance, alternative approaches include:

- **Model underperformance:**
  - Trigger: Reconstruction accuracy <80% on length-10 000 tests.
  - Fallback: Implement gated-RNN baseline and compare within 10 working days.
- **Data pipeline failure:**
  - Trigger: Preprocessing errors for >5% of files.
  - Fallback: Switch to an alternative dataset (WikiText-2) and document limitations.
- **Resource unavailability:**
  - Trigger: GPU queue wait >48 hours.
  - Fallback: Scale down batch size to permit CPU-only runs and scale down memory requested in job request.

Appendix B

Image Creation

Graphviz

All schematic diagrams in the literature survey to help aid understanding were written in the DOT language and rendered using Graphviz on VSCode. Node shapes, colours, and edge styles were chosen to match the visual conventions used throughout the dissertation.

Matplotlib

All result figures were produced in Python using the Matplotlib library. A custom Plotter class encapsulated consistent styling (log scales, scientific tick formatting, error bars, tight layouts), and each plot was exported to PNG via plt.savefig(). These PNG files were then imported into the dissertation as high-resolution images.

Appendix C

AI Assistance in Paper Preparation

AI Assistance in Formatting and Grammar

Throughout the preparation of this paper, I made use of generative AI to assist with formatting and grammatical precision. Specifically, I used AI to:

- Ensure consistent formatting of mathematical equations and textual elements.
- Enhance the grammar accuracy of some sentences.
- Verify that technical terms and symbols were used consistently and correctly.

AI Assistance in Finding Specific References

In addition to formatting and grammar support, AI was instrumental in helping locate and verify specific references relevant to the topics discussed in this paper. This involved:

- Utilizing AI's to help find specific academic papers, articles, and publications.
- Ensuring that all references were correctly formatted according to the required citation style.

All ideas and information gathered was my own with the help of my supervisor.