Dynamical Systems Approaches for Deep Learning

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science,
Mathematical Sciences

By:

Neil Kale

Project Advisor:

Randy Paffenroth

Date: April 2024

Abstract

This study explores iterative neural networks (INNs), which reimagine neural network designs as iterated functions, and the recently introduced Sequential2D framework for neural networks that frames INN functions as left matrix multiplications for enhanced computational efficiency. We investigate the effects of sparse and low-rank matrix approximations on model performance, particularly focusing on sparsity and weight distribution using the MNIST Random Anomaly Task. Our results highlight the delicate balance between parallelization advantages and the need for equitable weight distribution. The comparison of sparse, low-rank, and dense matrices reveals sparse matrices' role in boosting computational speed without drastically affecting model accuracy. Overall, this research advances our understanding of INNs and Sequential2D, underlining the significance of matrix representation methods in fine-tuning neural network architectures for improved performance and efficiency.

Acknowledgements

This project would not have been possible without tremendous help from others. First and foremost, I would like to thank my advisor, Professor Randy Paffenroth, for his technical guidance and witty aphorisms. Without his counsel, I might have ended this project without answering a single question. Thanks to his mentorship, I have answered one question, and more importantly, found about a hundred new questions that spark my curiosity.

I would also like to thank my colleagues in Professor Paffenroth's lab. In particular, thank you to Quincy Hershey and Harsh Pathak, who laid the building blocks for my work. This paper only stands completed thanks to their patience and readiness to help me understand and build upon their results. I stand on the shoulders of giants.

Lastly, I would like to thank my classmate and friend, Harsh Patel. Without his coding expertise in the early stages of this project, this work would have remained purely theoretical. Thanks to him and the others I've mentioned above, I have been able to demonstrate experimental results.

Contents

List of Tables

List of Figures

Executive Summary

This body of research aims to unravel the dynamics of the recently proposed iterative neural networks (INNs), focusing particularly on employing sparse and low-rank matrices within these models. INNs represent traditional neural networks as iterated functions; subsequently, they provide a more generalized approach for modifying neural architectures and selecting hyperparameters. Furthermore, INNs are 2-dimensional and therefore more easily interpretable. Despite their benefits, they are slow and unoptimized for current GPU-based accelerators. Our work is on the implementation of INNs and how it can be done faster.

First, we extend previous research on INNs' representational capacity and show that they capture the Simple Recurrent Unit (SRU), which despite its name is a fairly complex model involving light recurrence and GPU parallelization.

$$\begin{bmatrix}
I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\sigma(\,W_f & v_f & b_f & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\,) \\
0 & 0 & I & -I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
W & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & I & I & 0 & 0 & 0 & 0 & 0 & 0 \\
\sigma(\,W_r & v_r & b_r & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\,) \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & -I & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & I & 0
\end{bmatrix}
\times
\begin{bmatrix}
x_t \\ c_{t-1} \\ 1 \\ f_t \\ 1-f_t \\ Wx_t \\ c_{t1} \\ c_{t2} \\ c_t \\ r_t \\ 1-r_t \\ h_{t1} \\ h_{t2} \\ h_t
\end{bmatrix}
\odot
\begin{bmatrix}
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 \\
I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\times
\begin{bmatrix}
x_t \\ c_{t-1} \\ 1 \\ f_t \\ 1-f_t \\ Wx_t \\ c_{t1} \\ c_{t2} \\ c_t \\ r_t \\ 1-r_t \\ h_{t1} \\ h_{t2} \\ h_t
\end{bmatrix}$$

Figure 1: Our INN representation of the Simple Recurrent Unit.

We identify that parallelizable weights in the Simple Recurrent Unit appear in the same column of the INN. Drawing on the ideas of parallelization and sparsity, we begin the task of optimizing the INN. Through experimentation, we determine that low-rank weight matrices not only hold their ground against dense matrices but, in many instances, surpass them in test set accuracy. This remarkable finding is underscored by a significant reduction in the number of trainable parameters, highlighting the efficiency and potential simplification that low-rank matrices bring to the INN.

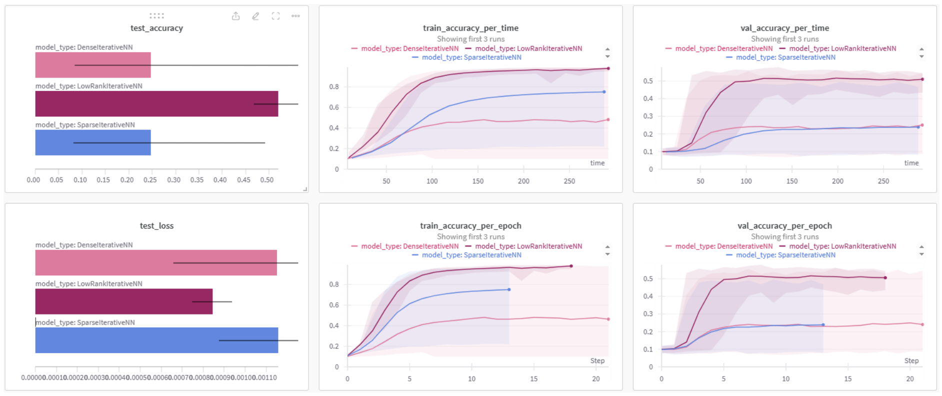

Figure 2: Model performance of dense, sparse, and low-rank weight matrices.

Further investigation into weight distribution indicates a sweet spot in concentrating a higher proportion of weights in the input column of the INN, which not only enhances model performance but also hints at increased parallelizability. We also identify a balance between feed-forward and recurrent connections within INNs, suggesting a trade-off where recurrent connections accelerate training but increase the risk of overfitting.

As we distill these results into actionable insights, it is evident that integrating sparse and low-rank matrices within INNs can catalyze model speed and accuracy. This research not only contributes to the theoretical understanding of INN architectures but also paves the way for practical applications that demand efficiency and accuracy in equal measure.

In conclusion, our exploration into the realms of sparse and low-rank matrices within iterative neural networks proves incredibly promising for the future of machine learning. By harnessing these matrices and fine-tuning the weight distribution and recurrence within neural networks, we can develop more sophisticated, efficient, and effective neural networks. As we continue to push the boundaries of what neural networks can achieve, the insights gleaned from this research will play a pivotal role in shaping the future of machine learning, enabling us to tackle increasingly complex tasks with unprecedented efficiency and accuracy.

1 Introduction

In the realm of machine learning (ML), there is a relentless demand for models that are not only accurate but also efficient and scalable. The complexity of modern ML tasks and burgeoning data demand models that can learn from vast datasets without compromising speed or requiring exorbitant computational resources. Inspired by the human brain's architecture, neural networks have emerged as a powerful tool for pattern recognition, decision-making, and prediction tasks. [1] These models represent complex functions as a series of layers, where each layer's output feeds into the next layer's input. [2] For example, a two-layer neural network could be,

$$F(\mathbf{x}, \theta) = f^{(2)}(f^{(1)}(\mathbf{x}, \theta_1), \theta_2) \tag{1}$$

However, as neural networks grow in size and depth to capture the intricacies of high-dimensional data, the need for innovative strategies to optimize their performance and efficiency becomes essential. [3]

The Iterative Neural Network (INN) is one such strategy, drawing inspiration from an area of mathematics known as dynamical systems theory. [4] This class of models redefines neural networks as iterated functions, turning Equation 1 into

$$F(\mathbf{x}, \theta) = f(f(\mathbf{x}, \theta)) \tag{2}$$

By representing an entire neural network with one function f, INNs provide a more generalized approach for modifying neural architectures and selecting hyperparameters. Furthermore, they only require one function and are therefore more easily interpretable. Despite their benefits, INNs are slow and unoptimized for current GPU-based accelerators. [5]

This study delves into the basic structure of iterative neural networks. By revisiting the fundamental components of these models, particularly the representation and distribution of weight matrices within INNs, we aim to uncover techniques to enhance their computational efficiency and learning performance. Our exploration is grounded in the application of sparse and low-rank matrix representations within INNs, as opposed to traditional dense matrices, to investigate their impact on model speed and accuracy.

Sparse matrices, characterized by primarily consisting of zero-valued elements, offer an enticing prospect for reducing computational overhead. By focusing computational efforts on the non-zero elements, sparse representations promise to accelerate matrix operations, a cornerstone of neural network computations. [6] Similarly, low-rank matrix approximations, which decompose a matrix into simpler, lower-dimensional matrices, offer a path to reduce matrix complexity, potentially enhancing learning efficiency and model interpretability.

Our empirical investigations are set against the MNIST Random Anomaly task, a challenging variant of the widely utilized MNIST handwritten digit recognition task. [7] This task, designed to test the models' ability to discern anomalies in sequences of transformed digit images, provides a fertile ground for assessing the efficacy of sparse and low-rank matrices in handling complex, sequential data.

A pivotal aspect of our study is examining the weight distribution within INNs. By manipulating the allocation of trainable parameters across the input, hidden, and output layers of the network, we seek to identify the influence of weight concentration on learning dynamics and model performance. This exploration is particularly poignant in the context of recurrent connections, which enable the network to leverage temporal dependencies in the data. [8] Balancing the proportion of feed-forward and recurrent connections emerges as a critical factor in tuning the model's capacity for fast learning and robust generalization to unseen data.

Through a series of experiments, we juxtapose the performance of INNs employing sparse, low-rank, and traditional dense weight matrices across varying configurations and effective weight percentages. The findings from these experiments shed light on the trade-offs between computational efficiency, learning speed, and model accuracy, providing valuable insights for the design of optimized neural network architectures.

This research contributes to the ongoing discourse on neural network optimization by offering empirical evidence on the benefits and limitations of sparse and low-rank matrices in INNs. By unraveling the effects of weight distribution and the balance between different types of connections, this study paves the way for the development of faster and more memory-efficient INNs.

2 Methodology

2.1 Notation

Notation	Description
a	Scalar (a real number)
$\mathbf{v}$	Vector (a column vector by default)
A	Matrix
$A \cdot B$	Matrix product of matrices A and B. This is equivalent to AB.
$\mathbf{a} \odot \mathbf{b}$	Hadamard product (element-wise product) of vectors $\mathbf{a}$ and $\mathbf{b}$
$\mathbf{1_n}$	A column vector $(1, 1, \ldots, 1) \in \mathbb{R}^n$

Table 1: Notation table

2.2 Neural Networks

Neural networks are a cornerstone of machine learning, inspired by the structure and function of the human brain. [2] At their core, they aim to model complex relationships between inputs and outputs, learn patterns, and make predictions. A neural network can be mathematically represented as an equation $\mathbf{y} = f(\mathbf{x}, \theta)$, where:

- $\mathbf{x}$ denotes the input data to the network. This could be any form of data such as images, text, or numerical values.
- $\mathbf{y}$ denotes the output from the network. This could be either a vector or a scalar value.
- θ represents the parameters or weights of the network. These are the values that the network adjusts through learning to improve its predictions.

The function f maps inputs to outputs through a series of layers, each consisting of nodes or neurons. The layers transform the input data step by step, using linear combinations followed by nonlinear activation functions, to capture complex relationships and patterns in the data. [2]

To refine the network's parameters for better accuracy, a process known as backpropagation is employed. Backpropagation calculates the gradient of the loss function with respect to each parameter by the chain rule, efficiently propagating the error backward through the network. This allows the network to adjust its parameters in a way that minimizes the error, making the function f a more accurate model of the relationship between $\mathbf{x}$ and $\mathbf{y}$ and enhancing the network's ability to make accurate predictions. [9]

Traditionally, a neural network is not just a single function but rather a composition of multiple nested functions, each associated with a specific layer in the network. [2] These layers are structured in a hierarchy, where each layer's output serves as the input for the next layer. This can be mathematically represented as:

$$f(\mathbf{x}, \theta) = f^{(n)}(f^{(n-1)}(\dots f^{(2)}(f^{(1)}(\mathbf{x}, \theta_1), \theta_2) \dots, \theta_{n-1}), \theta_n) \tag{3}$$

where:

- $f^{(1)}, f^{(2)}, \dots, f^{(n)}$ represent the functions associated with each layer from the input layer to the output layer.
- $\theta_1, \theta_2, \dots, \theta_n$ denote the sets of parameters or weights for each corresponding layer.
- n indicates the total number of layers in the neural network, excluding the input layer.

Each layer's function typically involves a linear transformation, given by $W\mathbf{x} + \mathbf{b}$ where $W \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$ are the weights and biases of the layer where m is the number of output features and n is the number of input features, i.e. the length of $\mathbf{x}$, followed by a nonlinear activation function like ReLU [10], Sigmoid [9], or Tanh [11]. This composition of linear and nonlinear operations allows neural networks to model highly complex and nonlinear relationships in the data.

The architecture of a neural network, including the number of layers and the number of neurons in each layer, is a critical factor that influences its ability to learn and generalize from the input data. [3] The choice of activation functions and the method of initializing the weights are also important considerations that affect the network's performance. [12]

2.3 Iterative Neural Networks

Notice that the neural network equations,

$$f(\mathbf{x}, \theta) = f^{(n)}(f^{(n-1)}(\dots f^{(2)}(f^{(1)}(\mathbf{x}, \theta_1), \theta_2) \dots, \theta_{n-1}), \theta_n)$$

can be rewritten using the function composition notation $\circ$, which represents the composition of functions in a more compact and mathematically elegant way. [4] In this context, function composition $(f \circ g)(x)$ means applying function g to x first and then applying function f to the result of $g(x)$. Applying this concept to the given equation, we can express the nested function applications as a chain of compositions. Therefore, the equation simplifies to:

$$f(\mathbf{x}, \theta) = f^{(n)} \circ f^{(n-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}, \theta_1, \theta_2, \ldots, \theta_{n-1}, \theta_n)$$

This notation streamlines the expression by eliminating the need for repeated parentheses and explicitly showing the sequence of function applications from right to left, starting with $f^{(1)}$ applied to $\mathbf{x}$ and parameters θ_1 through θ_n, and ending with $f^{(n)}$.

Iterative neural networks derive from dynamical systems theory, an area of mathematics interested in modeling complex systems such as feedback loops and fractals. [4] Their basic structure replaces the composition of nested functions from a traditional neural network with functions that iterate upon themselves as depicted in Equation 4 below.

$$f \circ f \circ \cdots \circ f \circ f(\mathbf{x}, \theta) \tag{4}$$

Instead of representing neural networks as n separate functions, where n is the depth of the network, iterative neural networks repeatedly apply the same function to the input, flattening the network into a single layer with parameters θ. Despite this seemingly limiting constraint, iterative neural networks can represent a wide variety of network architectures ranging from multi-layer perceptrons to recurrent neural networks and beyond. [4]

2.4 Generalized MLPs as Iterative Neural Networks

Multi-layer perceptrons are neural networks where the forward propagation of inputs through the network generates predictions. [2] The process involves the following steps:

1. An input vector $\mathbf{x}$ is fed into the input layer.
2. Each hidden layer transforms the inputs using a linear combination of weights W and biases $\mathbf{b}$, followed by a non-linear activation function σ. This can be represented as:

$$\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b}) \tag{5}$$

where $\mathbf{h}$ represents the output of the hidden layer.

3. The transformed data is passed through subsequent layers until it reaches the output layer, which produces the final prediction $\mathbf{y}$.

This process can be summarized for two layers by the following equation.

$$f^1 \circ f^0(\mathbf{x}) = \sigma(W^1\sigma(W^0\mathbf{x})) = \sigma(W^1\mathbf{h}) = \mathbf{y} \tag{6}$$

Here, W^0 and W^1 are implicit inputs to f^0 and f^1 respectively. To represent a multi-layer perceptron as an iterative neural network, there must be a single function f such that

$$f \circ f(\bar{\mathbf{x}}) = \mathbf{y} \tag{7}$$

Here $\bar{\mathbf{x}}$ represents a modified input for the network. This input consists of $\mathbf{x}$, $\mathbf{h}$, and $\mathbf{y}$ vertically concatenated together to form an expanded input vector where $\mathbf{h}$ and $\mathbf{y}$ are initialized as zero vectors. [4] We then define our function as follows.

$$f(\bar{\mathbf{x}}) = f\left(\begin{bmatrix}\mathbf{x}\\ \mathbf{h}\\ \mathbf{y}\end{bmatrix}\right) = \begin{bmatrix}\mathbf{x}\\ \sigma(W^0\mathbf{x})\\ \sigma(W^1\mathbf{h})\end{bmatrix} \tag{8}$$

The first iteration of f correctly computes $\mathbf{h}$, and the second iteration computes $\mathbf{y}$ as,

$$f \circ f(\bar{\mathbf{x}}) = f \circ f\left(\begin{bmatrix}\mathbf{x}\\ \mathbf{0}\\ \mathbf{0}\end{bmatrix}\right) = f\begin{bmatrix}\mathbf{x}\\ \sigma(W^0\mathbf{x})\\ \sigma(W^1\mathbf{0})\end{bmatrix} = \begin{bmatrix}\mathbf{x}\\ \sigma(W^0\mathbf{x})\\ \sigma(W^1(\sigma(W^0\mathbf{x}))\end{bmatrix}$$

Thus, a two-layer MLP can be replaced by an iterative neural network. [4] This idea generalizes for an MLP with any number of layers, only requiring additional iterations to compute the final $\mathbf{y}$.

2.5 Iterative Representations of Recurrent Neural Networks

Beyond multi-layer perceptrons, the iterative neural network architecture can extend to more complex models as well. [8]

Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed to recognize patterns in sequences of data, such as text, genomes, handwriting, or numerical time series data. [9] Unlike traditional neural networks, which process inputs independently, RNNs have loops within them, allowing information to persist. This looping mechanism enables RNNs to take not only the current input but also the previously received inputs into account, effectively giving them a form of memory. [13]

The fundamental operations within an RNN at each timestep can be described by the following equation. [13]

$$\mathbf{h_t} = \sigma(W^i\mathbf{x_t} + W^h\mathbf{h_{t-1}}) \tag{9}$$

Here, the input to the RNN is a sequence of vectors $\mathbf{x_0}, \mathbf{x_1}, \ldots \mathbf{x_n}$, W^i is the input weight matrix and W^h is the hidden weight matrix. The RNN uses its hidden state $\mathbf{h_t}$, which is updated at every timestep, to capture information from previous inputs. This is what gives the RNN its temporal dynamic behavior and the ability to process sequences of data. [13] After the final timestep T, the hidden state $\mathbf{h_T}$ serves as the output $\mathbf{y}$.

To represent an RNN as an iterative neural network, we must identify a single function f such that the following equation holds. [8] Here, f is applied T times, where T is the sequence length of the RNN.

$$f \circ f \circ ... \circ f(\bar{\mathbf{x}}) = \mathbf{y} \tag{10}$$

As with the MLP, $\bar{\mathbf{x}}$ represents a modified input to the network consisting of $\mathbf{x}$ and $\mathbf{h}$ vertically concatenated together to form a single expanded input vector. We can then define our function as follows.

$$f(\bar{\mathbf{x}}) = f\left(\begin{bmatrix}\mathbf{x_t}\\ \mathbf{h_t}\end{bmatrix}\right) = \begin{bmatrix}\mathbf{x_{t+1}}\\ \sigma(W^i\mathbf{x_t} + W^h\mathbf{h_t})\end{bmatrix} = \begin{bmatrix}\mathbf{x_{t+1}}\\ \mathbf{h_{t+1}}\end{bmatrix} \tag{11}$$

We initialize the input with,

$$\bar{\mathbf{x}} = \begin{bmatrix}\mathbf{x_0}\\ \mathbf{0}\end{bmatrix}$$

and after T iterations, the $\mathbf{h_T}$ section of the output will equal $\mathbf{y}$, the final output of the recurrent neural network, as shown below.. [8]

$$f \circ \cdots \circ f \circ f\left(\begin{bmatrix}\mathbf{x_0}\\ \mathbf{0}\end{bmatrix}\right) = f \circ \cdots \circ f\left(\begin{bmatrix}\mathbf{x_1}\\ \mathbf{h_1}\end{bmatrix}\right) = \cdots = f\left(\begin{bmatrix}\mathbf{x_{T-1}}\\ \mathbf{h_{T-1}}\end{bmatrix}\right) = \begin{bmatrix}\mathbf{x_T}\\ \mathbf{h_T}\end{bmatrix} \tag{12}$$

2.6 Iterative Representation of the Simple Recurrent Unit

One significant challenge that arises with Recurrent Neural Networks (RNNs), including advanced variants like LSTMs [14], QRNNS [15], and GRUs [16], is their difficulty in parallelizing operations effectively on GPUs (Graphics Processing Units). [17] GPUs excel at handling multiple operations simultaneously, making them ideal for the parallelizable tasks often found in multi-layer perceptrons and other feedforward architectures. [18] However, RNNs have a sequential nature where each step depends on the computations and outcomes of the previous steps, and this inherently limits the potential for parallel processing. [19] The dependency chain in RNNs introduces a bottleneck because each step must wait for the completion of its predecessor, preventing the full utilization of GPU capabilities for speed improvements. [14] This contrasts with the more straightforward parallelization of other neural network architectures, where inputs can be processed independently. As a result, training and inference with RNNs can be slower and more computationally intensive. [20]

The Simple Recurrent Unit (SRU), introduced by Lei et al. in 2017, addresses the parallelization challenges inherent in traditional Recurrent Neural Networks (RNNs) by redesigning the recurrent computation to reduce the sequential dependencies that typically bottleneck RNNs. [21] SRU incorporates elements of parallelization within the recurrent computations, enabling it to process the large weight matrices of the model concurrently. This design allows SRU to maintain the advantages of RNNs in handling sequential data while significantly increasing the speed of training and inference by leveraging the parallel processing capabilities of GPUs more effectively. [22] The SRU architecture demonstrates that it's possible to retain the sequential data processing capabilities crucial for language modeling and time series analysis, while also achieving computational efficiencies closer to those of feedforward and convolutional networks. It thus offers a promising solution to the parallelization issues that have traditionally limited RNN performance. [21]

Let positive integers $n, T \in \mathbf{N}$, and let an input sequence $\mathbf{x_0}, \mathbf{x_1}, \ldots, \mathbf{x_{T-1}}$ such that $\mathbf{x_i} \in \mathbb{R}^n \quad \forall \quad 0 \leq i < T$. The Simple Recurrent Unit (SRU) is defined by a set of equations as follows: [21]

1. $\mathbf{f_t} = \sigma(W_f \mathbf{x_t} + \mathbf{v_f} \odot \mathbf{c_{t-1}} + \mathbf{b_f})$

This equation computes the forget gate $\mathbf{f_t}$, which determines the extent to which the previous state $\mathbf{c_{t-1}}$ is retained in the current state. W_f and $\mathbf{b_f}$ are the weight matrix and bias for the forget gate, respectively, and σ denotes the sigmoid activation function, ensuring the output ranges between 0 and 1.

2. $\mathbf{c_t} = \mathbf{f_t} \odot \mathbf{c_{t-1}} + (\mathbf{1_n} - \mathbf{f_t}) \odot (W\mathbf{x_t})$

This equation defines the current state $\mathbf{c_t}$, which is updated by a blend of the previous state $\mathbf{c_{t-1}}$ and the new candidate state, $W\mathbf{x_t}$, modulated by the forget gate $\mathbf{f_t}$. Note that $\odot$ denotes element-wise multiplication, and $\mathbf{1_n}$ is a vector of ones.

3. $\mathbf{r_t} = \sigma(W_r\mathbf{x_t} + \mathbf{v_r} \odot \mathbf{c_{t-1}} + \mathbf{b_r})$

Here, $\mathbf{r_t}$ represents the reset state, which adaptively combines the input $\mathbf{x_t}$ and the past state $\mathbf{c_{t-1}}$. W_r and $\mathbf{b_r}$ are the weight and bias for the reset state.

4. $\mathbf{h_t} = \mathbf{r_t} \odot \mathbf{c_t} + (\mathbf{1_n} - \mathbf{r_t}) \odot \mathbf{x_t}$

The final output $\mathbf{h_t}$ is a combination of the input, $\mathbf{x_t}$, and the current state $\mathbf{c_t}$, regulated by the reset state $\mathbf{r_t}$.

These equations collectively enable SRU to efficiently manage the flow of information through the network, maintaining the capacity to capture sequential dependencies while benefiting from parallel computation. One significant improvement is that the weight matrices W_r, W_c, and W are all multiplied by $\mathbf{x_t}$. Consequently, their multiplications can be batched together as follows. [21]

$$U_t = \begin{pmatrix} W \\ W_f \\ W_r \end{pmatrix} [\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_T}] \tag{13}$$

The large weight matrix multiplications can not only be batched together across the SRU equations but also across the input sequence. Notice that because of the element-wise multiplications, the weight vectors $\mathbf{v_f}, \mathbf{b_f}, \mathbf{v_r}, \mathbf{b_r}$ all have length n, which is the length of the inputs $\mathbf{x^i}$. [21]

Converting the Simple Recurrent Unit into an iterative neural network is significantly more complex than representing a multi-layer perceptron or an RNN. The goal is to identify a function f such that satisfies the following equation.

$$\mathbf{c_t}, \mathbf{h_t} = f \circ f \circ ... \circ f(\bar{\mathbf{x_t}}) \tag{14}$$

Here, $\mathbf{h_t}$ is the final output as defined by the SRU equations, $\mathbf{c_t}$ is the control gate, and $\bar{\mathbf{x_t}}$ is a modified input to the equation consisting of $\mathbf{x_t}$ concatenated with $\mathbf{c_{t-1}}$ and several other intermediary values as shown below. We need to compute $\mathbf{c_t}$ since it becomes $\mathbf{c_{t-1}}$ for the next input. Notice that the most straightforward solution, given below, computes $\mathbf{h_t}$ in three iterations.

$$f(\bar{\mathbf{x_t}}) = f\left(\begin{bmatrix} \mathbf{x_t} \\ \mathbf{c_{t-1}} \\ \mathbf{f_t} \\ \mathbf{c_t} \\ \mathbf{r_t} \\ \mathbf{h_t} \end{bmatrix}\right) = \begin{bmatrix} \mathbf{x_t} \\ \mathbf{c_{t-1}} \\ \sigma(W_f\mathbf{x_t} + \mathbf{v_f} \odot \mathbf{c_{t-1}} + \mathbf{b_f}) \\ \mathbf{f_t} \odot \mathbf{c_{t-1}} + (1 - \mathbf{f_t}) \odot (W\mathbf{x_t}) \\ \sigma(W_r\mathbf{x_t} + \mathbf{v_r} \odot \mathbf{c_{t-1}} + \mathbf{b_r}) \\ \mathbf{c_t} \odot (W_h\mathbf{x_t} + \mathbf{b_h}) + (1 - \mathbf{c_t}) \odot \mathbf{r_t} \end{bmatrix} \tag{15}$$

Assume $\mathbf{x_t}$ is initialized as the input, $\mathbf{c_{t-1}}$ is initialized as the previous control state (or zeros, if this is the first input in the sequence), and the remaining values are initialized as zero. The first iteration of f successfully computes $\mathbf{f_t}$ and $\mathbf{r_t}$. The second iteration successfully computes $\mathbf{c_t}$. The third iteration successfully computes $\mathbf{h_t}$.

$$f \circ f \circ f\left(\begin{bmatrix} \mathbf{x_t} \\ \mathbf{c_{t-1}} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}\right) = f \circ f\left(\begin{bmatrix} \mathbf{x_t} \\ \mathbf{c_{t-1}} \\ \mathbf{f_t} \\ * \\ \mathbf{r_t} \\ * \end{bmatrix}\right) = f\left(\begin{bmatrix} \mathbf{x_t} \\ \mathbf{c_{t-1}} \\ \mathbf{f_t} \\ \mathbf{c_t} \\ \mathbf{r_t} \\ * \end{bmatrix}\right) = \begin{bmatrix} \mathbf{x_t} \\ \mathbf{c_{t-1}} \\ \mathbf{f_t} \\ \mathbf{c_t} \\ \mathbf{r_t} \\ \mathbf{h_t} \end{bmatrix} \tag{16}$$

Here, the asterisks ($*$) indicate values that are part of the computation but are not relevant for further discussion or analysis. Although this is technically an iterative function, this solution does little more than wrap the equations.

2.7 Iterative Neural Networks with Sequential2D

As demonstrated, iterative neural networks capture a variety of architectures from MLPs and RNNs to more complex architectures such as the Simple Recurrent Unit. We have shown that theoretically, these models are all representable as iterative neural networks, in their current form, these INN representations are not all particularly insightful.

One major issue is that the iterative functions f used to describe these networks have very different complexities. The iterative representation of the MLP uses only matrix multiplication and non-linearity functions. The iterative RNN function adds in matrix addition. [4] Finally, the iterative Simple Recurrent Unit function we developed in the previous section includes multiple matrix multiplications and element-wise multiplications (Hadamard products) summed together within the same line. The function is practically just a wrapper for the SRU equations.

The Sequential2D framework represents these models in a single unified format. Within Sequential2D, iterative functions are represented as left matrix multiplications. [5] This allows more complex models like the SRU to be broken down into larger but simpler architectures. Instead of involving more complex equations, the model requires a larger matrix multiplication and storage of more intermediary steps. The goal of 'flattening' neural network architectures is better served by the Sequential2D framework. [5]

The Sequential2D framework can be summarized as identifying a matrix F that satisfies the following equation. [5]

$$F \cdot \bar{\mathbf{x}} = f(\mathbf{x}) \tag{17}$$

Here, x is the input for the iterative function as described in the previous sections, and $\bar{x}$ is a modified input for the Sequential2D representation of the iterative function. This input can contain the same values that are in $\mathbf{x}$, or include additional intermediary steps. [5]

2.7.1 Multi-Layer Perceptron with Sequential2D

To represent the two-layer MLP from Section 2.4 with Sequential2D, we can use an $[n_x + n_h + n_y] \times [n_x + n_h + n_y]$ matrix where n_x is the input length, n_h is the hidden size, and n_y is the output size. [4]

$$F\bar{\mathbf{x}} = F \cdot \begin{bmatrix} \mathbf{x} \\ \mathbf{h} \\ \mathbf{y} \end{bmatrix} = \begin{matrix} \\ \sigma(\\ \sigma(\end{matrix} \begin{bmatrix} I & 0 & 0 \\ W^0 & 0 & 0 \\ 0 & W^1 & 0 \end{bmatrix} \begin{matrix} \\ ) \\ ) \end{matrix} \cdot \begin{bmatrix} \mathbf{x} \\ \mathbf{h} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ \sigma(W^0\mathbf{x}) \\ \sigma(W^1\mathbf{h}) \end{bmatrix} \tag{18}$$

Here, $I \in \mathbb{R}^{n_x \times n_x}$, $W^0 \in \mathbb{R}^{n_h \times n_x}$, and $W^1 \in \mathbb{R}^{n_y \times n_h}$. The 0s similarly represent zero matrices. Notice that this is not purely a matrix multiplication. The σ on the left-hand side of the second and third rows indicates that after the matrix multiplication, the sigmoid function is applied to the value in that row. That allowance for nonlinearity is what distinguishes a Sequential2D multiplication from an ordinary matrix multiplication. [5] With that allowance, the Sequential2D multiplication F performs identically to the iterative function described in Section 2.4. [4] We can initialize the input as

$$\bar{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}$$

One left multiplication by F correctly computes $\mathbf{h}$, and the second left multiplication correctly computes the final output $\mathbf{y}$. [4] In other words,

$$F(F\bar{\mathbf{x}}) = F(F \cdot \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}) = F \cdot \begin{bmatrix} \mathbf{x} \\ \mathbf{h} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ \mathbf{h} \\ \mathbf{y} \end{bmatrix}$$

as desired.

2.7.2 Recurrent Neural Network with Sequential2D

To represent the recurrent neural network from Section 2.5 with Sequential2D, we can use a $[n_x + n_h] \times [n_x + n_h]$ matrix where n_x is the input length and n_h is the hidden size. [8] Let the input sequence length T. Furthermore, we define a convenience function τ which takes an input and returns the next input in the sequence, so

$$\tau(\mathbf{x_t}) = \begin{cases} \mathbf{x_{t+1}} & 0 \leq t+1 < T \\ \mathbf{0} & \text{otherwise} \end{cases}$$

Note that this function can be interpreted as a nonlinear activation for the Sequential 2D, although its purpose is to iterate through the inputs. With that, the Sequential2D representation of the recurrent neural network is as follows. [4]

$$F\bar{\mathbf{x}}_\mathbf{t} = F \cdot \begin{bmatrix} \mathbf{x_t} \\ \mathbf{h_t} \end{bmatrix} = \begin{matrix} \tau(\\ \sigma(\end{matrix} \begin{bmatrix} I & 0 \\ W^i & W^h \end{bmatrix} \begin{matrix}) \\ ) \end{matrix} \cdot \begin{bmatrix} \mathbf{x_t} \\ \mathbf{h_t} \end{bmatrix} = \begin{bmatrix} \tau(\mathbf{x_t}) \\ \sigma(W^i\mathbf{x_t} + W^h\mathbf{h_t}) \end{bmatrix} = \begin{bmatrix} \mathbf{x_{t+1}} \\ \mathbf{h_{t+1}} \end{bmatrix} \tag{19}$$

A single multiplication by F (with non-linearity) computes the hidden state for an input. Notice that after each iteration, $\mathbf{x_t}$ is replaced by the following input $\mathbf{x_{t+1}}$. We can initialize the input as

$$\bar{\mathbf{x}}_{\mathbf{0}} = \begin{bmatrix} \mathbf{x_0} \\ \mathbf{0} \end{bmatrix}$$

Recall that T is the length of the input sequence. After T left multiplications by the Sequential2D matrix F, the output is

$$F \cdot F \cdot \dots \cdot F \cdot \bar{\mathbf{x}}_{\mathbf{0}} = \bar{\mathbf{x}}_{\mathbf{T}} = \begin{bmatrix} \mathbf{0} \\ \mathbf{h_T} \end{bmatrix}$$

Notice there is no $\mathbf{x_T}$ since the sequence is 0-indexed. Therefore, $\tau(\mathbf{x_{T-1}}) = \mathbf{0}$. The hidden state $\mathbf{h^T}$ is also the final output $\mathbf{y}$ of the RNN. [4]

2.7.3 Simple Recurrent Unit with Sequential2D

The Simple Recurrent Unit is significantly more complicated to represent as a Sequential2D. However, redefining it in this framework has several benefits. First, it provides a way to incrementally transition from simpler networks such as the multi-layer perceptron to more complex networks by changing one matrix block at a time. Secondly, it means that any performance improvements which are made to the Sequential2D architecture then extend to improving the performance of the Simple Recurrent Unit.

To that end, we derive a $14n \times 14n$ matrix multiplication representing the Simple Recurrent Unit, where n is the length of the input vector. This representation cleverly decomposes the SRU equations into individual matrix additions and multiplications. To account for the element-wise products within the SRU architecture, the representation requires the element-wise product of two Sequential2D matrix multiplications. This adds some complexity, however maintains the overall goal of Sequential2D, which is to flatten the neural network.

For this formulation, we need diagonal matrices V_f, B_f, V_r, B_r containing the entries of $\mathbf{v_f}, \mathbf{b_f}, \mathbf{v_r}, \mathbf{b_r}$. Diagonal matrices only contain non-zero entries on their main diagonal. For example, let $\mathbf{v_f} = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}^T$, then the diagonal matrix corresponding to $\mathbf{v_f}$ is

$$V_f = \begin{bmatrix} v_1 & & \\ & \ddots & \\ & & v_n \end{bmatrix}$$

As shown below, left multiplication by the diagonal matrix V_f has the same action as elementwise multiplication by the vector $\mathbf{v_f}$.

$$\mathbf{v_f} \odot \mathbf{x} = \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{bmatrix} \odot \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} v_1x_1 \\ v_2x_2 \\ \dots \\ v_nx_n \end{bmatrix} = \begin{bmatrix} v_1 & 0 & \dots & 0 \\ 0 & v_2 & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \dots & 0 & v_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = V_f \cdot \mathbf{x} \tag{20}$$

With this in mind, the equation below computes the Simple Recurrent Unit with Sequential2D matrix multiplications.

$$F(\bar{\mathbf{x}}) = G\bar{\mathbf{x}} \odot H\bar{\mathbf{x}} \tag{21}$$

where,

$$G\bar{\mathbf{x}} = \begin{array}{r} \\ \\ \\ \sigma(\\ \\ \\ \\ \\ \\ \sigma(\\ \\ \\ \\ \\ \end{array} \begin{bmatrix} I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ W_f & V_f & B_f & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & -I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ W & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & I & I & 0 & 0 & 0 & 0 & 0 & 0 \\ W_r & V_r & B_r & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & -I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & I & 0 \end{bmatrix} \begin{array}{l} \\ \\ \\ ) \\ \\ \\ \\ \\ \\ ) \\ \\ \\ \\ \\ \end{array} \cdot \begin{bmatrix} \mathbf{x_t} \\ \mathbf{c_{t-1}} \\ \mathbf{1_n} \\ \mathbf{f_t} \\ \mathbf{1_n} - \mathbf{f_t} \\ W\mathbf{x_t} \\ \mathbf{c_{t1}} \\ \mathbf{c_{t2}} \\ \mathbf{c_t} \\ \mathbf{r_t} \\ \mathbf{1_n} - \mathbf{r_t} \\ \mathbf{h_{t1}} \\ \mathbf{h_{t2}} \\ \mathbf{h_t} \end{bmatrix} \tag{22}$$

and,

$$H\bar{\mathbf{x}} = \begin{bmatrix} 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 \\ I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x_t} \\ \mathbf{c_{t-1}} \\ \mathbf{1_n} \\ \mathbf{f_t} \\ \mathbf{1_n} - \mathbf{f_t} \\ W\mathbf{x_t} \\ \mathbf{c_{t1}} \\ \mathbf{c_{t2}} \\ \mathbf{c_t} \\ \mathbf{r_t} \\ \mathbf{1_n} - \mathbf{r_t} \\ \mathbf{h_{t1}} \\ \mathbf{h_{t2}} \\ \mathbf{h_t} \end{bmatrix} \tag{23}$$

Recall that the SRU weight vectors $\mathbf{v_f}, \mathbf{b_f}, \mathbf{v_r}, \mathbf{b_r}$ all have length n. Since each section of the input $\bar{\mathbf{x}}$ has the same length, the identity matrices are square, and element-wise multiplication throws no errors. We initialize the input as follows.

$$\bar{\mathbf{x}} = \begin{bmatrix} \mathbf{x_t} & \mathbf{0} & \mathbf{1_n} & \frac{1}{2}\mathbf{c_{t-1}} & \frac{1}{2}\mathbf{c_{t-1}} & \mathbf{0} & \frac{1}{2}\mathbf{c_{t-1}} & \frac{1}{2}\mathbf{c_{t-1}} & \mathbf{c_{t-1}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}^T \tag{24}$$

For the first input in the sequence $\mathbf{x_0}$, we let $\mathbf{c_{-1}}$ be a zero vector. After two left multiplications by Sequential2D matrix G, followed by four left multiplications by F, the output is

$$F(F(F(F(G(G(\bar{\mathbf{x}})))))) = \begin{bmatrix} \mathbf{x_t} & \mathbf{c_t} & \mathbf{1_n} & * & * & \mathbf{Wx_t} & * & \mathbf{c_{t2}} & \mathbf{c_t} & * & * & * & \mathbf{h_{t2}} & \mathbf{h_t} \end{bmatrix}^T$$

Recall that the asterisks (*****) indicate values that are part of the computation but are not relevant for further discussion or analysis. After these six multiplications, the Sequential2D representation correctly computes $\mathbf{c_t}$, the current state, and $\mathbf{h_t}$, the hidden state. Henceforth, we refer to two iterations of left multiplication by G followed by four iterations of left multiplication by F as one complete iteration of the

Sequential2D representation of the Simple Recurrent Unit.

$\bar{\mathbf{x}}$	Input	Iter 1	Iter 2	Iter 3	Iter 4	Iter 5	Iter 6		Output
$\mathbf{x}$	✓	✓	✓	✓	✓	✓	✓	$\rightarrow$	✓
$\mathbf{c_{t-1}}$	✓	✓	✓	✓	$\mathbf{x}$	$\mathbf{c_t}$	$\mathbf{c_t}$	$\rightarrow$	$\mathbf{c_t}$
$\mathbf{1_n}$	✓	✓	✓	✓	✓	✓	✓	$\rightarrow$	✓
$\mathbf{f_t}$	$\frac{1}{2}\mathbf{c_{t-1}}$	✓	✓	✓	✓	$*$	$*$	$\rightarrow$	$\frac{1}{2}\mathbf{c_t}$
$\mathbf{1-f_t}$	$\frac{1}{2}\mathbf{c_{t-1}}$	$*$	✓	✓	✓	✓	$*$	$\rightarrow$	$\frac{1}{2}\mathbf{c_t}$
$\mathbf{Wx_t}$	$\mathbf{0}$	✓	✓	✓	✓	✓	✓	$\rightarrow$	$\mathbf{0}$
$\mathbf{c_{t1}}$	$\frac{1}{2}\mathbf{c_{t-1}}$	$\frac{1}{2}\mathbf{c_{t-1}}$	$*$	✓	✓	✓	$*$	$\rightarrow$	$\frac{1}{2}\mathbf{c_t}$
$\mathbf{c_{t2}}$	$\frac{1}{2}\mathbf{c_{t-1}}$	$\frac{1}{2}\mathbf{c_{t-1}}$	$*$	✓	✓	✓	✓	$\rightarrow$	$\frac{1}{2}\mathbf{c_t}$
$\mathbf{c_t}$	$\mathbf{c_{t-1}}$	$\mathbf{c_{t-1}}$	$\mathbf{c_{t-1}}$	$*$	✓	✓	✓	$\rightarrow$	$\mathbf{c_t}$
$\mathbf{r_t}$	$\mathbf{0}$	✓	✓	✓	✓	$*$	$*$	$\rightarrow$	$\mathbf{0}$
$\mathbf{1-r_t}$	$\mathbf{0}$	$\mathbf{1_n}$	✓	✓	✓	✓	$*$	$\rightarrow$	$\mathbf{0}$
$\mathbf{h_{t1}}$	$\mathbf{0}$	$\mathbf{0}$	$*$	$*$	$*$	✓	$*$	$\rightarrow$	$\mathbf{0}$
$\mathbf{h_{t2}}$	$\mathbf{0}$	$\mathbf{0}$	$*$	✓	✓	✓	✓	$\rightarrow$	$\mathbf{0}$
$\mathbf{h_t}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$*$	$*$	✓	✓	$\rightarrow$	$\mathbf{h_t}$

Table 2: State of the modified input vector $\bar{\mathbf{x}}$ after each iteration of the Sequential2D formulation of the Simple Recurrent Unit.

Here, ✓ represents correct values, i.e. the checkmarks in the first row indicate that the $\mathbf{x}$ section of $\bar{\mathbf{x}}$ always equals the input values $\mathbf{x}$. Note, after the sixth step at the end of the complete iteration, we apply the convenience function τ to the first row of F.

$$\tau(\mathbf{x_t}) = \begin{cases} \mathbf{x_{t+1}} & 0 \leq t+1 < T \\ \mathbf{0} & \text{otherwise} \end{cases}$$

This function swaps in the next input in the sequence. We apply similar convenience functions to every row to correctly initialize $\bar{\mathbf{x}}$ for the next input sequence. After completing the six steps as described above for each input $\mathbf{x_t} \mid t \in [0, T]$ where T is the sequence length, then $\mathbf{h_T}$ is the final output. It takes $6T$ total steps to compute $\mathbf{h_T}$.

Note that each multiplication by F is the element-wise product of two matrix multiplications.

$$F(\bar{\mathbf{x}}) = G\bar{\mathbf{x}} \odot H\bar{\mathbf{x}}$$

The Sequential2D representation flattens the SRU into a much simpler matrix multiplication. However, this comes at a cost. Two multiplications by G and four multiplications by F totals to 10 matrix multiplications. Thus, it takes $10T$ total matrix multiplications to compute the final output. Since $G, H \in \mathbb{R}^{14n \times 14n}$, each matrix multiplication requires $14n * 14n = 196n^2$ float multiplications, where n is the length of the input vectors.

A majority of these operations are multiply-by-zeros. The identity matrices I and the diagonalized weight matrices V_f, B_f, V_r, B_r each contain n non-zero entries. The weight matrices W_f, W, W_r each contain n^2 non-zero entries. The non-zero matrix entries of the Sequential2D formulation of the SRU are highlighted for clarity in Figure 3 below.

$$
\begin{array}{r|cccccccccccccc|l}
 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & \\
 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
\sigma(& W_f & v_f & b_f & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 &) \\
 & 0 & 0 & I & -I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & W & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & I & I & 0 & 0 & 0 & 0 & 0 & 0 & \\
\sigma(& W_r & v_r & b_r & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 &) \\
 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & -I & 0 & 0 & 0 & 0 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & I & 0 &
\end{array}
\times
\begin{bmatrix} x_t \\ c_{t-1} \\ 1 \\ f_t \\ 1-f_t \\ Wx_t \\ c_{t1} \\ c_{t2} \\ c_t \\ r_t \\ 1-r_t \\ h_{t1} \\ h_{t2} \\ h_t \end{bmatrix}
\odot
\begin{bmatrix}
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 \\
I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\times
\begin{bmatrix} x_t \\ c_{t-1} \\ 1 \\ f_t \\ 1-f_t \\ Wx_t \\ c_{t1} \\ c_{t2} \\ c_t \\ r_t \\ 1-r_t \\ h_{t1} \\ h_{t2} \\ h_t \end{bmatrix}
$$

Figure 3: The 14x14 iterative representation of the Simple Recurrent Unit.

Referring to Figure 3, Sequential2D matrix G contains 15 identity matrices, 4 diagonalized weight matrices, and 3 normal weight matrices, all in $\mathbb{R}^{n \times n}$. This totals to $19n + 3n^2$ non-zero entries. Similarly, Sequential2D matrix H contains 14 identity matrices in $\mathbb{R}^{n \times n}$, totalling to $14n$ non-zero entries. This means that for each complete iteration of the SRU, out of $196n^2 * 10$ matrix multiplications, only $6(19n + 3n^2) + 4(14n) = 18n^2 + 170n$ are not multiply-by-zero operations. As the problem size grows large,

$$\lim_{n \to \infty} \frac{18n^2 + 170n}{196n^2} = \frac{18}{196} \approx 9.2\% \tag{25}$$

Therefore for large input vectors, less than one out of every ten SRU operations is non-zero. The table below provides the percentage of non-zero multiplications for various input lengths n.

n	Total Ops ($196n^2$)	Non-zero Ops ($18n^2 + 170n$)	% Non-zero Ops
10	19600	3500	17.86
50	490000	53500	10.92
100	1960000	197000	10.05
576	65028096	6069888	9.33
1000	196000000	18170000	9.27

Table 3: Number of non-zero multiplications in the Sequential2D SRU at various input sizes.

While this is certainly slower than the original SRU equations, the Sequential2D formulation does retain one of the crucial SRU speed-ups. Note that in the Sequential2D representation of the Simple Recurrent Unit, the weight matrices W_f, W_r, W all appear in the same column. This means that the weight matrix multiplications can all be computed in parallel, as described in Equation 11. They are all multiplied by the input x, so the multiplications can be done simultaneously. However, even with that optimization, the Sequential2D representation of the Simple Recurrent Unit will train much slower than the original equations because of unnecessary multiply-by-zero operations.

2.8 Improvements to Iterative Neural Networks

As demonstrated, iterative neural networks have a very high representational capacity and can emulate multi-layer perceptrons, recurrent neural networks, and even Simple Recurrent Units. [4] We have further shown that, for the given examples, iterative neural networks can be represented by Sequential2D matrices - which are essentially matrices with added non-linearity. Although a general proof is not included here, it stands to reason that many other complex networks like the Simple Recurrent Unit, and possibly even all neural networks, could be represented by Sequential2D. [5] By improving the performance of Sequential2D networks, we can foreseeably improve the performance of the various architectures that Sequential2D can contain.

2.8.1 Sparsity

One potential improvement to Sequential2D is using sparse matrices to eliminate the many multiplication-by-zero operations that slow down its performance.

Sparse matrices are characterized by having most of their elements as zeros. For example, a diagonal matrix $D \in \mathbb{R}^{n \times n}$ has n non-zero elements out of n^2 total elements. This diagonal matrix can be written as:

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

where $a_{11}, a_{22}, \ldots, a_{nn}$ are the non-zero elements, and the rest are zeros. As n grows larger, diagonal matrices get increasingly sparse.

The Compressed Sparse Column (CSC) format is an efficient way to store sparse matrices. [6] It uses three arrays to represent the matrix:

1. **Values Array** (`val`): Stores all the non-zero elements of the matrix, column by column.
2. **Row Indices** (`row_ind`): Contains the row indices for each non-zero element in the `val` array.
3. **Column Pointers** (`col_ptr`): Stores index pointers to the first element of each column in the `val` array. The size of this array is one more than the number of columns, with the last element pointing just beyond the last element of `val`.

For example, consider the following sparse matrix:

$$A = \begin{bmatrix} 0 & 4 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 5 & 6 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The CSC representation of matrix A would be:

`val` = [3, 4, 5, 6]

`row_ind` = [1, 0, 2, 2]

`col_ptr` = [0, 1, 3, 4, 4]

Here, `col_ptr[0]` points to the start of the first column in `val`, `col_ptr[1]` points to the start of the second column, and so on. The last element of `col_ptr` helps in identifying the range of the last column. Here, using CSC representation reduces the storage space for matrix A from $4 \times 4 = 16$ values to $4 + 4 + 5 = 13$ values. Moreover, given an arbitrary vector $x \in \mathbb{R}^{4\times 1}$, computing Ax requires only 4 multiplications now instead of 16. This suggests that given a neural network whose weight matrices W have many zero entries, using sparse representations could significantly improve the model speed.

The idea of a sparse matrix representation can be extended to the Simple Recurrent Unit matrices as well. Notice that, at a block level, we can treat the Sequential2D formulation of the Simple Recurrent Unit from Equation 21 as two 14×14 matrices of functions. In this case, the CSC representation for the Simple Recurrent Unit matrix G from Equation 22 is:

`val` = $[I, W_f, W, W_r, V_f, V_r, I, B_f, B_r, I, -I, I, I, I, I, I, -I, I, I, I, I]$

`row_ind` = [0, 3, 5, 9, 3, 9, 2, 3, 9, 10, 4, 6, 7, 8, 8, 1, 10, 11, 12, 13, 13]

`col_ptr` = [0, 4, 6, 10, 12, 13, 13, 14, 15, 16, 18, 19, 20, 21]

Notice that the `val` and `row_ind` arrays always have a length equal to the number of nonzero entries in the matrix, and `col_ptr` has length equal to the number of columns in the matrix. Treating the matrices each as one value, CSC representation reduces the number of stored values from $14 \times 14 = 196$ values to $21+21+14 = 56$ values. The representation for matrix H is similarly concise. If we instead break the weight matrices, identity matrices, and zero matrices into their component values, the resulting CSC representation saves even more space. In this case, both matrices G, H from Equation 21 are in $\mathbb{R}^{14n \times 14n}$ where n is the length of each input vector $\mathbf{x}$.

As derived above in Section 2.7.3, there are $19n+3n^2$ non-zero values in G and $14n$ non-zero values in H. The CSC representation for G would then contain $(19n+3n^2)+(19n+3n^2)+14n = 6n^2+52n$ values, and similarly, the CSC representation for H would contain $(14n) + (14n) + 14n = 42n$ values. The table below contains the number of values in the dense representations and the sparse representations of G and H.

These values are computed based on the formulas provided for each of the matrix representations (dense and sparse) for G and H.

n	G (dense, $196n^2$)	G (sparse, $6n^2+52n$)	H (dense, $196n^2$)	H (sparse, $42n$)
10	19600	1120	19600	420
50	490000	17600	490000	2100
100	1960000	65200	1960000	4200
576	65028096	2020608	65028096	24192
1000	196000000	6052000	196000000	42000

Table 4: Number of values in the sparse and dense representations of the Sequential2D SRU matrices.

Aggregated over both G and H, there are $196n^2 + 196n^2 = 392n^2$ values in the dense representations, and just $(6n^2+52n)+(42n) = 6n^2+94n$ values in the sparse representation. As n grows large,

$$\lim_{n\to\infty} \frac{(6n^2+52n)+(42n)}{(196n^2)+(196n^2)} = \frac{6}{392} \approx 1.5\% \tag{26}$$

The CSC representation requires 1.5% of the space used by the dense representation. We also only need to do the non-zero operations, so the CSC representation requires just 9.2% of the time required by the dense representation, as computed in Equation 25 in the previous section.

2.8.2 Parallelization

Another potential improvement to Sequential2D neural networks is to aggregate weight matrices in the same column so they can be computed in parallel. This is already done in the Simple Recurrent Unit network, but could also be extended to more generalized neural networks. For example, consider the network below.

$$F\bar{\mathbf{x}} = F \cdot \begin{bmatrix} \mathbf{x} \\ \mathbf{h} \\ \mathbf{y} \end{bmatrix} = \begin{matrix} \\ \sigma(\\ \sigma(\end{matrix} \begin{bmatrix} I & 0 & 0 \\ W^{00} & W^{01} & 0 \\ W^{10} & W^{11} & 0 \end{bmatrix} \begin{matrix} \\ ) \\ ) \end{matrix} \cdot \begin{bmatrix} \mathbf{x} \\ \mathbf{h} \\ \mathbf{y} \end{bmatrix} \tag{27}$$

Suppose that the total number of non-zero entries across all the weight matrices is fixed. $W^{00}\mathbf{x}$ and $W^{10}\mathbf{x}$ can be computed in parallel, as can $W^{01}\mathbf{h}$ and $W^{11}\mathbf{h}$. As a result, keeping most of the weights in one column, i.e. W^{00} and W^{10}, and having very few weights in the other column, i.e. W^{01} and W^{11}, will compute faster than spreading the weights evenly across the columns.

In practice, we can achieve this uneven distribution of weights by forcing the two columns to have different sparsities. For example, suppose we fix the sparsity of W^{01} and W^{11} to be 10%. We can do this by randomly selecting 90% of the values within W^{01} and W^{11} and permanently setting them to zero. Even as the network trains, those values remain zeroes, so the sparsity of the matrices remains 10%. Let $\mathbf{x}$ and $\mathbf{h}$ both be in $\mathbb{R}^n$. If we similarly fix the sparsity of W^{00} and W^{10} to 90%, then the total number of weights is $0.1(n^2 + n^2) + 0.9(n^2 + n^2) = 2n^2$. This is the same total weights as if we had fixed all of the matrices to 50% sparsity. The 10%-90% split allows for more parallelization than the 50%-50% split, and through our experiments, we determine whether the uneven distribution of weights adversely affects the train/test performance of a neural network.

2.9 Low Rank Matrix Representations

In addition to removing zeroes by representing Sequential2D weight matrices in Compressed Sparse Column format, another way to reduce the size of the weight matrices is to replace them with low-rank matrix approximations. This technique has already been attempted with reasonable success to improve deep neural network performance [23] and to compress extremely large neural networks such as large language models [24]. In this case, consider a weight matrix $W \in \mathbb{R}^{m\times n}$ that is intrinsic to a model's architecture, such as in a neural network layer. The traditional method directly employs W during training and inference, which can be computationally intensive and memory-demanding, especially for large m and n.

To address this, W can be redefined as the product of two lower-dimensional matrices A and B, where $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$, with k being significantly smaller than both m and n ($k \ll m, n$). This formulation replaces the original weight matrix with:

$$W \approx A \cdot B \tag{28}$$

Here, A and B are initialized as part of the model's parameters and are subject to optimization during the training process, just like any traditional weight matrix. This restructuring not only reduces the number of free parameters, thereby diminishing the model's memory footprint but also reduces the number of multiplications required to compute $W\mathbf{x}$ for an input vector $\mathbf{x} \in \mathbb{R}^n$. [23] By the associative property of matrix multiplication,

$$W\mathbf{x} = (AB)\mathbf{x} = A(B\mathbf{x}) \tag{29}$$

Computing $B\mathbf{x} \in \mathbb{R}^k$ requires $k \times n$ multiplications. Then, left-multiplying by A requires an additional $m \times k$ multiplications. In total, the low-rank multiplication requires $m \times k + k \times n$ multiplications. Recall that the dense matrix multiplication requires mn multiplications. By algebra,

$$m \times k + k \times n < mn \;\forall\; k < \frac{mn}{m+n} \tag{30}$$

Therefore when $m = n$, the low-rank matrix multiplication requires fewer individual multiplications for any rank $k < \frac{n^2}{n+n} = \frac{n}{2}$.

Furthermore, this approach can potentially enhance the model's generalization capabilities by mitigating overfitting, as the rank constraint on W limits the complexity it can represent. Thus, by integrating this low-rank matrix factorization strategy directly into the model architecture, we can achieve a more efficient and potentially more robust model. [23]

2.9.1 Comparing Sparse and Low Rank Matrix Representations

Both sparse matrices and low-rank matrix approximations provide a way to reduce the memory footprint of a matrix with fixed dimensions. That said, they do so in distinctly different ways.

Recall that the SRU matrix G contains three weight matrices $W, W_f, W_r \in \mathbb{R}^{n\times n}$. Suppose, we want to halve the storage space required for each weight matrix. In other words, we aim to store each matrix with just $\frac{1}{2}n^2$ values. As described in Section 2.8.1, the Compressed Sparse Column (CSC) representation of each weight matrix contains $j + j + n$ entries, where j is the number of non-zero entries and n is the width of the matrix. Setting $j + j + n = \frac{1}{2}n^2$, we find that each matrix must contain $j \approx \frac{1}{4}n^2 - \frac{1}{2}n$ non-zero entries.

In practice, we apply this by randomly choosing $n^2 - (\frac{1}{4}n^2 - \frac{1}{2}n)$ entries and fixing them at zero.

Now, suppose that we replaced the weight matrices $W, W_f, W_r \in \mathbb{R}^{n\times n}$ with low-rank approximations AB, A_fB_f, A_rB_r. Let $A, A_f, A_r \in \mathbb{R}^{n\times k}$ and $B, B_f, B_r \in \mathbb{R}^{k\times n}$ where k is the hidden rank. Then, each low-rank matrix approximation contains $n \times k + k \times n$ entries. Setting $2kn = \frac{1}{2}n^2$, we find that the hidden rank must be $k \approx \frac{1}{4}n$ to store each weight matrix in half as many values.

Although this successfully balances the storage space of the sparse and low-rank weight matrices, in practice, we are more interested in balancing the number of weights. Furthermore, weight matrices are not always square.

Suppose instead of halving storage space, we want to halve the number of weights stored in a weight matrix $W \in \mathbb{R}^{m\times n}$. In CSC format, we simply set the number of non-zero elements $j = \frac{1}{2}mn$. For the low-rank approximations, we want $m \times k + k \times n = \frac{1}{2}mn$, so $k \approx \frac{mn}{2(m+n)}$. This setting allows us to compare the speed and training performance of sparse and low-rank weight matrices with the same representational capacity.

In this case, we halve the number of weights in each weight matrix, leaving 50% as many weights. In general, we can create sparse representations and low-rank approximations with E% of the original weights, where $E \in [0, 100]$. Henceforth, we refer to E as the effective weight percentage of the matrix.

3 Results

3.1 Comparing Sparse, Low-Rank, and Dense Matrix Multiplication

We first test the computing speed of sparse, low-rank, and dense matrices of varying sizes. To do so, we randomly initialize a dense matrix M with E% of the entries set to zero. For each matrix M, we measure the time required to calculate $M\mathbf{x}$ for a randomly initialized vector $\mathbf{x}$. We also measure the time for $M_s\mathbf{x}$ where M_s is the CSC representation of M, and we measure $M_{lr}\mathbf{x}$ where M_{lr} is a low-rank substitution for M with effective weight percentage E.

We also measure the time required to compute `LinearModel(x)`, `SparseModel(x)`, and `LowRankModel(x)` where `LinearModel`, `SparseModel`, and `LowRankModel` are PyTorch Linear models with the same initialization as M, M_s, and M_{lr} respectively. Note, `SparseModel` is a modified Pytorch Linear which implements the `PySparse` library for sparse matrix multiplication.

First, we explore how the problem size $m \times n$ affects the computing speed of dense, sparse, and low-rank matrix multiplication on a single GPU for a fixed weight percentage $E = 1\%$.

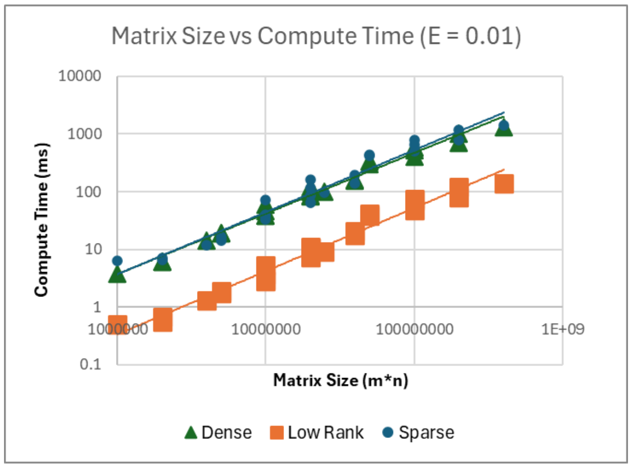

Figure 4: Log-log scale plot of the compute time of matrix multiplications of various sizes with 1% of entries set to non-zero.

The low-rank multiplications are approximately 10 times faster than the dense multiplications. However, there is no significant difference in speed between the sparse and dense multiplications. We hypothesize that the difference between sparse and dense multiplication will become more apparent at a lower effective weight percentage $E = 0.1\%$.

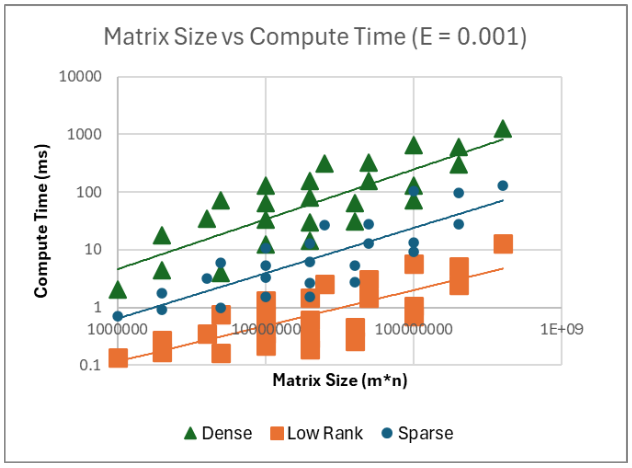

Figure 5: Log-log scale plot of the compute time of matrix multiplications of various sizes with 0.1% of entries set to non-zero.

At $E = 0.1\%$, the sparse multiplication operates faster than the dense multiplication. However, the low-rank multiplication remains the fastest by an order of magnitude. For both weight percentages, the Pytorch wrapper models `LinearModel(x)`, `SparseModel(x)`, and `LowRankModel(x)` follow the same trend as the raw weight matrix multiplications.

3.2 MNIST Random Anomaly

We next investigate whether iterative neural networks (INNs) using low-rank or sparse weight matrices ca perform on par with dense matrices on a supervised learning problem. The MNIST handwritten digits data is a common and widely available benchmark for evaluating the performance of machine learning algorithms. [7] It consists of 60,000 28×28 pixel images of handwritten digits (0-9). However, for our purposes, we need to test large deep-learning models where the speed difference between the various matrix representations is significant enough to provide a high signal-to-noise ratio. Furthermore, we desire a problem with sequential data, to explore the effect of recurrence on iterative neural network performance.

For these reasons, we test our models against the MNIST Random Anomaly problem built out of the MNIST handwritten digits dataset. [4] Each point in the Random Anomaly dataset consists of ten images drawn from MNIST, all corresponding to the same integer. Note, the images are all resized from 28×28 to 50×50 pixels. Nine images have the same combination of random transformations applied, while the remaining image has a different random set of transformations applied. The transformations include distortion, random erasing, added Gaussian noise, and/or normalization. The output is the index of the differently altered image. This problem is significantly harder than MNIST digit classification. Furthermore, it requires passing in a sequence of 10 digits rather than just a single digit making it suitable for iterative neural networks with recurrence. [4]

3.2.1 Model Architecture

For the following experiments on MNIST Random Anomaly, we fix the iterative neural network to have three hidden layers, each with 434 elements. This means the input to each network is,

$$\bar{\mathbf{x}} = \begin{bmatrix} \mathbf{x} & \mathbf{h_1} & \mathbf{h_2} & \mathbf{h_3} & \mathbf{y} \end{bmatrix}^T \tag{31}$$

Here, the input is $\mathbf{x} \in \mathbb{R}^{2500}$, the hidden layers are $\mathbf{h_1}, \mathbf{h_2}, \mathbf{h_3} \in \mathbb{R}^{434}$, and the output is $\mathbf{y} \in \mathbb{R}^{10}$. Therefore, the total length of each input sequence is $2500 + 434 \times 3 + 10 = 3812$ elements. Note that the networks do not only contain feed-forward layers. In general, the INNs takes the form,

$$F \cdot \bar{\mathbf{x}} = \begin{array}{c|ccccc|} & 2500 & 434 & 434 & 434 & 10 \\ 2500 & I & 0 & 0 & 0 & 0 \\ 434 & R_{10} & R_{11} & R_{12} & R_{13} & R_{14} \\ 434 & R_{20} & R_{21} & R_{22} & R_{23} & R_{24} \\ 434 & R_{30} & R_{31} & R_{32} & R_{33} & R_{34} \\ 10 & R_{40} & R_{41} & R_{42} & R_{43} & R_{44} \end{array} \cdot \begin{bmatrix} \mathbf{x} \\ \mathbf{h_1} \\ \mathbf{h_2} \\ \mathbf{h_3} \\ \mathbf{y} \end{bmatrix} \tag{32}$$

Here, R represents matrices with $E\%$ of their entries trainable and the rest fixed at zero. The labels on the left and top of the matrix indicate the dimensions of each weight matrix. Note that the first row only contains an identity matrix, so the original input is preserved throughout every model.

In the following experiments, we study the iterative neural networks by varying the effective weight percentage $E\%$ and using different implementations for the weight matrices R (i.e. sparse, low-rank, or dense).

3.3 Effect of Unequal Weight Distribution on Model Performance

To study the impact of unequally distributing the weights among columns, we split the weights as follows. We distinguish the weights that are multiplied by the input layer as R_i, those multiplied by the hidden layer as R_h, and those multiplied by the output layer as R_o. Our iterative neural network is,

$$F = \begin{array}{c|ccccc|} & 2500 & 434 & 434 & 434 & 10 \\ 2500 & I & 0 & 0 & 0 & 0 \\ 434 & R_i & R_h & R_h & R_h & R_o \\ 434 & R_i & R_h & R_h & R_h & R_o \\ 434 & R_i & R_h & R_h & R_h & R_o \\ 10 & R_i & R_h & R_h & R_h & R_o \end{array} \tag{33}$$

In total, there are $2500 \times (434 \times 3 + 10) = 3,280,000$ R_i weight matrices. Similarly, there are $(434 \times 3) \times (434 \times 3 + 10) = 1,708,224$ R_h weight matrices, and $10 \times (434 \times 3 + 10) = 13,120$ R_o weight matrices. Suppose 20% of all the weights are trainable. Then, in total, there are $\frac{3,280,000+1,708,224+13,120}{3} =$ $1,000,268$ trainable weights. Now, let the effective weight percentage for R_i be E_i and the effective weight percentage for R_h be E_h. To keep the number of weights constant while varying E_i and E_j, we must satisfy, $3,280,000E_i + 1,708,224E_h + 13,120 \times 0.20 = 1,000,268$. From this equation, we can compute different settings of E_h and E_i that keep the total number of weights constant as shown in Table 5 below.

Total params	E_i	E_h	E_o
1,000,268	0	0.584	0.2
1,000,268	0.001	0.582	0.2
1,000,268	0.01	0.565	0.2
1,000,268	0.05	0.488	0.2
1,000,268	0.1	0.392	0.2
1,000,268	0.2	0.2	0.2
1,000,268	0.225	0.152	0.2
1,000,268	0.25	0.104	0.2
1,000,268	0.275	0.056	0.2
1,000,268	0.3	0.008	0.2

Table 5: Effective weight percentage settings for R_i and R_h that keep the total number of weights at 1,000,268.

Note that the effective weight percentage E_o corresponding to the output column is always 20% to ensure that the output layer is never too sparse. By varying E_i and E_h but keeping the total number of weights constant, we can isolate the effect of concentrating more weights in fewer columns. Recall from Section 2.7.3 that weights in the same column can be multiplied in parallel, so if the weights are concentrated in fewer columns, the INN becomes more parallelizable.

We test the ten models as described in Table 5 on the Random Anomaly task. In this case, we implement the weights as dense matrices multiplied by a binary elementwise mask to force $100 - E\%$ of the entries to zero. The results show that keeping the weights evenly distributed is not optimal. Setting E_i to 25% and E_h to 10.4% achieves the best results. This setting is also more parallelizable than $E_i = E_h = 20\%$, as described above.

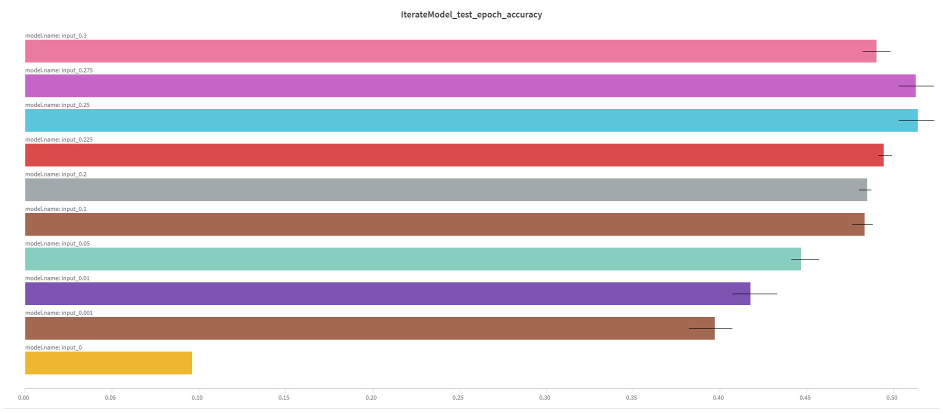

Figure 6: Model performance on MNIST Random Anomaly with an unequal distribution of weights among columns.

3.4 Effect of Recurrence on Model Performance

Next, we test whether there is an optimal ratio of feed-forward and recurrent connections for model performance. To do so, we split the weights as follows. We distinguish the weights that feed information forward as R_l, those that keep information on the same level as R_d, and recurrent weights that send information backward as R_u. Our iterative neural network is,

$$F = \begin{array}{c|ccccc|} & 2500 & 434 & 434 & 434 & 10 \\ 2500 & I & 0 & 0 & 0 & 0 \\ 434 & R_l & R_d & R_u & R_u & R_u \\ 434 & R_l & R_l & R_d & R_u & R_u \\ 434 & R_l & R_l & R_l & R_d & R_u \\ 10 & R_l & R_l & R_l & R_l & R_d \end{array} \tag{34}$$

In total, there are 3,858,088 R_l weight blocks, 565,168 R_d weight blocks, and 578,088 R_u weight blocks. Suppose that 20% of all blocks are trainable, then there are 1,000,268 trainable blocks in total. Let the R_u effective weight percentage be E_u% and the R_l effective weight percentage be E_l%. To keep the number of weights constant while varying E_u and E_l,, we must satisfy, $3,858,088E_l + 578,088E_u + 565,168 \times 0.20 = 1,000,268$. From this equation, we can compute different settings of E_u and E_r that keep the total number of weights constant as shown in Table 6 below.

Total parameters	ρ_u	ρ_l	ρ_d
1,000,268	0	0.23	0.2
1,000,268	0.001	0.2298	0.2
1,000,268	0.01	0.228	0.2
1,000,268	0.05	0.222	0.2
1,000,268	0.1	0.215	0.2
1,000,268	0.2	0.2	0.2
1,000,268	0.4	0.17	0.2
1,000,268	0.8	0.11	0.2
1,000,268	1	0.08	0.2

Table 6: Number of values in the sparse and dense representations of the Sequential2D SRU matrices.

Note that the effective weight percentage E_d corresponding to the main diagonal is always 20%, since these weights are neither feed-forward nor recurrent. By varying E_l and E_u but keeping the total number of weights constant, we can isolate the effect of recurrence on the network.

We test the ten models as described in Table 6 on the Random Anomaly task. As with the previous experiment, we implement the weights as dense matrices multiplied by a binary elementwise mask to force

$100 - E\%$ of the entries to zero.

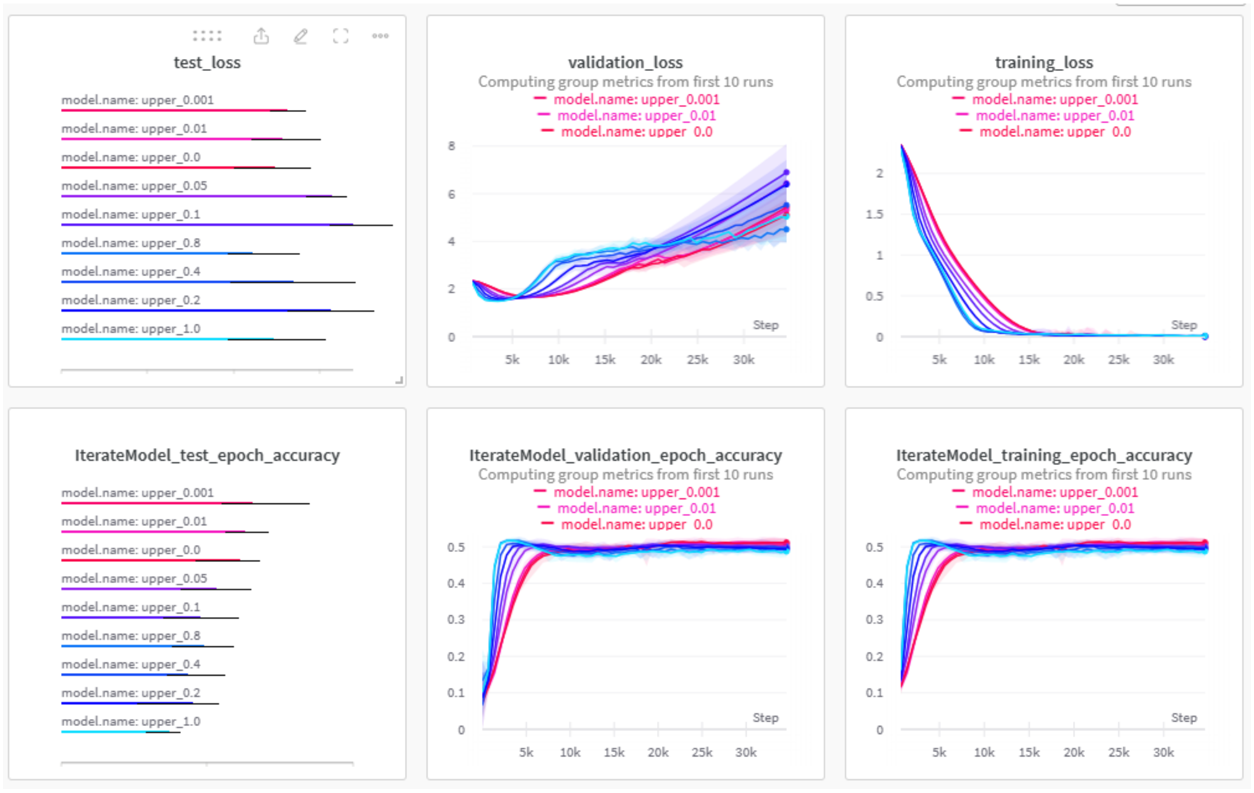

Figure 7: Model performance on Random Anomaly of networks with no recurrence (red) to full recurrence (cyan).

The validation loss per epoch shows that models with more recurrent connections learn faster than those with more feed-forward connections. However, models with less recurrent connections overfit less and perform better on the test set. This suggests that the recurrent connections increase the model variance more than an equal number of feed-forward connections.

3.5 Comparing Model Performance of Sparse, Low-Rank, and Dense Weight Matrices

Lastly, we study whether using CSC or low-rank representations for weight matrices significantly affects their performance. Due to training time constrants, we replace the three hidden layers of length 434 with two hidden layers of 300 and 20 elements, respectively. The resulting INNs take the form,

$$F \cdot \bar{\mathbf{x}} = \begin{array}{c|cccc|} & 2500 & 300 & 20 & 10 \\ 2500 & I & 0 & 0 & 0 \\ 300 & R_{10} & R_{11} & R_{12} & R_{13} \\ 20 & R_{20} & R_{21} & R_{22} & R_{23} \\ 10 & R_{30} & R_{31} & R_{32} & R_{33} \end{array} \cdot \begin{bmatrix} \mathbf{x} \\ \mathbf{h_1} \\ \mathbf{h_2} \\ \mathbf{y} \end{bmatrix} \tag{35}$$

We experiment with using different matrix representations (dense, sparse, low-rank) for all of the weights R as depicted in Equation 35 and adjusting the effective weight percentage $E\%$. The one exception is R_{32}, which is always fully dense so that the output $\mathbf{y}$ is non-zero.

For each setting, we train on MNIST Random Anomaly 25 times to account for noise. Since we are interested in whether CSC or low-rank weight matrices allow models to learn faster, training time is of more interest than the number of training epochs. For this reason, each model is time-limited to 300 seconds of training.

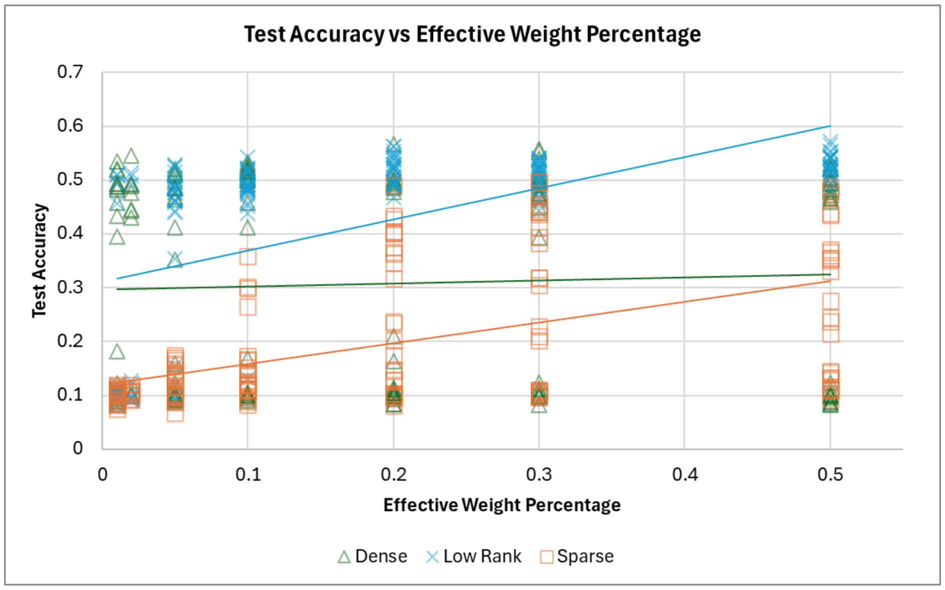

Figure 8: Model performance at various $E\%$ for dense, sparse, and low-rank representations of weight matrices.

Note that the dense INNs always have every parameter trainable. Although they are initialized with only $E\%$ of entries set to non-zero, all of the weights can become nonzero, i.e. $E = 100\%$. This explains the zero-slope trendline for dense test accuracy in Figure 8 above. On average, low-rank weight matrices outperform dense weight matrices at every setting of $E\%$. Moreover, they do so with fewer weights. The figure below contains the training, validation, and test statistics for the various representations at $E = 20\%$.

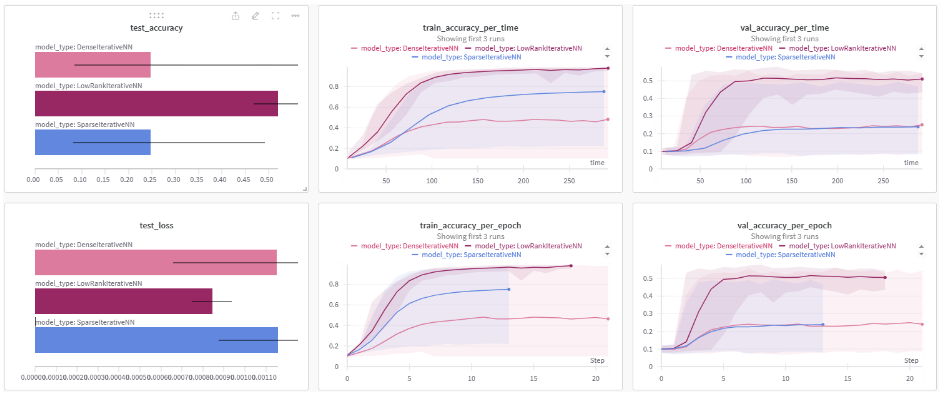

Figure 9: Model performance at $E = 20\%$ for dense, sparse, and low-rank representations of weight matrices.

The table below contains the number of trainable weights at each effective weight percentage.

Effective Weight Percentage (%)	Dense Weights	Low Rank Weights	Sparse Weights
1	933900	16180	10803
2	933900	22380	19978
5	933900	47780	47024
10	933900	94340	90412
20	933900	188430	170789
30	933900	279770	243481
50	933900	467340	368956

Table 7: Number of trainable weights in each weight matrix representation per effective weight percentage.

Notice that since the low-rank representation must have an integer hidden rank, the number of weights is not exactly $E\%$; however, this will suffice for our analysis.

4 Conclusions and Discussion

In our investigation into the effects of various matrix representations and weight distributions within iterative neural networks (INNs) applied to the MNIST Random Anomaly task, we observe several key findings that underscore the delicate balance between computational efficiency, model complexity, and learning performance.

4.1 Computational Efficiency of Matrix Representations

Our initial comparison between computing speeds for sparse, low-rank, and dense matrices reveals that low-rank matrix multiplications are significantly faster than their dense counterparts, with computational times being nearly 10 times shorter for matrices with 1% non-zero entries. This gap widens as the effective weight percentage decreases to 0.1%, where sparse matrix multiplication also begins to outpace dense multiplication, demonstrating the computational advantages of sparse and low-rank matrices in large-scale problems.

4.2 MNIST Random Anomaly Task Performance

The core of our research centers around the MNIST Random Anomaly task, a more complex variant of the traditional MNIST digit classification problem designed to test the models' capacity to identify anomalies in sequences of transformed digit images.

4.2.1 Effect of Low-rank and Sparse Matrix Representations

In our experiments with varying the effective weight percentage (E%), low-rank matrix representations consistently match or outperform dense matrices in test accuracy across the board, while requiring far fewer trainable parameters. This is evident at $E = 10\%$, where low-rank matrices achieve test accuracies comparable to dense matrices but with only 94,340 trainable parameters compared to 933,900 in dense matrices, underscoring the parameter efficiency of low-rank approximations.

There are several possible explanations for the success of low-rank matrix representations. Firstly, low-rank matrices are essentially a compressed representation of the original data, capturing the most critical information or patterns within the dataset. [23] If the data or the relationships within the data can be effectively captured in a lower-dimensional space, then low-rank matrices can perform as well as dense matrices, because they're essentially distilling the data to its most informative components. [25]

Notice that the low-rank matrix INNs also learn faster than the dense matrix INNs, as evidenced by the sharper increase in validation accuracy over time (see Figure 9). During training, the low-rank model can allocate its capacity more precisely, adjusting a smaller set of parameters to capture the underlying data distributions, which can lead to faster convergence and potentially better generalization. [25]

Utilizing low-rank matrices can also act as a form of regularization, reducing the likelihood of overfitting to the training data. By limiting the number of free parameters, low-rank matrices constrain the model, forcing it to focus on the most salient features in the data. This can lead to models that generalize better to unseen data, contributing to comparable or sometimes superior test accuracies. For tasks where the underlying data structure is inherently low-rank or can be well approximated in a lower-dimensional space, low-rank matrices can naturally excel. [26] The MNIST dataset, for example, despite its high-dimensional input space, is believed to have an underlying structure that can be captured in fewer dimensions, making low-rank approximations particularly effective. [7]

Similarly, low-rank approximations can help filter out noise in the data. By focusing on the primary structure or patterns in the dataset and disregarding components that contribute less variance, low-rank matrices might be discarding noisy, less informative aspects of the data, which can lead to improved model performance on test data. [25]

4.2.2 Effect of Weight Distribution within INNs

The distribution of weights within the INNs also plays a crucial role in model performance. Our findings indicate that models with a higher concentration of weights in the input column of the INN ($E_i = 25\%$ and $E_h = 10.4\%$) achieve better results on the MNIST Random Anomaly task. In addition, these models theoretically benefit from increased parallelizability. This suggests that prioritizing the forward flow of information can enhance both the performance and computational efficiency of neural networks on certain tasks.

From the training and validation loss over time (see Figure 9), the exploration into the optimal ratio of feed-forward and recurrent connections reveals that while more recurrent connections facilitate faster learning, they also lead to greater overfitting. Conversely, models with a greater emphasis on feed-forward connections demonstrate slower learning rates but ultimately achieve better generalization on the test set. This trade-off highlights the need for a balanced architecture that supports both efficient learning and robust generalization.

One possible explanation for this result is that recurrent connections increase a model's capacity to capture sequential dependencies within the data due to their ability to maintain and update a hidden state over time. While this increased capacity can accelerate learning by leveraging these dependencies, it also raises the model's complexity, potentially leading to quicker overfitting on the training data if not properly regularized or if the training data is not diverse enough to support the additional complexity. [17] By focusing more on feed-forward connections, a model might adopt a simpler representation of the data that hinges more on the present input's salient features rather than intricate sequential patterns. This simplification may lead to better generalization on unseen data, as the model is less likely to overfit to noise or to specific sequential artifacts present in the training set.

4.3 Concluding Insights

In summary, our study provides compelling evidence that leveraging sparse and low-rank matrix representations within INNs can lead to substantial gains in computational efficiency without sacrificing performance. The careful distribution of weights and the strategic balance between feed-forward and recurrent connections enhance the models' ability to learn effectively and generalize well.

This research contributes valuable insights into the ongoing pursuit of optimizing neural network architectures for enhanced performance and efficiency. By exploring the benefits of sparse and low-rank matrices and the effects of weight distribution and recurrent connections, our work paves the way for the development of more sophisticated and resource-efficient iterative neural network models, capable of tackling complex tasks with high accuracy and computational efficiency.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[2] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015.

[4] Q. Hershey, "Exploring neural network structure through iterative neural networks: Connections to dynamical systems," master's thesis, Worcester Polytechnic Institute, Worcester, MA, 3 2023. Advisor: Professor Randy Paffenroth, Reader: Professor Seyed Zekavat.

[5] H. N. Pathak, R. Paffenroth, and Q. Hershey, "Sequentia12d: Organizing center of skip connections for transformers," in *2023 International Conference on Machine Learning and Applications (ICMLA)*, pp. 362–368, 2023.

[6] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer, "A comparison of several bandwidth and profile reduction algorithms," *ACM Trans. Math. Softw.*, vol. 2, p. 322–330, dec 1976.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] Q. Hershey, R. Paffenroth, H. Pathak, and S. Tavener, "Rethinking the relationship between recurrent and non-recurrent neural networks: A study in sparsity," 2024.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, pp. 318–362, 1986.

[10] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, 2010.

[11] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," tech. rep., Neural Networks, Tricks of the Trade, Springer, 1998.

[12] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *AISTATS*, 2010.

[13] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, pp. 179–211, 1990.

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[15] J. Bradbury, S. Merity, C. Xiong, and R. Socher, "Quasi-recurrent neural networks," 2016.

[16] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014.

[17] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*, pp. 1310–1318, 2013.

[18] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[19] B. Li, E. Zhou, B. Huang, J. Duan, Y. Wang, N. Xu, J. Zhang, and H. Yang, "Large scale recurrent neural network on gpu," in *2014 International Joint Conference on Neural Networks (IJCNN)*, pp. 4062–4069, 2014.

[20] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, pp. 157–166, 1994.

[21] T. Lei, Y. Zhang, S. I. Wang, H. Dai, and Y. Artzi, "Simple recurrent units for highly parallelizable recurrence," 2018.

[22] T. Lei, Y. Zhang, and Y. Artzi, "Training RNNs as fast as CNNs," 2018.

[23] S. R. Kamalakara, A. Locatelli, B. Venkitesh, J. Ba, Y. Gal, and A. N. Gomez, "Exploring low rank training of deep neural networks," 2022.

[24] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021.

[25] V. Sindhwani, T. N. Sainath, and S. Kumar, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4624–4628, IEEE, 2015.

[26] A. Wang, C. Li, M. Bai, Z. Jin, G. Zhou, and Q. Zhao, "Transformed low-rank parameterization can help robust generalization for tensor neural networks," 2023.