Example-Driven Emulation of Lua Execution with Recurrent Neural Networks

UNIVERSITY OF
LINCOLN

Richard Coric
26414124

26414124@students.lincoln.ac.uk

School of Computer Science

College of Science

University of Lincoln

Submitted in partial fulfilment of the requirements for the
Degree of BSc(Hons) Computer Science

Supervisor Dr. Heriberto Cuayahuitl Portilla

May 2025

"It is our choices, not our abilities, that show who we truly are"

- Albus Dumbledore

Acknowledgements

First and foremost, I would like to extend my utmost gratitude to my supervisor, Dr Heriberto Cuayahuitl Portilla, for his support, expertise, and encouragement throughout this dissertation. Heriberto's knowledge and patience, while I learned about the ins and outs of deep learning and language modelling, have been an invaluable asset throughout the year.

Secondly, I would like to thank the University of Lincoln for providing me with the opportunity, experience and confidence to conduct this study. My time here was filled with so many memories; the last three years have been both the most difficult and enriching years of my life. I have met so many people, some of whom will remain in my life forever. I am proud to say I have grown as a person since I began this degree, and I will forever look back on this experience with awe.

To my amazing partner, Oliver, for supporting me and being there for me throughout the research. For sitting through me talking his ears off about deep learning, for wiping my tears when things didn't go to plan. I couldn't have done it without you. Thank you.

I would also like to extend my gratitude to Olivia Rodrigo for making the Guts album, which has gotten me through many periods of life, including battling with CUDA errors and PyTorch matrix dimension misalignments.

Last, but not least, I would like to thank myself. For getting through this dissertation with minimal breakdowns and for maintaining a deep interest in the subject matter, which helped me keep motivated this year.

Abstract

Traditionally, programs have been written and compiled strictly using a deterministic approach via a compiler; such execution can be computationally expensive and difficult to implement. Previous efforts like Codex and AlphaCode use large transformer models with high inference and training times. This work evaluates, Recurrent Neural Networks (RNNs), and minimal RNN architectures with fast training and inference times in emulating Lua execution by learning semantic embeddings, and applies Mixtures of Experts (MoE), with Attention, trained via curriculum learning to enhance generalisation of these models. The study finds that standard models overfit and struggle to learn meaningful patterns, and fail on complex examples whereas minimal versions generalise better and fail on fewer examples. The research finds that it was limited by dataset size and quality, leading to over-fitting and little generalisation. Finally, it contextualises the trade-offs between inference efficiency and output accuracy.

Table of Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Code Execution and Generation . . . 1

1.1.2 Rationale . . . 2

1.1.3 Current Work . . . 3

1.1.4 Problems in Current Research . . . 3

1.2 Aims & Objectives . . . 4

1.2.1 Objectives . . . 4

1.3 Dissertation Structure . . . 5

2 Literature Review **6**

2.1 Background . . . 6

2.1.1 Recurrent Neural Networks . . . 6

Standard RNN . . . 6

Long-Short Term Memory (LSTM) . . . 7

Gated Recurrent Unit (GRU) . . . 8

MinLSTM and MinGRU . . . 9

2.1.2 Mixture of Experts . . . 10

2.2 Related Works . . . 11

2.3 RNNs in Code-Related Tasks . . . 12

2.3.1 Methodologies . . . 12

2.3.2 Evaluation Metrics . . . 12

2.3.3 Limitations . . . 14

2.4 Transformer Models . . . 15

2.4.1 Limitations . . . 17

3 Requirements Analysis **19**

3.1 Functional Requirements . . . 19

3.2 Non-Functional Requirements . . . 20

3.2.1 Hardware Considerations . . . 21

3.2.2 Software Requirements . . . 21

4 Design & Methodology **22**

4.1 Research Philosophy . . . 22
4.2 Project Management . . . 22
4.3 System Design . . . 24
4.3.1 System Architecture . . . 24
4.4 Model Selection . . . 25
4.4.1 RNN . . . 25
4.4.2 LSTM . . . 25
4.4.3 GRU . . . 26
4.4.4 MinLSTM and MinGRU . . . 26
4.4.5 Mixture of Experts . . . 27
4.4.6 Attention . . . 27
4.4.7 Hyperparameter Tuning . . . 28
4.5 Dataset . . . 28
4.5.1 Curriculum Learning . . . 28
4.5.2 Lua Semantics . . . 29
4.6 Benchmarking . . . 30
4.6.1 Evaluation Metrics . . . 30
4.6.2 Baseline Models . . . 32
4.7 Overview of Experiments . . . 32
4.8 Ethical Considerations . . . 32
4.9 Risk Management . . . 33

5 Implementation **34**

5.1 Development Environment and Tools . . . 34
5.2 Language Parsing . . . 34
5.2.1 Building Data . . . 34
5.2.2 Semantic Tokenisation . . . 38
5.3 Dataset . . . 38
5.3.1 Data Splitting . . . 39
5.3.2 Building the Vocabulary . . . 41
5.4 Training Pipeline . . . 41
5.4.1 Language Model . . . 41
Training . . . 41
Sampling . . . 42
Check-pointing . . . 42

5.4.2 Language Model Hyperparameter Tuning . . . 43
5.5 Model Implementation . . . 43
5.5.1 RNN . . . 43
5.5.2 LSTM and GRU . . . 44
5.5.3 MinGRU and MinLSTM . . . 44
5.5.4 Mixture of Experts . . . 45
5.5.5 Attention . . . 45
5.6 Evaluation . . . 46
5.6.1 Sample K . . . 47
5.7 Challenges . . . 48
5.7.1 Overfitting . . . 48
Label Smoothing . . . 48
L2 Regularisation . . . 48
Dropout . . . 49
Teacher Forcing . . . 49
Learning Rate Scheduling . . . 49
5.7.2 Data Quality . . . 49

6 Results & Discussion **51**
6.1 Computational Performance and Resources . . . 51
6.2 Results . . . 52
6.2.1 RNN Performance . . . 52
6.2.2 GRU Performance . . . 53
6.2.3 LSTM Performance . . . 54
6.2.4 MinGRU Performance . . . 54
6.2.5 MinLSTM Performance . . . 55
6.2.6 Inference Times . . . 56
6.2.7 Example Outputs . . . 57
6.3 Discussion . . . 59
6.3.1 Dataset Collection . . . 59
6.3.2 Model Development . . . 60
Architecture Performance . . . 60
Mixture of Experts . . . 60
Alternative Architectures . . . 61
6.3.3 Interpreting Results . . . 61
Generalisation vs Overfitting . . . 61
Comparison with Baselines . . . 61
Errors and Limitations . . . 62

6.3.4 Implications for the Real world 63
6.3.5 Limitations of Study 63

7 Conclusion **64**
7.1 Findings 64
7.2 Contributions 65
7.3 Limitations 65
7.4 Future Work 66
7.5 Self-Reflection 67

References **67**

List of Figures

1.1 An overview of the proposed architecture. Where $RNN*$ stands for any Recurrent Neural Network architecture. 4

2.1 A visual representation of how RNNs pass data. 6
2.2 LSTM Architecture 7
2.3 MinLSTM Architecture 10
2.4 A Visual Comparison of the LSTM and MinLSTM Architecture. . . . 11

4.1 Gantt Chart Outlining the time for each iteration and the due dates for each milestone. 23
4.2 High-Level Class Diagram of the Implementation, of single unit architectures for testing, and the class that rely on them. Also highlighting the libraries each class depends on. 24
4.3 Simple Lua program to print a message. 29
4.4 The equivalent S-Expression for the print statement in 4.3. 29

5.1 UML of Model Architectures and Parent Language Model, modelling the interactions between different classes. 35
5.2 UML of the Tokenisation, Data and Evaluation part of the pipeline. . 36
5.3 Prompt template used to generate Lua code examples, where <> indicate semantic features such as selection and looping. 36
5.4 Simple Program input-output example, of a string function running inside a print statement. 36
5.5 Complex Program input-output example, to find the smallest and highest numbers of an array. 37
5.6 Semantic Tokens for Figure 5.4 38
5.7 Semantic Tokens for Figure 5.5 39
5.8 Chart visualising the data splits and the number of samples per split, using a 70-15-15 split and 677 data samples. 40

5.9 Computation Graph illustrating the execution flow of the Mixture of Experts Layer; the input is passed through a gating network and experts separately, and then combined using a product and sum. . . . 46
5.10 Attention Computation, of the energy or compatibility score of QK^T, the sqrt $\sqrt{d_k}$, its SoftMax and finally scaling by values V 46
5.11 Python PyTorch Implementation of EM 47

6.1 Vanilla RNN shows reduced loss using 2 experts rather than none, however, both models overfit the data. 52
6.2 GRU model training and validation loss plots. 53
6.3 LSTM model training and validation loss plots. 54
6.4 minGRU model training and validation loss plots. 55
6.5 minLSTM model training and validation loss plots. 55
6.6 RNN generation output, has string as the second token, however with such a high perplexity, it can be said the model is guessing. 57
6.7 GRU generation, showed some improvement and generated some strings and a number 58
6.8 LSTM generation was similar to GRU except generated a function call, which aligns with the increased perplexity. 58
6.9 MinLSTM generation, generated strings, identifiers and a function call indicating high perplexity. 58
6.10 MinGRU output is very similar semantically to the MinLSTM. 59

List of Tables

2.1 Summary of research using RNNs in Code-Related Tasks, outlining methods, the metrics used and the results achieved 13
2.2 Summary of Transformer Models in Code-Related Tasks 16

4.1 Architectural Variations on models tested. Values were updated from the base model. Hidden state size (h), number of experts ($expts$), and training steps. The model was evaluated on a Exact Match (EM), F1 Score ($F1$), Sentence Similarity (SS), Pass@1 ($P@1$), Perplexity ($Perp$), Recall (Rcl) and Precision (Prs). 33

6.1 Training times in minutes for each model by number of experts, and the time per iteration 51
6.2 Timings in Seconds for each step of the training pipeline 52
6.3 Experiment Results per model such that hidden state size (h), number of experts ($expts$), and training steps. The model was evaluated on a Exact Match (EM), F1 Score ($F1$), Sentence Similarity (SS), Pass@1 ($P@1$), Validation Perplexity ($Perp$), Recall (Rcl) and Precision (Prs). 56
6.4 Depicts Inference Time of each model expressed in seconds, with respect to small and large code snippets. 57
6.5 Baseline Results for StarCoder and Qwen 2.5 Coder 62

Chapter 1

Introduction

1.1 Motivation

1.1.1 Code Execution and Generation

Traditionally, programs written in any given programming language are written by a human in a text editor or Integrated Development Environment (IDE), and then passed to either a compiler or an interpreter. Both compilers and interpreters will use a Lexer - a piece of software that takes text (in the form of a program) and outputs a set of lexemes or tokens which represent the program. The next step after lexing is syntactic analysis, this process converts the tokens into an internal intermediate representation (IR) or Abstract Syntax Tree (AST), and keeps track of data in virtual tables. It is then common for IR to be either interpreted or converted to machine code through a virtual machine (VM) which emulates the operations of the program, such as arithmetic and jump operations like loops, functions and selection. Traditional code generation from a VM involves either rule-based parsing, where machine code instructions are emitted from an IR or syntax tree. This approach is often very strict and non-adaptive, meaning it is difficult to infer the intent from examples.

While traditional execution pipelines rely on deterministic, manually designed systems, recent developments in neural language modelling have introduced a range of approaches which instead learn from patterns in large code datasets. Modern models like Codex and GPT can generate and provide reasoning over code without the

need for a VM or explicit programming of the language grammar. This move has sparked interest in exploring whether neural models can emulate the execution of a program from examples rather than rules. However, most existing work focuses on large, expensive-to-train transformer models.

1.1.2 Rationale

Neural networks continue to revolutionise various computer science domains, including compiler development, with many works applying machine learning to compilers (Chris Cummins, Grubisic, Baptiste *et al.*, 2024; Leather and C. Cummins, 2020; Imada and Katsuhiko Nakamura, 2008). This research aims to bridge this gap by investigating whether smaller models like RNNs, can still be useful in such a field. In addition to RNNs being smaller models, the LSTM gating mechanism could act similarly to the idea of a virtual table in a compiler. The model could potentially develop a mechanism to maintain a representation of variables in scope while attenuating those that have moved beyond the current scope.

The success of this work could lead to more adaptive implementations of programming languages, with a potential application in domain-specific language development and compiler optimisation. Training a neural network for such a purpose could pose challenges due to its complex nature. However, Lua is an ideal candidate for this research due to its widespread use in embedded systems and its simplicity. This study pushes existing deep learning ideas towards language development, and in the future, towards more adaptive computer systems.

Such a compiler could be applied to code playground software such as Quokka JS [1], which provides instant in-editor code outputs - useful for debugging. A deep learning approach would omit the complexity requirements of running the code to generate an output, and would be able to provide feedback by inference. Computational complexity is, in this way, shifted from language interpretation to model interference.

[1] https://quokkajs.com/

1.1.3 Current Work

Several studies have explored the use of RNNs for code-related tasks such as program synthesis, code completion and error detection, using both word-level and character-level models. Researchers have explored a range of architectures with varying levels of effectiveness, most commonly LSTM, vanilla RNN, GRU or combinations forming encoder-decoder architectures. These studies train models to learn and emulate simple algorithms for copying, sorting, and reversing lists of numbers.

A number of researchers demonstrated that example-based learning – using input-output pairs – gave the models sufficient information to learn the relation between the input program and the output sequence. Training strategies often incorporate curriculum or reinforcement learning to improve the model's ability to generalise to more complex scenarios.

1.1.4 Problems in Current Research

Despite the success on short sequences, many of these models fail to generalise to longer, more complex input sequences. Longer sequences allow the model to use context from an increasing number of time steps ago. In practice, this means variables defined earlier in the program flow can be remembered for a longer time, making tools for compiler-free compilation more accurate.

Additionally, many studies have found that models succeed in learning syntax but fail in any semantic understanding. Language semantics give rise to understanding what the code is doing without being influenced by any form of noise from syntax or syntax sugar [2]. Therefore, such a system could be made universal across multiple programming languages - special semantics from semantically different languages can be learned as extra rules on top of the compiler.

Figure 1.1: An overview of the proposed architecture. Where $RNN*$ stands for any Recurrent Neural Network architecture.

1.2 Aims & Objectives

This project aims to develop a system, outlined in Figure 1.1 that can, given examples of code, predict the output of low (those with simple structures such as 'print' and 'if') and medium (those with more complex structures such as loops) and high (those with functions and tables) complexity programs, assuming they terminate. This is done to evaluate the effectiveness of the method.

1.2.1 Objectives

1. Collect a suitable and substantial enough dataset of at least 500 examples of Lua code in order to train a machine-learning model.
2. Develop an LSTM model, embedding model and lexer for parsing Lua, and then learn the model's embeddings.
3. Evaluate and appraise other methods of achieving an SLM (Small Language Model) compiler, which have already been researched.
4. Conduct a literature review to gain a critical understanding of work that has already been done in this field.
5. Train different RNN models on embeddings created from examples of code and outputs, allowing them to find patterns and predict the output of code.
   (a) Focusing on standard RNNs, LSTM, GRUs, and minimal versions of RNN and LSTM.
6. Test and evaluate the model based on new examples of Lua code, and perform relevant evaluation metrics such as Perplexity and F1 Score.

[2]Syntax that is purely for readability rather than functionality of the program.

1.3 Dissertation Structure

The remainder of this dissertation will focus on implementing and evaluating a range of models for a subset of Lua programs. The next chapter will evaluate the state of current research, looking more specifically at RNN and Transformer-based systems. It determines where it falls short and how this research addresses these shortcomings. Following this, Chapter Three will focus on the software and hardware requirement analysis for this work. Chapter Four explains the methods and approaches behind emulating Lua execution via neural network architectures; the next chapter, Chapter Five, will give a technical overview of the implementation models and the training. Chapter Six will analyse and provide the results of experiments run in this work, concluding with the success and limitations of the approach in Chapter Seven.

Chapter 2

Literature Review

2.1 Background

2.1.1 Recurrent Neural Networks

Standard RNN

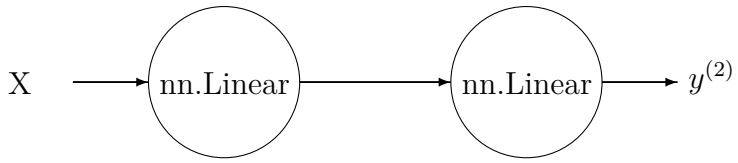


Figure 2.1: A visual representation of how RNNs pass data.

The standard Recurrent Neural Network (RNN) is characterised by its feedback connections wherein hidden states are propagated across sequential timesteps. The network outputs recur, as illustrated by a perceptron: the output from the first network (2.1) is passed as an input into the second (2.2). This is represented visually as two Pytorch Linear Layers in Figure 2.1.

$$h^{(1)} = \sigma(XW + B) \tag{2.1}$$

$$y^{(2)} = \sigma(h^{(1)}W + B) \tag{2.2}$$

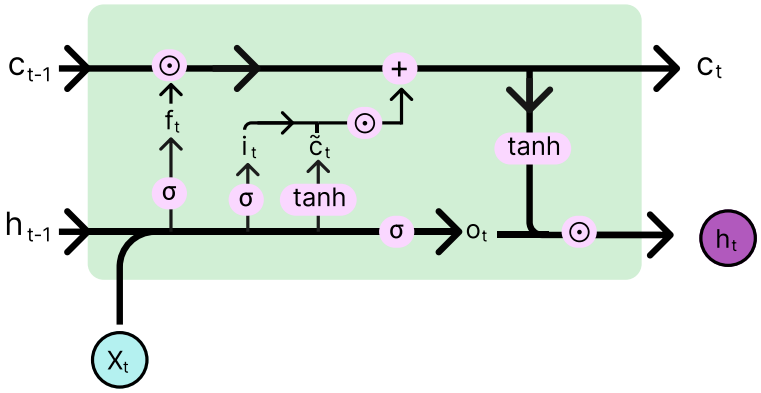


Figure 2.2: LSTM Architecture

Long-Short Term Memory (LSTM)

RNN networks suffer from vanishing gradients because partial gradients become smaller between layers during backpropagation, until their values do not affect the output vector. To solve this, Hochreiter and Schmidhuber (1997) introduced the Long-Short Term Memory architecture, which presents the *Cell State* along with its hidden state. The LSTM works via a gating mechanism; different gates allow it to read (*input*), write (*output*), and erase (*forget*) information.

$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f) \tag{2.3}$$

$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i) \tag{2.4}$$

$$o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o) \tag{2.5}$$

For a sequence of inputs $x^{(t)}$ at time step t, the LSTM computes several states: the forget gate (2.3), which controls what information is kept and forgotten from the previous cell state. The input gate (2.4) determines which information from the new cell content is written to the cell state, and the output gate (2.5) controls what information is written to the hidden state.

$$\tilde{c}^{(t)} = \tanh(\mathbf{W}_c h^{(t-1)} + \mathbf{U}_c x^{(t)} + b_c) \tag{2.6}$$

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \tilde{c} \tag{2.7}$$

$$h^{(t)} = o^{(t)} \odot \tanh \tilde{c}^{(t)} \tag{2.8}$$

Equation 2.6 computes the new content to be written, which becomes part of the cell state computation (2.7) and part of the hidden state (2.8), responsible for reading the output. The process is illustrated in Figure 2.2.

Gated Recurrent Unit (GRU)

The Gated Recurrent Unit (GRU), first introduced by Cho *et al.* (2014), represents a significant refinement on the standard RNN architecture. It was designed as a simplified alternative to the LSTM model, while maintaining the ability to learn long-term dependencies through a gating mechanism.

$$r_j = \sigma\left([\mathbf{W}_r x]_j + [\mathbf{U}_r h_{\langle t-1 \rangle}]_j\right) \tag{2.9}$$

$$z_j = \sigma([\mathbf{W}_z x]_j + [\mathbf{U}_z h_{\langle t-1 \rangle}]_j) \tag{2.10}$$

$$h_j^{\langle t \rangle} = z_j h_j^{\langle t-1 \rangle} + (1 - z_j)\tilde{h}_j^{\langle t \rangle} \tag{2.11}$$

$$\tilde{h}_j^{\langle t \rangle} = \phi([\mathbf{W}\mathbf{x}]_j + [\mathbf{U}(\mathbf{r} \odot h_{\langle h-1 \rangle})]_j) \tag{2.12}$$

The study describes the activation for the j-th hidden layer as a computation through a sequence of gates. First, the reset gate (equation 2.9) determines whether the previous hidden states should be ignored with the current input vector; then the update gate (equation 2.10) controls how much information the previous hidden state will save to the current hidden state. The combination of these gates (equation 2.11) computes the current hidden state. For the candidate hidden state (equation 2.12), the common activation function (ϕ) is often a hyperbolic tangent function (tanh).

MinLSTM and MinGRU

The previously mentioned RNN-based architectures are not parallelisable; consequently, Feng, Tung, Ahmed *et al.* (2024) introduced minimal architectures with removed hidden states while incorporating normalisation techniques, thereby circumventing the need for the computational drawbacks of Backpropagation Through Time (BPTT).

Blelloch (1990) uses the term *prefix scan* to refer to a special type of associative scan algorithm which efficiently computes prefix sums in two steps known as up-sweep and down-sweep. The computation broadly resembles that of the GRU hidden state recurrence shown in equation 2.11. Its recurrence is dependent on the previous hidden state h_{t-1} (equation 2.10), and therefore prefix scanning is not applicable, since its inputs depend upon knowing all the outputs at once as opposed to over time.

To use prefix scan, the study removed the dependency on the previous hidden state (h_{t-1}). It removed the reset gate entirely since it is no longer needed to control the weight of the previous hidden state. An updated minGRU is defined as an update gate (equation 2.14) and candidate hidden state (equation 2.15). In standard GRU and LSTM models, the tanh function stabilises training and helps avoid vanishing gradients; sigmoid activations are then applied to the hidden state. Eliminating hidden states removes the need for the tanh activation, and increases the range of the hidden states from $h \in (-1, 1)$.

$$h_j^{\langle t \rangle} = z_j h_j^{\langle t-1 \rangle} + (1 - z_j)\tilde{h}_j^{\langle t \rangle} \tag{2.13}$$

$$z_j = \sigma([\mathbf{W}_z x]_j) \tag{2.14}$$

$$\tilde{h}_j^{\langle t \rangle} = [\mathbf{W}\mathbf{x}]_j \tag{2.15}$$

The authors also modified the LSTM architecture, shown in Figure 2.3, by removing its hidden states and dropping the candidate state range restriction. Similarly, the output gate is dropped because it scales the hidden state, which is no longer needed.

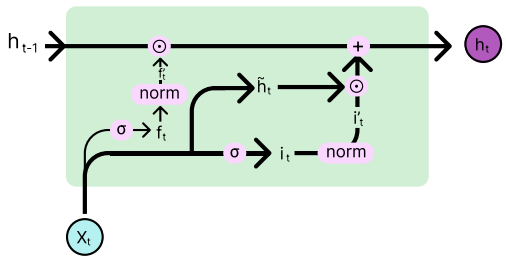


Figure 2.3: MinLSTM Architecture

The lack of an output gate results in the cell state and hidden state being equal, so it is unnecessary, leaving only the hidden state (equations 2.19 and 2.16).

$$h^{(t)} = f^{(t)} \odot h_{t-1} + i^{(t)} \odot \tilde{h}^{(t)} \tag{2.16}$$

$$f^{(t)} = \sigma(W_f x^{(t)} + b_f) \tag{2.17}$$

$$i^{(t)} = \sigma(W_i x^{(t)} + b_i) \tag{2.18}$$

$$\tilde{h}^{(t)} = \mathbf{W}_h x^{(t)} + b_h \tag{2.19}$$

2.1.2 Mixture of Experts

Jacobs *et al.* (1991) introduced a learning technique based on how the human brain learns. The researchers called this an associative version of Competitive Learning, a form of unsupervised Learning where Artificial Neural Network (ANN) nodes compete for weight updates to a Subset of the data (Grossberg, 1987). Competitive learning makes each neural node more specialised the deeper it gets in the network. Associative learning is a concept that comes from Hebbian Theory (Hebb, 1950), which explains how the brain learns - two neurons which repeatedly help each other gain a stronger connection.

In deep learning, the researchers named this technique Mixture of Experts (MoE), where different parts of a network train experts on specific subtasks, and a gating network aims to select the most appropriate expert. Additionally, the researchers created a method that picks only one expert per task by scaling the residual $(d^c - o_i^c)$

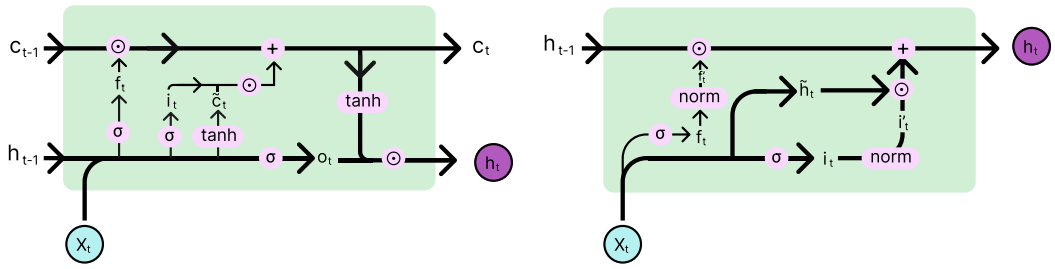


Figure 2.4: A Visual Comparison of the LSTM and MinLSTM Architecture.

by the weight of an expert's contribution (p_i^c) and outputting the whole output vector. The loss function is shown in equation 2.20.

$$E^c = \sum_i p_i^c ||d^c - o_i^c||^2 \tag{2.20}$$

2.2 Related Works

Prior to transformer models (Vaswani *et al.*, 2017), Recurrent Neural Networks (RNNs) were widely used for natural language processing (NLP) due to their ability to process sequential data and their capability to learn longer-term dependencies than Feed-Forward Networks.

There have been many improvements to RNNs, Hochreiter and Schmidhuber (1997) addressed the issue of vanishing gradients and developed the Long-Short Term Memory (LSTM) model, which can keep track of longer-term dependencies via a gating mechanism. Later, the Gated Recurrent Unit (GRU) was introduced as a simpler LSTM architecture in an Encoder-Decoder RNN for Machine Translation.

The transformer model has been the standard for NLP tasks (Patwardhan, Marrone and Sansone, 2023) ever since its introduction in 2017 (Vaswani *et al.*, 2017). RNNs continue to demonstrate usage in specific domains and can reach similar outcomes to a transformer model with the added benefit of parallelisability (Feng, Tung, Ahmed *et al.*, 2024) by updating the architecture to remove hidden states, adding a normal-

isation element (this is illustrated in figure 2.4) thus removing the need for BPTT, allowing the use of parallel algorithms such as parallel prefix-scan (Blelloch, 1990).

2.3 RNNs in Code-Related Tasks

Beyond NLP, RNNs have been used for compiler-like tasks since 2014 with the Neural Turing Machine (Graves *et al.*, 2014). Researchers employed various architectures to achieve their objectives. Common models included standard RNNs, LSTMs, or a combination to create an encoder-decoder structure (Burgueño *et al.*, 2021; Graves *et al.*, 2014; Priya *et al.*, 2017; Rahman, Watanobe and Keita Nakamura, 2020; Zaremba, Mikolov *et al.*, 2016).

2.3.1 Methodologies

Table 2.1 summarises research in code-related tasks using RNNs. It gives an overview of the methods used by other researchers.

Past research looked at two main approaches to code prediction: Character-based RNNs (Priya *et al.*, 2017), where the model predicts the next character from a probability distribution. Word-based models, in contrast, have full word tokens and predict the next complete word. Results show that word-based RNNs achieve a higher semantic accuracy (Burgueño *et al.*, 2021) than character-based RNNs (Priya *et al.*, 2017). Supported by these findings, this research implements word-level RNNs in order to attempt to generate more error-free code. Specifically, our model aims to predict program output from a given input without directly executing the code.

2.3.2 Evaluation Metrics

Researchers commonly use accuracy to evaluate their models (Burgueño *et al.*, 2021; Priya *et al.*, 2017; Zaremba and Sutskever, 2014; Zaremba, Mikolov *et al.*, 2016). Accuracy tells us the fraction of correct examples but lacks any notion of semantic correctness; this can be an issue for code generation because there are often multiple correct solutions. Better methods involve using Bilingual Evaluation Understudy

Table 2.1: Summary of research using RNNs in Code-Related Tasks, outlining methods, the metrics used and the results achieved

Author	Dataset	Method	Metric	Results
Graves. et al. (2014)	A mix of data of simple algorithms.	RNN with external memory, supervised learning, 3 layer.	Bits-per-sequence	Able to generate code, dataset size had no impact.
Wojciech and Sutskever (2015)	Set of custom made, Python like Programs.	LSTM, and curriculum learning.	Accuracy	99% accuracy for addition, LSTM learned programs.
Reed. et al. (2015)	A set of programs custom made.	NPI (Neural Programmer Interpreter) - an LSTM based network with arguments and Supervised Learning. Also used Curriculum Learning.	Ability to generalise and per-sequence accuracy	Was able to learn to execute complex programs, <90% accuracy in most cases.
Wojciech. et al. (2016)	Short python programs.	RNN controller: LSTM and GRU, Reinforcement Learning	Accuracy.	Q-Learning failed on most tasks, but succeeded with Watkins and Dynamic Count.
Balog. et al. (2017)	Custom made DSL.	Encoder Feed Forward, Decoder RNN.	Time to find a program.	DeepCoder showed performance speed ups up to 475×
Priya R. et al. (2017)	Open Source from Github (Python, Java, C#).	Character-based RNN 512 hidden state.	Loss and Accuracy.	Performed well, sometimes generated code with errors, RNN were effective.
Rahman. et al. (2020)	Code from Oj competitive programming system.	LSTM and Attention.	Precision, Recall, F1.	Attention improved accuracy from 31% to 62% and approx. 0.9 F1
Laich L. et al. (2020)	Rico dataset of PlayStore apps.	MLP, CNN and RNN.	Existing Research and Accuracy	Improved from 35% to 71% accuracy.
Burgueño. et al. (2021)	Custom Dataset using GANs (Generative Adversarial Networks).	LSTM + Attention, with Dropout and Regularisation.	Accuracy and training time.	0.94 accuracy after increasing dataset size, this was also proportional to train time.

(BLEU) to measure the token-level similarity (Burgueño *et al.*, 2021), providing additional insight into the performance.

Researchers used both execution-based and functional correctness metrics. Balog *et al.* (2017) computed the time taken to find a program – however, most struggle to evaluate for functional correctness. For instance, Laich, Bielik and Vechev (2020) used human feedback to find how many corrections a user would need to make for the program to be correct.

This research aims to use a balance of metrics to give an overview of how well the model performs. This will include Loss, Perplexity and Sentence Similarity.

2.3.3 Limitations

Current research has a few limitations; one limitation is that studies reported difficulty producing compilable and error-free code (Priya *et al.*, 2017). They found that this did not improve with more data. This is important for tools that aid with code because generating error-free code is vital, so as not to hinder the user. One way this research addresses this is Syntax-aware training, which involves using annotated embeddings or semantic embeddings to allow the model to associate semantic concepts with the output.

Other researchers had issues with the dataset size and quality. Burgueño *et al.* (2021) found that the dataset bottlenecks the model. They proposed using GANs (Generative Adversarial Networks) to generate Synthetic data. However, Balog *et al.* (2017) found that synthesised programs may sometimes be too simplistic and fail to address real-world applications. Other researchers had issues with the dataset size and quality (Burgueño *et al.*, 2021; Rahman, Watanobe and Keita Nakamura, 2020; Reed and Freitas, 2015). Variety in the dataset is important for real-world problems because it increases the scope in which they can be applied, thus benefiting more people. This research aims to use Large Language Models (LLMs) for Data Augmentation to generate synthetic data, which will be used to train the model, using

prompting techniques to generate a larger range of data.

Another way current models are limited is that they fail to understand the Semantic Meaning of code, which affects any error detection or prediction tasks (Rahman, Watanobe and Keita Nakamura, 2020). To address semantic understanding, this research aims to use Semantic Embeddings to encode the program's meaning or its AST (Abstract Syntax Tree) into the training data. This will allow the model to understand the meaning of the code and not just the syntax.

RNNs also tend to fail to generalise for longer inputs (Reed and Freitas, 2015) or follow a program specification too closely (Laich, Bielik and Vechev, 2020). This can also lead to the model memorising the dataset rather than learning the underlying patterns (Zaremba and Sutskever, 2014). This research aims to address this by using Curriculum Learning, which involves training the model on simpler tasks first and gradually increasing the complexity of the tasks. Curriculum learning has been shown to improve generalisation and learning speed (Reed and Freitas, 2015, Zaremba and Sutskever, 2014).

2.4 Transformer Models

The transformer architecture is an improvement upon sequencing models like RNNs. RNN-based models have been shown to struggle with long-distance dependencies and are inherently difficult to parallelise. Its architecture consists of an encoder and a decoder block, where the encoder processes the input sequence, and the decoder generates the output sequence. The transformer model uses attention mechanisms to learn the dependencies between words in a sentence. This allows the model to learn long-range dependencies more effectively than RNNs.

Additionally, the transformer model is highly parallelisable because the model does not process the input in sequence but all at once.

Transformer models have recently been used for code generation, completion, and

Table 2.2: Summary of Transformer Models in Code-Related Tasks

Author	Dataset	Method	Metric	Results
Bunel. et al (2016)	Generated I/O Examples for Swap Increment Addition and Sort	Neural Network ANC	correctness, halting, efficiency, and confidence, weighted sum to form the total loss function	adapts to simple generic algorithms
Chen. et al. (2021)	Python GitHub Repositories	Temperature, Transformer	pass@k - number of samples that pass per problem, Human Eval.	improved pass rate from 28.8% to 70.2% on the HumanEval dataset.
D. O. Bui. et al. (2023)	HumanEval, CodeXGLUE, Human Input	GPT Transformer	pass@k CodeBLEU	a library which makes it easy to manipulate code models.
Cummins. et al. (2023)	A suite of benchmark datasets of C/C++ code examples	Transformer BPE	BLEU and Success Rate.	90.5% success rate 0.952 BLEU
Munley. et al. (2023)	OpenACC and their own Dataset	GPT, DeepSeek, CodeLlama + RAG	GPU Time Taken, and Pass/Fail Rates.	Tested various models, most passed the majority of tasks.
Gu. Qiuhan (2023)	Processing Go Code using Tree Sitter	CodeT5, Transformer	Code coverage and the percentage of syntax error and undefined behaviour	3.38% coverage, 2.79% syntax errors, 0% of undefined behaviour.
Kim. et al. (2023)	HotpotQA, Movie Recommendation, ParallelQA and other benchmark tasks.	Transformers, GPT	Accuracy and Latency, Success Rate	latency speed up 3.7x, cost savings of up to 6.7x, and accuracy improvement of up to ∼9%
Li, L. et al. (2024)	MAmmoTH model, 18,000 examples from GSM8K, 3,000 from NumGLUE, and 15,000 from MATH.	Attention, CoT, PoT, Reinforcement Learning, Transformer, Llama and GPT	Base-Models and out-of-domain datasets (one that differs a lot from the training data).	Improvement of 6.5% on Llama-Base model, 4.3% on MistralBase model across 8 mathematical calculation datasets.
Grubisic. et al. (2024)	LLVM IR (Intermediate Representation).	Llama - Transformer	Track % improvement over Random Sampling, Greedy Decoding, and Nucleus	2.87% to 5% improvement.
Hu. et al. (2024)	Sintel movie dataset and Custom Made dataset.	Multimodal Learning, GPT-V (GPT Vision).	CLIP Similarity Score (Text-to-Image metric), Human Evaluation	5.6 CLIP on BlenderGPT baseline, and 88.9 on Scene Craft
Chae. et al. (2024)	Big-Bench Hard Reasoning Tasks.	GPT3 and CodeLlama, Zero-Shot CoT.	Improvement over Baseline Models	11% improvement over GPT3.5-Turbo
Cummins. et al. (2024)	CodeLlama LLVM IR. MiBench Benchmark	CodeLlama as a Base.	Vary for tasks: Perplexity, BLEU, Success Rate, pass@k	Outperforms in 61% of cases, significant improvement over baseline models.

summarisation tasks. These are summarised in Table 2.2. Most authors fine-tune pre-trained models like GPT3 for specific tasks or use existing models as a base. This was effective in most cases, where most research showed a significant improvement compared to baseline models. Comparing against baseline models allowed past research to put results in context with other current models.

2.4.1 Limitations

Transformer-based approaches tended to suffer from similar limitations to RNN-based approaches. Researchers found that pre-trained models are biased towards specific languages or frameworks when trained on various languages (Bui *et al.*, 2023; Chen *et al.*, 2021). This research only uses Lua examples and thus should not suffer from this level of bias. However, it should be noted that the model may favour certain implementations over others; to mitigate this, the model will be trained on a diverse dataset consisting of many different ways to achieve the same result.

Additionally, models struggled with processing large input sequences (Chris Cummins, Grubisic, Elhoushi *et al.*, 2023) and solved it using a restricted input; however, as proposed in section 2.2.3, curriculum learning is a viable approach to train the model on increasing input sizes. As well as input size, the models produced are algorithmically complex and sequential (Grubisic *et al.*, 2024), making it impossible to train in parallel.

The use of MinLSTM and MinGRU will allow for longer sequences and faster training due to their simplified and, therefore, parallelisable nature (Feng, Tung, Ahmed *et al.*, 2024).

MinLSTM and MinGRU architectures have been shown to outperform most RNN architectures, and reach performance comparable to that of large transformer models Feng, Tung, Ahmed *et al.* (2024). These new architectures have been tested in a minimal range of fields, and their current application spans foundation modelling (Sieber *et al.*, 2024), language model optimisation (De *et al.*, 2024; Akyürek *et al.*, 2024; Afzal *et al.*, 2025), which took advantage of the model's parallelisability. Additionally, their use in State Space Modelling (Liu and Li, 2024; Parnichkun *et al.*, 2024)

involves handling dynamic inputs such as time-series analysis, and physics-informed machine learning (Hu *et al.*, 2024). This study aims to explore their application in code generation.

Chapter 3

Requirements Analysis

This chapter will highlight the study's requirements, which are split into functional and non-functional requirements. Functional requirements are the core features the software needs to meet, and non-functional requirements are the properties required for the system to run or to enhance it.

3.1 Functional Requirements

Dataset: A dataset of Lua code snippets is required. This then needs to be processed by running through a Lua compiler, ensuring there are no errors in the code and that we can generate an output. The programs must be error-free for this study to effectively evaluate the model's ability to generate output, rather than detect errors.

Preprocessor: Code snippets needs to be parsed and processed, therefore part of the software is needed to parse Lua programs into tokens, from which program embeddings are generated. This is necessary because the process cleans the data and removes any syntactic noise, for examples brackets and other punctuation that add no semantic meaning, from the program.

Data Splitting: Data must be split into 3 sets - the training, testing and validation sets. In Machine Learning, the training set allows the model to learn patterns in the data, and the testing set is used after training to evaluate how well the model performs on unseen data. Lastly, the validation set is used during hyperparameter tuning to provide an unbiased set of data to validate parameters. This is often done

in an 70/15/15 split, meaning 70% of the data is used for training, and the test and validation sets become 15% of the data.

Models: Once the data is split, it needs to be passed into a model. This study aims to implement and train an RNN, LSTM, GRU, MinGRU and MinLSTM model along with utility layers such as Mixture of Experts. A wide range of RNN models gives a better overview of how well RNNs can emulate Lua compilation.

Language Model: A universal language model training pipeline, which allows for swapping of the underlying architecture, is vital for testing each model fairly with no bias towards certain parameters and training pipelines.

Hyperparamter Tuning: Once all the models are implemented, it is required that the correct parameters are used to achieve the best performance. The implementation must support tuning of the learning rate, number of input/ hidden layers, and the number of iterations to train the model for.

Evaluation: After training, the model must be evaluated to gauge its performance on unseen data. This should be done using a variety of metrics such as F1 Score, Pass@k and Sentence Similarity. Using a range of metrics indicates how well different aspects of the model are performing - perplexity indicates how sure the model is of its prediction, whereas pass@k notions towards the percentage of passing test cases.

Baselines: Finally, this study will compare how well the model is doing against other baselines, for instance, existing Large Language models. As a result, this study is put into perspective in terms of current state-of-the-art models.

3.2 Non-Functional Requirements

Data Visualisation: This research will use a range of metrics along with Loss Plots to present the findings and compare against other models or architectures. A table is also needed to present the configurations and performance of each model.

3.2.1 Hardware Considerations

The training of the models will be run locally on a single *NVIDIA GeForce RTX 3060 GPU*, with *12.11GB VRAM* and *3584* CUDA Cores. The experiments will run on a desktop with the following specifications:

- Operating System: Windows 11
- CPU: Intel core i5-12400F (6 cores, 12 threads)
- RAM: 16GB DDR4

3.2.2 Software Requirements

A range of software is required for this study:

- Visual Studio Code: a code editor.
- PyTorch v2.5.0 with CUDA: a Python Deep Learning Library.
- Optuna: hyperparameter tuning library.
- Matplotlib: Plotting data on charts for visualisation.
- Python3: The Python Programming Language.
- Browser: To interface with ChatGPT for data augumentation.
- Treesitter: A Parsing library along with the Lua Grammar DLL.

Chapter 4

Design & Methodology

4.1 Research Philosophy

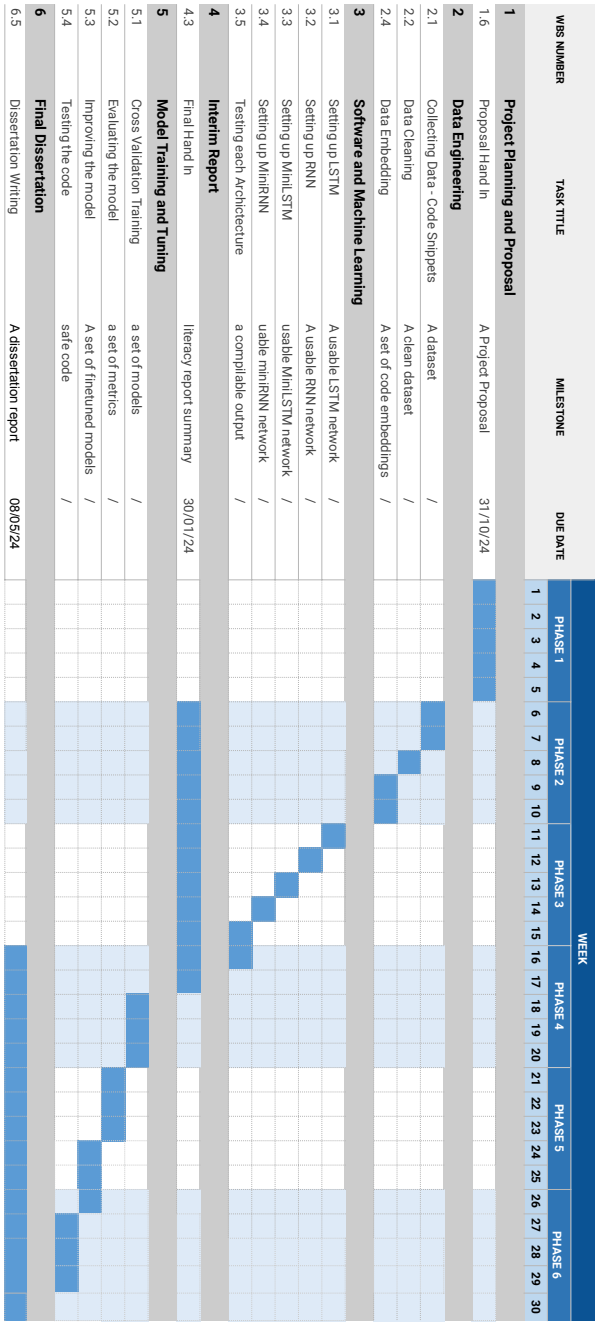
This chapter introduces the methods used to test whether RNNs can emulate compilation. First, looking at how the system will be designed, followed by a justification of the model selection. Finally, the evaluation strategies and ethics will be discussed.

Lua emulation will be evaluated by testing a range of models on the same input-output dataset to determine how well each model performs on the data, highlighting the differences in architecture rather than any other factors like training time and data difficulty.

4.2 Project Management

The software design will follow the iterative model. This model involves starting with a small subset of the requirements and then integrating them until all the requirements are met.

In this Software Development Lifecycle (SLDC), the project starts with an incomplete software specification and implements it. The requirements are then iterated upon until the software is complete. This research will first implement a basic character-based RNN to complete the word "hello". Once that requirement is met, it is iterated to predict full words and sentences. Figure 4.1 outlines each iteration in terms of milestones with the date they are planned to be met.



WBS NUMBER	TASK TITLE	MILESTONE	DUE DATE
1	**Project Planning and Proposal**		
1.6	Proposal Hand In	A Project Proposal	31/10/24
2	**Data Engineering**		
2.1	Collecting Data - Code Snippets	A dataset	/
2.2	Data Cleaning	A clean dataset	/
2.4	Data Embedding	A set of code embeddings	/
3	**Software and Machine Learning**		
3.1	Setting up LSTM	A usable LSTM network	/
3.2	Setting up RNN	A usable RNN network	/
3.3	Setting up MiniLSTM	usable MiniLSTM network	/
3.4	Setting up MiniRNN	uable miniRNN network	/
3.5	Testing each Archictecture	a compilable output	/
4	**Interim Report**		
4.3	Final Hand In	literacy report summary	30/01/24
5	**Model Training and Tuning**		
5.1	Cross Validation Training	a set of models	/
5.2	Evaluating the model	a set of metrics	/
5.3	Improving the model	A set of finetuned models	/
5.4	Testing the code	safe code	/
6	**Final Dissertation**		
6.5	Dissertation Writing	A dissertation report	08/05/24

Figure 4.1: Gantt Chart Outlining the time for each iteration and the due dates for each milestone.

4.3 System Design

4.3.1 System Architecture

The training and testing pipeline will be built using Python, *PyTorch v2.5.0*, and *CUDA v12.6*. The pipeline will consist of several components; these can be seen in 4.2. The system will consist of a set of model architectures, passed to a high-level *LanguageModel*, where they will be trained on data from the *CodeDataset*. Finally, the outputs will be passed into an Evaluator, which will compute a set of metrics to gauge the performance of each model.

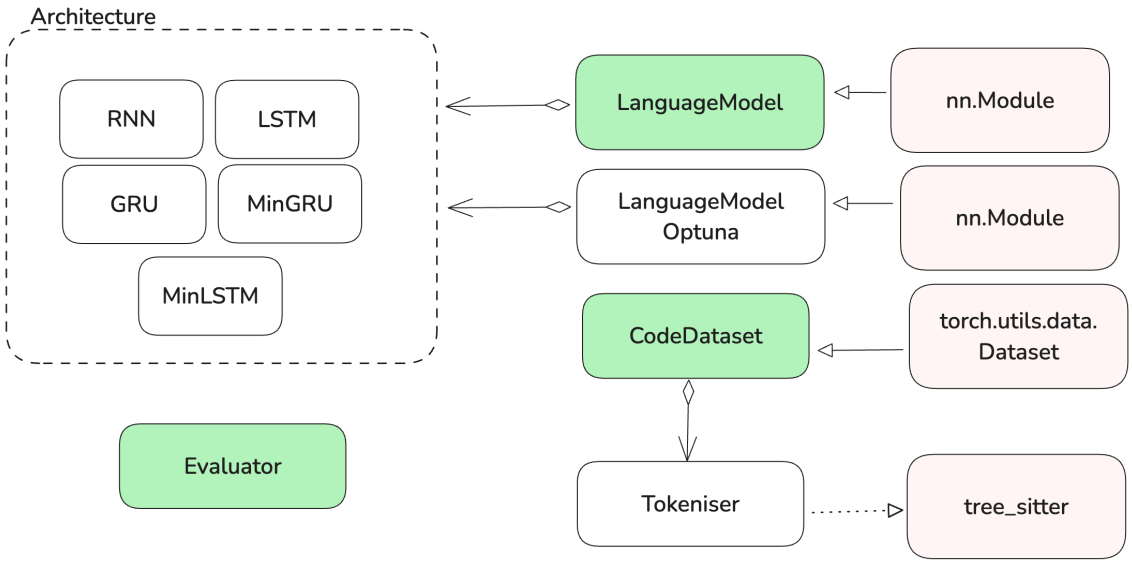


Figure 4.2: High-Level Class Diagram of the Implementation, of single unit architectures for testing, and the class that rely on them. Also highlighting the libraries each class depends on.

Entities marked in green are the main components that run in the programme. The software will be structured like this for easy running of experiments because the *LanguageModel* will be a universal component that can hot-swap and train different architectures.

4.4 Model Selection

This research will train five different *RNN-based* architectures: *standard RNN*, *LSTM*, *GRU*, *MinGRU*, and *MinLSTM*. These were picked to test how different architectures perform.

4.4.1 RNN

This research will use an RNN as a baseline to compare how architectural changes affect the emulation of Lua. RNN models are the simplest recurrent neural architecture and have shown varying levels of success. Priya *et al.* (2017) used a character-level RNN for code generation, which was error-prone; more success was achieved using word-level RNN models (Graves *et al.*, 2014; Zaremba, Mikolov *et al.*, 2016), where a "word" is a programming language token such as "IF".

The advantages of an RNN approach include its low computational requirements compared to larger model architectures, simplicity of implementation, and the ability to handle variable-length sequences like programs. Additionally, RNNs maintain an internal state, which could allow them to capture longer-term patterns and dependencies in code, making them suitable for Lua emulation.

However, RNNs can suffer from vanishing gradients, making them inefficient for modelling programs, since they may lose information about certain semantic elements over time.

4.4.2 LSTM

LSTM networks are commonly used to model time-dependent data (Zhao *et al.*, 2023) because they can learn context over time and refer back to learned contexts. As previously discussed, LSTM networks can be considered a forgetting and remembering program that stores data over time, or as the program continues. LSTM models have shown success in a variety of studies, such as program completion (Rahman, Watanobe and Keita Nakamura, 2020), executing small programs (Zaremba and Sutskever, 2014), and executing more complex programs (Reed and Freitas,

2015). This study will use an LSTM to model how a compiler can refer to a previous scope or variable during compilation, and help model the emulation of Lua.

In addition to the gating mechanism, LSTM networks solve the issue of vanishing gradients (Noh, 2021), making them ideal for learning input-output sequences. They can, however, take a longer time to train than an RNN and may be more computationally intensive due to the complexity of gate computations.

4.4.3 GRU

A *GRU* will also be trained to compare how other architectures perform, such that a better-performing LSTM would indicate the LSTM has an advantage over other architectures. The GRU has seen success in learning simple algorithms, only falling short at more complex ones (Zaremba, Mikolov *et al.*, 2016). The cited study looked at how these models perform in addition to a reinforcement learning technique known as Q-Learning. Zaremba, Mikolov *et al.* (2016) showed that GRUs could maintain the performance of the LSTM up until more complex examples; this makes the GRU an ideal model to compare with the RNN and LSTM in emulating Lua.

The advantages of GRUs are similar to those of RNNs; they are computationally more efficient than LSTM models because they require fewer parameters while maintaining comparable performance. This results in faster training times and reduced memory requirements. Compared to LSTMs, they are also simpler to implement as they require fewer gates.

However, GRUs can also present some limitations; their simplified gating mechanism may restrict their capacity to remember information over very long sequences, thus limiting the effectiveness in modelling complex programs. Although GRUs has a simpler architecture than LSTM, these models may be too simple to learn complex programming language patterns.

4.4.4 MinLSTM and MinGRU

Additionally, two more recent architectures will be trained: the MinLSTM and MinGRU (Feng, Tung, Ahmed *et al.*, 2024); these simpler RNN models can be trained in

parallel and perform as well as a transformer model. Studies involving the MinLSTM and MinGRU have succeeded in language modelling, but remain untested for code-related tasks.

4.4.5 Mixture of Experts

MoE has significantly improved the inference of LLMs such as Deepseek R1 (DeepSeek-AI, 2025). This research will implement a similar notion of experts to determine whether such specialised neurons improve the code understanding of the model. A MoE approach has shown success in speech recognition using LSTMs, outperforming other models without using experts (Chazan, Goldberger and Gannot, 2017).

MoE in emulation could benefit the model in learning separate parts of Lua syntax and being able to route between the correct syntactic features. Conversely, introducing too many experts could increase the model's sparsity and cause the model to overfit the data.

4.4.6 Attention

Attention is used in Large Language Models, to decide which tokens the model should put more importance on. Vaswani *et al.* (2017) introduced attention for the transformer architecture, where it was termed scaled dot product multi-head attention. Equation 4.1 computes self-attention from queries (Q), keys (K), and values (V) - a set of scores representing the similarity of each token with all other tokens.

$$Attention(Q, K, V) = SoftMax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.1}$$

Feng, Tung, Hajimirsadeghi *et al.* (2024) has shown that attention is useful in RNNs to allow them to learn more complex patterns by letting the network focus on specific parts of the input. So, this study will train models using self-attention to improve the inference of each RNN architecture.

4.4.7 Hyperparameter Tuning

Models will be tuned using Optuna to find a configuration which reaches the most optimal minimum. The objective function optimises the learning rate, the number of iterations, number of hidden layers, and the number of input layers.

4.5 Dataset

Programs used to train the models will be written in the Lua programming language. Lua was chosen because it is a very small language built to be embedded into any application (Ierusalimschy and Figueiredo, 2009). Simpler programs reduce the amount of syntactic noise the compiler, or in this case, reducing the amount the model, needs to learn, allowing it to focus on semantics over syntax.

The dataset of Lua code snippets will be synthetically created using programmatic and Generative model-based data augmentation. Generative models such as ChatGPT from Openai and Claude from Anthropic often hallucinate code outputs; all code will be passed through a Lua compiler to ensure it correctly compiles and gives an output.

4.5.1 Curriculum Learning

Curriculum learning is a technique in which models are trained on data presented in a meaningful order of increasing difficulty, similar to how our education system is structured. Bengio *et al.* (2009) first formalised this technique and proved that curriculum learning has a benefit in vision and language tasks. Reed and Freitas (2015) and Zaremba, Mikolov *et al.* (2016) were able to show that curriculum learning improved model accuracy, bringing it in the 90%+ range.

This research will use a curriculum of Lua programs in three categories: easy programs - those with minimal branching (1-2); medium programs - those with more complex branching (3-4) and structures; and hard programs - ones with multiple branches (5+) and state changes.

4.5.2 Lua Semantics

Current research fails to train models that understand the semantic meaning of code (Rahman, Watanobe and Keita Nakamura, 2020). This research will use S-expressions generated from the Lua program's Abstract Syntax Tree (AST) representation. The *tree-sitter* library will generate the AST and S-Expression.

S-expressions, or symbolic expressions (often abbreviated sexp, sexpr), are a way to represent "tree-like" structures in an expression format. They were first used in the Lisp programming language to represent source code.

To illustrate, a simple print statement, presented in Figure 4.3, is parsed into the S expression in Figure 4.4.

```
print("Hello Sheep!")
```

Figure 4.3: Simple Lua program to print a message.

```
(chunk
    ;call print() with arguments
    (function_call name: (identifier)
        arguments: (
            arguments (
                string content: (string_content)
            )
        )
    )
)
```

Figure 4.4: The equivalent S-Expression for the print statement in 4.3.

S-Expressions help the model learn Lua semantics because user-specific names are stripped out into an 'identifier' token, reducing the noise in the input. This should help the model understand that a $print(\text{"}hello\text{"})$ and $func(\text{"}world\text{"})$ are semantically both function calls. The S-Expression will be further parsed to generate *semantic tokens* such as "STRING(hello)" to provide a more meaningful, but less noisy output.

4.6 Benchmarking

4.6.1 Evaluation Metrics

This research will use metrics representing different aspects of the model for Lua program outputs. To determine how well overall the model is doing across all test cases, *Pass@k* (Chen *et al.*, 2021) - shown in equation 4.2 - will be used to measure the number of correctly predicted outputs.

$$pass@k := \mathbb{E}_{problems}\left[1 - \frac{C(n-c,k)}{C(n,k)}\right] \tag{4.2}$$

Where $C(n-c,k)$ computes the number of ways to choose k incorrect samples$(n-k)$, it reflects a scenario where all k samples are incorrect. $\frac{C(n-c,k)}{C(n,k)}$ gives the probability that all samples are incorrect.

Individual examples will be evaluated using Sentence Similarity, which is implemented as a cosine similarity. This will allow the evaluator to measure how close the model got to the correct output. The computation is shown in equation 4.3, involving a dot product of the two vectors divided by the product of their magnitudes.

$$\cos(\theta) = \frac{A \cdot B}{||A||\ ||B||} \tag{4.3}$$

Perplexity measures how well the model predicts a sample and captures the uncertainty level in its output. It gives an overview of how well the model thinks it is doing and thus shows whether it is learning the correct patterns. Perplexity is computed as the inverse probability of the corpus according to the Language model, normalised over the number of words ($\frac{1}{T}$); the calculation is shown in equation 4.4.

$$Perp = \prod_{t=1}^{T}\left(\frac{1}{P_{LM}(x^{(t+1)}|x^{(t)},\ldots,x^{(1)})}\right)^{\frac{1}{T}} \tag{4.4}$$

Cross-entropy will be used to train the network to optimise the loss function. Cross-entropy measures the difference between probability distributions of a given variable or set of events. The computation is shown in equation 4.5 - P is the true distribution of the data, and Q is the predicted distribution - it directly measures the distribution of the generated output against the correct probability distribution.

$$H(P, Q) = -\sum_{x} P(x) \log Q(x) \tag{4.5}$$

Predicting code output can also be considered a question-answering (QA) system, where the user asks a question in terms of a Lua program and wants an answer, which consists of the program output. As such, it can be evaluated in terms of a QA system through computing an F1 score (shown in Equation 4.8), which gives information about the token gap between the expected and predicted outputs.

$$Precision = \frac{\text{Number of Words Predicted + GT Words}}{\text{Total Words Predicted}} \tag{4.6}$$

$$Recall = \frac{\text{Number of Words Predicted + GT Words}}{\text{Total Words in GT}} \tag{4.7}$$

$$F1 = 2 \cdot \frac{Precision \times Recall}{Precision + Recall} \tag{4.8}$$

Finally, an Exact Match (EM) will be computed to give an overview of perfect predictions; the metric allocates a value of 1 to an output that exactly matches the expected output and a 0 for outputs that do not match. This is shown mathematically in Figure 4.9. EM gives an overview of how many examples were predicted correctly compared to those not.

$$EM = \begin{cases} 1, & \text{if } pred = expected \\ 0, & \text{otherwise} \end{cases} \tag{4.9}$$

Using various methods gives an overview of how well the models perform in different dimensions - $Pass@K$ and Cross-entropy Loss evaluate the system across the whole dataset. EM, Perplexity, and Sentence Similarity evaluate the performance per sample.

4.6.2 Baseline Models

As a baseline, this research will compare all trained models against the standard RNN to determine how the performance compares against the LSTM, GRU and Minimal versions—additionally, using LLMs specifically, StarCoder 2, and Qwen 2.5 Coder to generate outputs, which will be passed into a similar evaluation component in order to compute relevant comparative metrics.

4.7 Overview of Experiments

Table 4.1 provides an overview of different model configurations that this study will test. The first row shows the base configuration, which remains unchanged unless modified on a specific row. This research varies the architecture and number of experts to determine whether architecture has an effect on the ability of the RNN architecture to emulate Lua programs, and compute the set of metrics described previously.

4.8 Ethical Considerations

Large Language Models such as ChatGPT are trained using RLHF (Reinforcement Learning with Human Feedback) and are computationally expensive. It should be considered that using such tools for generating data and data augmentation comes with a large environmental cost.

Table 4.1: Architectural Variations on models tested. Values were updated from the base model. Hidden state size (h), number of experts ($expts$), and training steps. The model was evaluated on a Exact Match (EM), F1 Score ($F1$), Sentence Similarity (SS), Pass@1 ($P@1$), Perplexity ($Perp$), Recall (Rcl) and Precision (Prs).

	h	expts	train	EM	F1	SS	P@1	Perp	Rcl	Prs
base	900	0	3000							
RNN		2 4 10								
GRU		2 4 10		Table serves as an overview of experiments Full Results Presented and Explained in Chapter 6						
LSTM		2 4 10								
Mini GRU		2 4 10								
Mini LSTM		2 4 10								

The carbon footprint associated with running these models is exponentially large. Strubell, Ganesh and McCallum (2019) suggests that training a large transformer could emit just as much CO_2 as five cars throughout their entire usage period. Additionally, it is estimated that thousands of gallons of water are used to cool data centres when training large models (Luccioni, Viguier and Ligozat, 2022).

4.9 Risk Management

This research does not involve any human participants. Therefore, all risks involved are specific to the software implementation. One such risk involves the models failing to learn with a feasible amount of data.

Chapter 5

Implementation

5.1 Development Environment and Tools

The artefact was developed using Visual Studio Code and a Python virtual environment to track dependencies and isolate it from the system environment, doing so provides unified versions for each package, and avoids version conflicts.

5.2 Language Parsing

5.2.1 Building Data

The first requirement of the system is to read a dataset and be able to generate the outputs of each snippet of code. The dataset was generated by prompting (the prompt is illustrated in Figure 5.3) ChatGPT, and is available online [1]. This was done using the Python *Threading* and *concurrent.futures* library to read each Lua file in parallel and save the program, with its output in a separate *.luax* file which stores the Lua examples. A training example input is separated from the output using a *<PROGRAM END>* token, an example of a single simple program is shown in Figure 5.4, along with a more complex example in Figure 5.5. Since all Lua files were independent, it is trivial to process each path in a directory on separate threads using a *ThreadPoolExecutor* [2].

[1] https://github.com/MeRichard123/ml-compiler/tree/main/training_examples
[2] https://github.com/MeRichard123/ml-compiler/blob/main/src/Utils/build_data.py

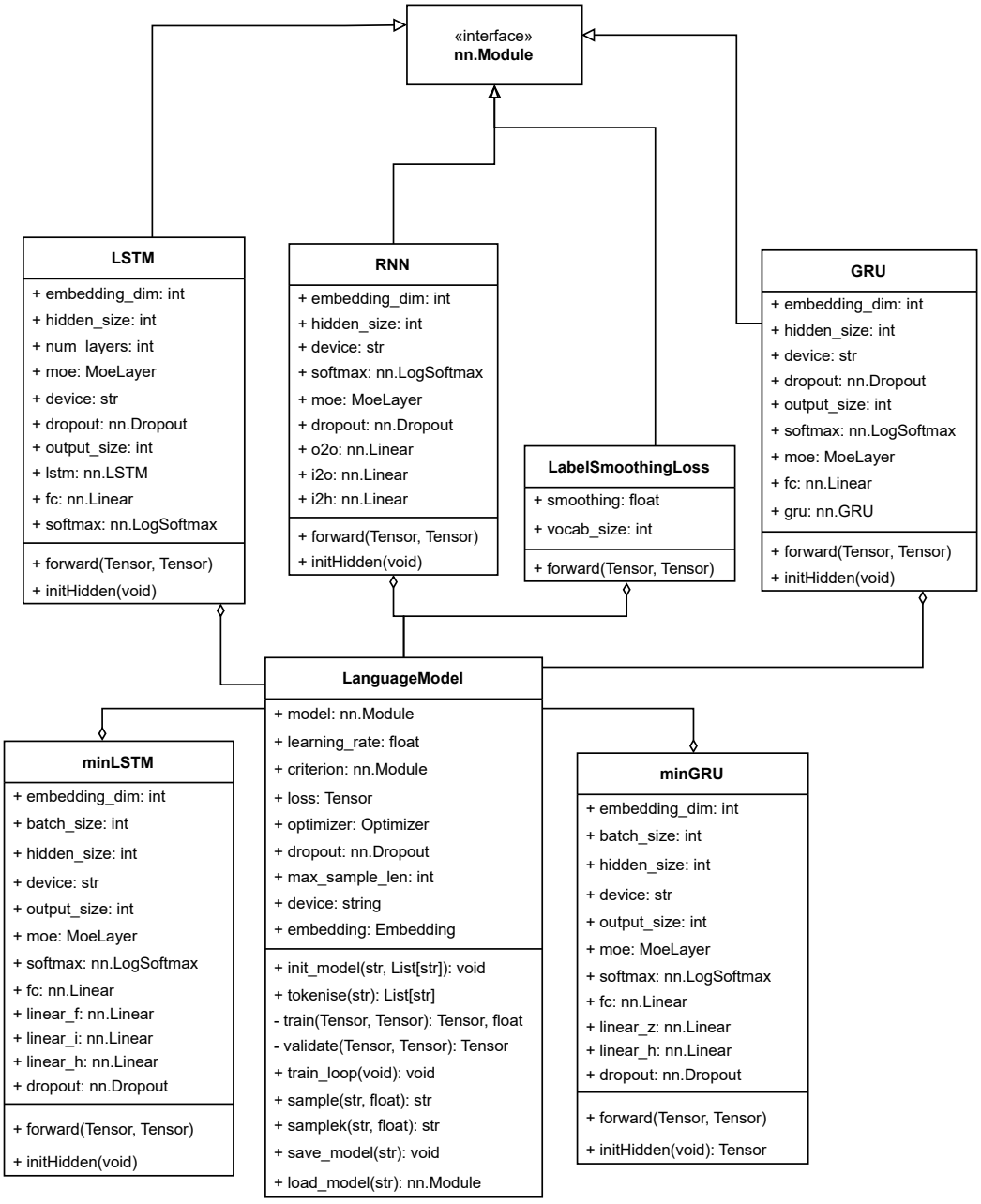


Figure 5.1: UML of Model Architectures and Parent Language Model, modelling the interactions between different classes.

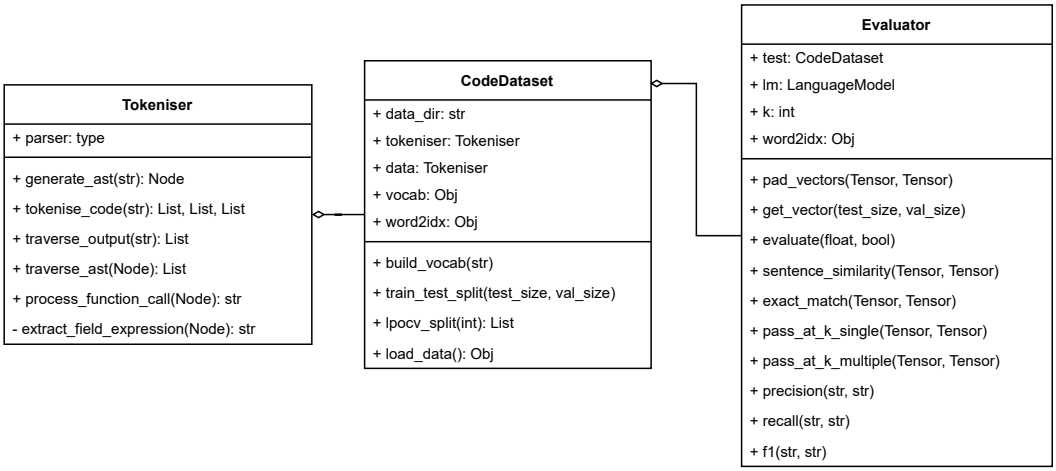


Figure 5.2: UML of the Tokenisation, Data and Evaluation part of the pipeline.

```
"Create 100 simple examples of Lua which only use
<language construct> statements, and demonstrate the
semantics of <language feature> in Lua, ensuring only a
single output and task per sample."
```

Figure 5.3: Prompt template used to generate Lua code examples, where <> indicate semantic features such as selection and looping.

```
message = "Hello World"
print(string.upper(message))
<PROGRAM END>
HELLO WORLD
```

Figure 5.4: Simple Program input-output example, of a string function running inside a print statement.

```
function getMinMax(numbers)
    local min = numbers[1]
    local max = numbers[1]

    for i = 2, #numbers do
        if numbers[i] < min then
            min = numbers[i]
        end
        if numbers[i] > max then
            max = numbers[i]
        end
    end

    return min, max
end

local values = {7, 2, 9, 4, 5}
local minimum, maximum = getMinMax(values)

print("In the given set, the minimum is "  \
        .. minimum .. " and the maximum is " \
        .. maximum)
<PROGRAM END>
In the given set, the minimum is 2 and the maximum is 9
```

Figure 5.5: Complex Program input-output example, to find the smallest and highest numbers of an array.

5.2.2 Semantic Tokenisation

Language models work with tokens created from a corpus of text using a tokeniser. This study uses *tree_sitter* to parse the AST, and generate a clean set of tokens free of syntax sugar. Treesitter first parses the code to generate an AST and then appends prompt-specific tokens such as *<PROGRAM END>*[3]. AST-specific tokens get converted into cleaner *Semantic Tokens* consisting of the token type and its data. The tokeniser is shown in Figure 5.2, consisting of AST generation and traversal, along with tokenisers for the input and output sections of the prompt. Tokenised output examples for the simple and complex Lua programs are illustrated in Figures 5.6 and 5.7

```
IDENTIFIER(message) EQUALS STRING(Hello)
    STRING(World)
FUNCTION_CALL(print) FUNCTION_CALL(string.upper)
ARGUMENTS(IDENTIFIER(str))
<PROGRAM END>
STRING(Hello) STRING(World)
```

Figure 5.6: Semantic Tokens for Figure 5.4

5.3 Dataset

The dataset was implemented using the PyTorch *IterableDataset* class; this was used over *Dataset* because it allows for a larger amount of data to be efficiently iterated by creating a Python Generator object. The class (visualised in Figure 5.2) encapsulates data processing into a *PyTorch* compatible dataset, which provides the added benefits of batch-processing for more efficient data manipulation. Batch processing works by splitting data into groups and handling them as larger groups rather than passing individual samples through the model.

Additionally, the implemented object handles switching between the next curriculum during training via curriculum learning. This is done via an instance variable, which updates the current dataset with the next curriculum data.

[3] https://github.com/MeRichard123/ml-compiler/blob/main/src/Parser.py

```
FUNCTION IDENTIFIER(getMinMax) IDENTIFIER(numbers)
    LOCAL IDENTIFIER(min) EQUALS
        IDENTIFIER(numbers) NUMBER(1)
    LOCAL IDENTIFIER(max) EQUALS
        IDENTIFIER(numbers) NUMBER(1)
    FOR IDENTIFIER(i) EQUALS NUMBER(2) COMMA
            IDENTIFIER(numbers) DO
        IF IDENTIFIER(numbers) IDENTIFIER(i) LESS_THAN
                IDENTIFIER(min) THEN
            IDENTIFIER(min) EQUALS IDENTIFIER(numbers)
                IDENTIFIER(i)
        END
        IF IDENTIFIER(numbers) IDENTIFIER(i)
                GREATER_THAN IDENTIFIER(max) THEN
            IDENTIFIER(max) EQUALS IDENTIFIER(numbers)
                IDENTIFIER(i)
        END
    END
    RETURN IDENTIFIER(min) COMMA IDENTIFIER(max)
END
LOCAL IDENTIFIER(values) EQUALS NUMBER(7) COMMA
    NUMBER(2) COMMA NUMBER(9) COMMA NUMBER(4)
    COMMA NUMBER(5)

LOCAL IDENTIFIER(minimum) COMMA IDENTIFIER(maximum)
    EQUALS FUNCTION_CALL(getMinMax) IDENTIFIER(values)

FUNCTION_CALL(print)
STRING(In the given set, the minimum is) COMMA
    STRING(and the maximum is)
<PROGRAM END>
STRING(In) STRING(the) STRING(given) STRING(set)
STRING(the) STRING(minimum) STRING(is) NUMBER(2)
STRING(and) STRING(the) STRING(maximum) STRING(is)
NUMBER(9)
```

Figure 5.7: Semantic Tokens for Figure 5.5

5.3.1 Data Splitting

In training Deep Learning models, it is common to split the data into three datasets. The training set, which consists of 70% of the data, is what the models use during training to learn patterns. The validation set, which consists of 15% of the data, is used during training to measure the level of overfitting. Validation gives an overview

of how well the models perform on unseen data as training progresses. The last 15% is also unseen and used for testing and is used to compute evaluation metrics, and shows how well the model performs on unseen data after training. The datasplits are visualised in the Pie Chart shown in Figure 5.8, where the total number of samples is 677. The function [4] returns three subsets of the dataset, which gives the class the ability to fetch a list of prompts to use during evaluation.

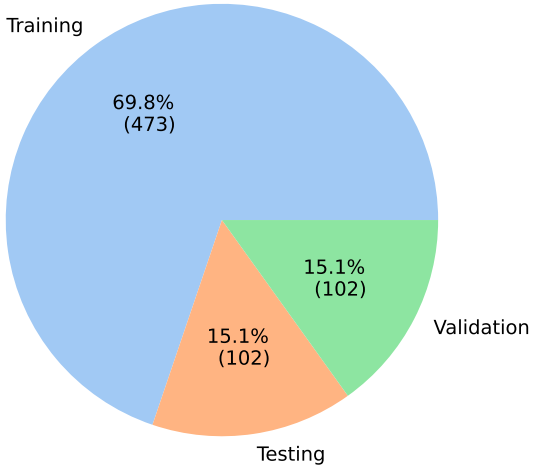


Figure 5.8: Chart visualising the data splits and the number of samples per split, using a 70-15-15 split and 677 data samples.

Additionally, this study implements Leave-p-out Cross-Validation [5]; which is a technique used for model evaluation that uses a sliding window of test and train samples. Such a method helps assess the performance and generalisation of a model because it leads to a broader range of data being used for training and testing, reducing the model's bias towards specific data samples. Another way this is achieved is by shuffling the data during splits, which randomises the samples in each data split.

[4] https://github.com/MeRichard123/ml-compiler/blob/main/src/Data.py#L153
[5] https://github.com/MeRichard123/ml-compiler/blob/main/src/Data.py#L185

5.3.2 Building the Vocabulary

The dataset generates the vocabulary (V), which the models use to compute probabilities and sample from. This is implemented via the *build_data* [6] method (Figure 5.2), and works by first tokenising each sample and then creating a two-way mapping between the token and the token index. The method returns both *idx2word* and *word2idx* mappings along with V and $|V|$.

5.4 Training Pipeline

Training and inference occur in the *LanguageModel*, a universal language model training container. It allows for hot swapping of the model architecture by passing in the model as an instance variable; therefore, keeping the same training process per model eliminates bias for specific models, which differ in training. A universal training container gives way to running experiments via swapping only the model architecture.

5.4.1 Language Model

The *LanguageModel* class can be seen with its parameters in Figure 5.1. The model is first initialised [7], which builds the vocabulary. This method also initialises the loss function by passing in the vocabulary size, needed for computing label smoothing confidence (see section 5.7).

Training

Model training begins in the *train_loop* function, which uses the *TQDM* library to display training progress and runs for 3000 epochs. Each epoch loops over data batches and splits the batch into input and output tokens to be passed through the *train* function, additionally iterating over the validation set and calling *validate* to compute the validation loss.

[6] https://github.com/MeRichard123/ml-compiler/blob/main/src/Data.py#L94
[7] https://github.com/MeRichard123/ml-compiler/blob/main/src/LanguageModel.py#L39

Training the model takes place in the *train* method, which converts the input into an embedding and passes it to the model to initialise the hidden state of the seq2seq model. The model is trained by computing the loss between the target and model output tensors. The same approach is used to calculate the validation loss on the dataset.

Sampling

The process of sampling from the model involves predicting a given input; the model makes an inference about some unseen data. This procedure [8] requires appropriate prompt formatting by appending the *<PROGRAM END>* token. Like training, the input is processed by passing it through the model, updating the hidden states. The core of model sampling is the generation loop, which uses a greedy decoding approach to sample the probability distribution and pick the top-most likely token. Greedy Decoding was selected over Top K and Nucleus Sampling because, unlike natural language, programming languages tend to lack output variance, and the model should only be concerned with the most likely next token.

Finally, the output is updated to the next token by updating hidden states based on the current output. Each output is concatenated on each generation step and returned as the final model generation output.

Check-pointing

An integral part of the testing pipeline was being able to save and load [9] the model states. This was useful because it permitted keeping trained models in checkpoints to be loaded for inference, saving the time needed to retrain them. Check-pointing was implemented via the save and load model methods, which fetch and update all the model properties to match the checkpoint.

[8] https://github.com/MeRichard123/ml-compiler/blob/main/src/LanguageModel.py#L281

[9] https://github.com/MeRichard123/ml-compiler/blob/main/src/LanguageModel.py#L400

5.4.2 Language Model Hyperparameter Tuning

Deep learning models often consist of many parameters which affect training, but are not direct parameters of the model. This study uses the *Optuna*, framework to tune and find the most optimal values of the model learning rate, number of iterations, and number of layers of the model. In *Optuna* this process is implemented [10] by first creating an objective function which runs the model training and defines the parameters to test, once that is in place, a tuning function is used to create an *Optuna Study* which repeatedly runs the objective with the set of trail parameter until the parameters which minimise the objective function the most are found.

5.5 Model Implementation

For consistency, all models were implemented with the same interface; using the PyTorch module base class, they implemented an initialisation method and some pseudo-methods for returning the same model for logging.

Each input vector is represented in three dimensions: (N, L, $H_{in/out}$), where N is the batch size, L is the sequence length and $H_{in/out}$ represents the input or output size. Various checks are implemented to ensure the dimensions match the expected dimensionality and that the correct dimensions are *squeezed* and *unsqueezed* where needed.

5.5.1 RNN

The standard RNN subclass is implemented using three linear neural networks [11] from the PyTorch *nn* class. The first converts the input into hidden states, then the hidden state gets converted to the output, and finally, the hidden state output gets converted to a network output.

Its forward method processes each part of the input sequence individually, combining

[10] https://github.com/MeRichard123/ml-compiler/blob/main/src/LanguageModelOptuna.py#L20

[11] https://github.com/MeRichard123/ml-compiler/blob/main/src/Architectures/RNN.py#L20

the hidden and input states to pass the vectors through the network layers. The output is then concatenated using *torch.stack* along the sequence dimension.

5.5.2 LSTM and GRU

LSTM and GRU models are implemented using a similar approach to the RNN; however, they use the *nn.LSTM* [12] and *nn.GRU* [13] layers rather than using recurrent linear layers.

Individual models are parameterised by first the embedding dimension, which specifies the dimensionality of the embedding vectors (equal to $|V|$), and the batch size, referring to the number of samples processed in one batch. The model is also characterised by the size of hidden and output states, which indicate the number of neural network layers.

All models are also optionally configured to accept the Mixture of Experts configuration and to toggle the attention mechanism, which trains the model with and without attention and MoE for experiment purposes.

5.5.3 MinGRU and MinLSTM

Minimal GRU [14] and LSTM [15] models are implemented directly as described in (Section 2.1.1), as three linear layers, the softplus function, and a range of utility functions defined in the *Utils* module.

The *Utils* module first implements the *log_g* function [16] for converting the input vectors into log space. Feng, Tung, Ahmed *et al.* (2024) showed that log space computation improved numerical stability compared to the standard version.

The parallel prefix scan is also implemented via *cumsum* - computing the partial sums

[12] https://github.com/MeRichard123/ml-compiler/blob/main/src/Architectures/LSTM.py#L18
[13] https://github.com/MeRichard123/ml-compiler/blob/main/src/Architectures/GRU.py#L25
[14] https://github.com/MeRichard123/ml-compiler/blob/main/src/Architectures/minGRU.py#L9
[15] https://github.com/MeRichard123/ml-compiler/blob/main/src/Architectures/minLSTM.py#L9
[16] https://github.com/MeRichard123/ml-compiler/blob/main/src/Utils/scan.py#L5

during up-sweep and *logcumsumexp* propagating the values down the tree [17], in the down-sweep step based on the Feng, Tung, Ahmed *et al.* (2024) implementation.

5.5.4 Mixture of Experts

Mixture of Experts (MoE) is implemented as a Utility layer [18] which runs optionally for each model when an appropriate configuration is provided. MoE is configured by providing the number of experts the network should use. This study implements dense MoE as described by Jacobs *et al.* (1991); in dense MoE all experts are activated for each input, conversely, sparse MoE activates only a subset of experts per input. Dense mixtures of experts reduce information loss caused by sparse experts; this benefits small models such as RNNs by removing the overhead of the sparse routing mechanism.

Adaptive Mixtures of Experts is implemented through a set of m linear layers, where m is the number of experts, alongside a gating router for selecting the correct expert. The computation begins by calculating the gate logits and deriving expert probabilities via the softmax function. The layer then computes the output for all experts, scaled by the probability distribution of using the experts. Finally, these weighted expert outputs are combined through a sum to compute the final output; the execution flow is shown in Figure 5.9.

5.5.5 Attention

An additional layer part of Utils implements the attention mechanism by directly using the formula from Section 4.4.6 [19]. The *AttentionLayer* first computes keys, values, and queries through a *nn.Linear* and transposing each matrix into the correct shape. The calculation is shown in code snippet 5.10.

[17] https://github.com/MeRichard123/ml-compiler/blob/main/src/Utils/scan.py#L27
[18] https://github.com/MeRichard123/ml-compiler/blob/main/src/Utils/Moe.py#L4
[19] https://github.com/MeRichard123/ml-compiler/blob/main/src/Utils/Attention.py#L21

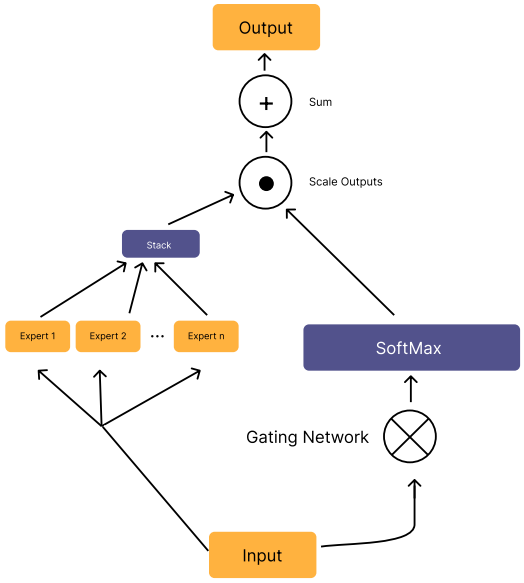


Figure 5.9: Computation Graph illustrating the execution flow of the Mixture of Experts Layer; the input is passed through a gating network and experts separately, and then combined using a product and sum.

```
# Calculate attention
energy = queries @ keys.transpose(-2, -1)
sqrt = self.head_dim**(1/2)
attention = torch.softmax(energy / sqrt, dim=-1)
out = attention @ values
```

Figure 5.10: Attention Computation, of the energy or compatibility score of QK^T, the sqrt $\sqrt{d_k}$, its SoftMax and finally scaling by values V

5.6 Evaluation

The models were evaluated after training via an Evaluator class [20] which uses the dataset *get_prompts* function to fetch all prompts from the test set. These are split into the input and expected output based on the position of the *<PROGRAM END>* token.

The class implements utilities to get the vector, which converts the input tokens using an index look-up in the *word2idx* dictionary, and casts to a tensor. Additionally, the

[20] https://github.com/MeRichard123/ml-compiler/blob/main/src/Evaluation.py#L17

vectors are padded so that the sizes match, making it possible to compute each metric.

Sample input is passed through the model to generate the model outputs to be compared against the ground truth. Comparisons are done by a set of functions which compute the metrics proposed in section 4.6.1. Cosine similarity is implemented through the PyTorch interface using *nn.functional.cosine_similarity*. The Exact Match procedure is implemented in Figure 5.11 by checking if every element matches that of the corresponding vector component.

```
def exact_match(self, vec1, vec2):
    if vec1.size(1) != vec2.size(1):
        return 0
    return int(torch.all(vec1 == vec2).item())
```

Figure 5.11: Python PyTorch Implementation of EM

Precision and recall are implemented using the set intersection operator [21] to find the overlap between the ground truth and predicted tokens and then compare against the length of the generated and expected tokens, respectively. As discussed in Section 4.6.1, F1 can be implemented in terms of precision and recall.

5.6.1 Sample K

The Pass@K metric is implemented through a separate *samplek* [22] method on the Language Model. This resamples the model *num_samples* times to generate a range of candidate outputs, which are used to determine whether at least one (for pass@1) candidate matches the correct output.

Pass@K is implemented in two functions in the evaluator, the first $Pass@K\_Single \in (0, 1)$, which computes the values for one sample, returning one in the case of a match, and 0 in the case of a miss.

The final pass@k computation is done by looping through each sample in the test

[21] https://github.com/MeRichard123/ml-compiler/blob/main/src/Evaluation.py#L168
[22] https://github.com/MeRichard123/ml-compiler/blob/main/src/LanguageModel.py#L273

set and accumulating the single Pass@K for 10 candidates per sample. Pass@K is finally calculated by averaging over the total number of samples.

5.7 Challenges

5.7.1 Overfitting

A major implementation challenge was data quality and leaning the model towards generalisation rather than over-fitting. This research implemented several forms of regularisation to help the model better generalise the dataset.

Label Smoothing

The first approach towards reducing over-fitting was altering the loss function to soften "hard" labels - labels in which the model was overconfident, which makes the model less certain and aims to improve generalisation. By reducing the probability of errors from true labels, the model becomes more robust in dealing with noise. Such a loss function is implemented using a smoothing value of 0.2 to compute the confidence and scale the probabilities. Finally, the loss function [23] computes Kullback-Lieber Divergence Loss $D_{KL}(P||Q)$, which measures how much a probability distribution Q deviates from a true probability P; such a metric is useful for unknown probability distributions.

L2 Regularisation

Another approach which improved generalisation was weight decay, often known as L2 regularisation. Weight decay reduces high weights to encourage smaller weights to have a larger effect on the output, discouraging reliance on one particular feature. This work uses a weight decay factor of $1e-3$ and the ADAM optimiser, which is implemented in the language model [24].

[23] https://github.com/MeRichard123/ml-compiler/blob/main/src/LanguageModel.py#L22
[24] https://github.com/MeRichard123/ml-compiler/blob/main/src/LanguageModel.py#L30

Dropout

Dropout drops a fraction of neurons in the work to prevent the model from becoming too reliant on any given Neuron, aiding in model generalisation because it prevents co-adaptation of neurons, forcing them to work independently and so prevents the model from overfitting. In this study, models use a dropout ratio of 0.5, dropping half the neurons on each timestep [25] of the model.

Teacher Forcing

Teacher forcing (or student forcing) keeps the model close to the ground truth by feeding the ground truth vectors at the end time step instead of the previous output vector [26]. This technique is often valuable for training RNNs (Kolen and Kremer, 2001) and has been shown to help them learn, because mistakes are not propagated through the model.

Learning Rate Scheduling

Another method which helps the model learn is learning rate scheduling, which can alter the learning rate during training to ensure the model is not learning too slowly or too quickly. This approach increases learning stability and helps the model avoid over-fitting. This study uses *ReduceLROnPlateau* [27], which reduces the learning rate by a factor of 0.5 if the loss remains unchanged for more than 10 iterations (patience 10).

5.7.2 Data Quality

Although Lua is a simple language with a lot of online code, most available Lua code consists of large amounts of embedded software, large production applications, and configurations. These code bases do not serve well for input and output examples, so Chatgpt generated a set of examples for training to counteract this.

[25] https://github.com/MeRichard123/ml-compiler/blob/main/src/Architectures/minGRU.py#L62

[26] https://github.com/MeRichard123/ml-compiler/blob/main/src/LanguageModel.py#L93

[27] https://github.com/MeRichard123/ml-compiler/blob/main/src/LanguageModel.py#L144

The downside of this approach is that the range of data that ChatGPT can generate is very sparse, and takes a lot of prompting to generate a reasonably sized dataset, to do attempt to counteract this, a data augmentation system [28] is set up to change each program slightly and provide more variety in examples.

[28] https://github.com/MeRichard123/ml-compiler/blob/main/src/Utils/DataAugumentation.py

Chapter 6

Results & Discussion

6.1 Computational Performance and Resources

Each model was trained for 3000 iterations, a batch size of 128, which are values found from hyperparameter tuning with optuna and $|V| = 951$ on a GPU. They all used an ADAM (Adaptive Moment Estimation) optimiser and the same number of parameters and identical loss functions. The training times are indicated in Table 6.1, showing that training time increases with the number of experts, due to the added complexity of the MoE layer. RNN training times are relatively short compared to large language models, which often take days or weeks to train (Kaplan *et al.*, 2020).

Table 6.1: Training times in minutes for each model by number of experts, and the time per iteration

model	train time per n experts				time per iteration per n experts			
	0	2	4	10	0	2	4	10
LSTM	55	63	73	96	0.0183	0.0213	0.0257	0.0323
GRU	47	58	63	90	0.0175	0.0193	0.0213	0.0303
RNN	48	53	61	82	0.0175	0.0177	0.0205	0.0275
MinGRU	79	83	85	111	0.0265	0.0278	0.0287	0.0372
MinLSTM	71	86	90	114	0.0237	0.0288	0.0302	0.0380

Table 6.2 summarises the timing of each step of the training pipeline, the test indicates no severe bottleneck in the execution, and the model achieves efficient times compared to larger models.

Table 6.2: Timings in Seconds for each step of the training pipeline

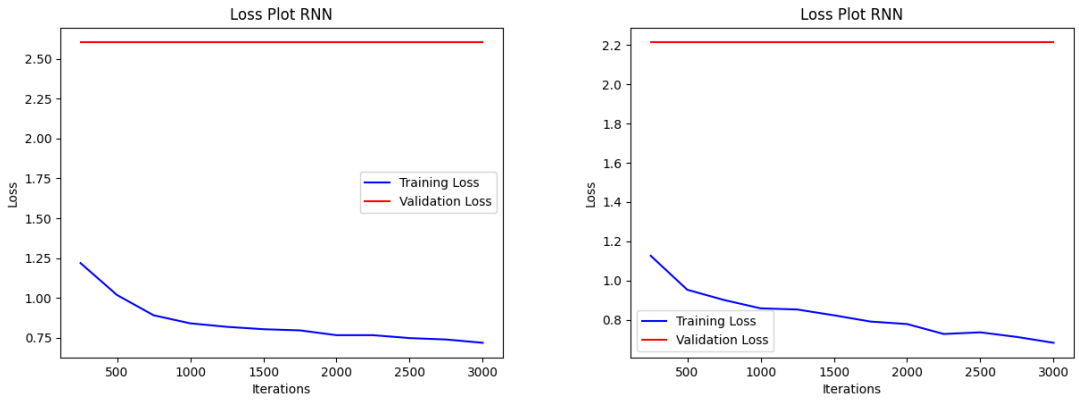
Step	Time
Tokenising	0.100
Training	1800
Sampling	0.207
Evaluation	6.034

6.2 Results

Results for carried out experiments are presented in Table 6.3; the following sections will describe the patterns within the results and visualise the outcomes.

6.2.1 RNN Performance

The standard RNN model achieved the worst overall performance, with a consistent value of 0.00 for Exact Match and Pass@1, indicating it failed to predict any test programs perfectly. The RNN also scored the highest in validation perplexity among tested architectures. F1 score indicated a low fraction of samples where the number of tokens matched the expected amount. The model also predicted some vector outputs that matched the expected output, since the cosine similarity is > 0.



(a) No Mixtures of Experts

(b) 2 Expert Networks

Figure 6.1: Vanilla RNN shows reduced loss using 2 experts rather than none, however, both models overfit the data.

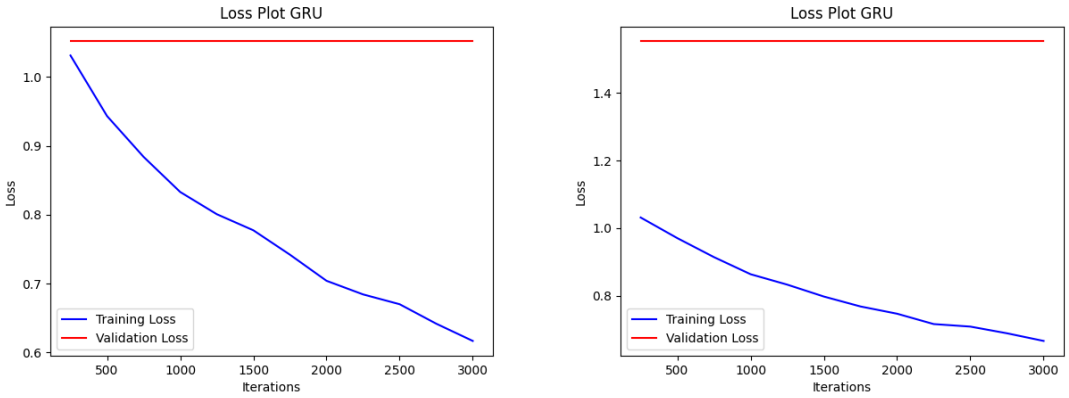
Figure 6.1 shows the loss plots during training and validation for the RNN model, which demonstrates that Mixture of Experts consistently produced a lower loss; it

should be noted that validation loss remained consistently higher than training loss. These results do not support vanilla RNNs in successfully emulating Lua programs.

Notably, the lack of improvement over different configurations was consistent across all standard RNN architectures. Additionally, the number of experts did not affect the production of the correct number of tokens, with the F1 score remaining low across each experiment.

6.2.2 GRU Performance

The GRU model performed similarly to the RNN model with similarly low results, with no perfect predictions and a low number of correctly predicted outputs based on the number of tokens outputted. The model showed slight improvement with increasing number of experts; however, F1 only differed by ±0.001 and therefore too low to be significant. Using four experts in this case achieved a higher F1 than other values. The GRU demonstrated a significantly lower perplexity compared to the vanilla RNN. The F1 score of 0.0093 does not follow the pattern of low values for the model and may be considered an outlier.



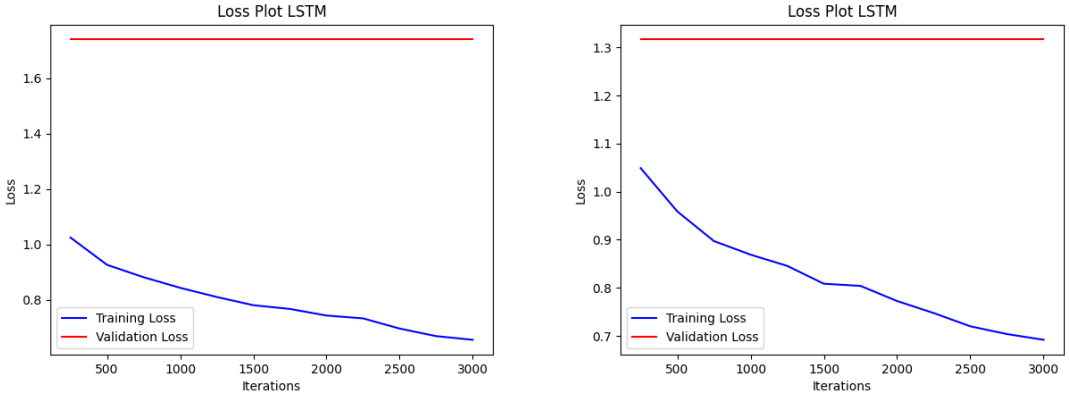
(a) No Mixtures of Experts. (b) 2 Expert Networks.

Figure 6.2: GRU model training and validation loss plots.

Figure 6.2 visualises the training and validation loss for the GRU, where a similar trend to the RNN arises. The plot indicates a lower loss in training and validation than the RNN, showing only slight improvement with an increasing number of expert networks.

6.2.3 LSTM Performance

The LSTM architecture indicated results proportionate to the GRU model, with low values and no correct outputs regarding Pass@1 and EM. Like the RNN, the LSTM model performed best with only two experts. This architecture also resulted in higher perplexity than the GRU. Regarding F1, the LSTM slightly improved over the GRU, predicting the correct number of words more frequently. Sentence Similarity remained consistent across all three RNN architectures, with a mean of 0.455 and a standard deviation of 0.02.



(a) No Mixtures of Experts.

(b) 2 Expert Networks.

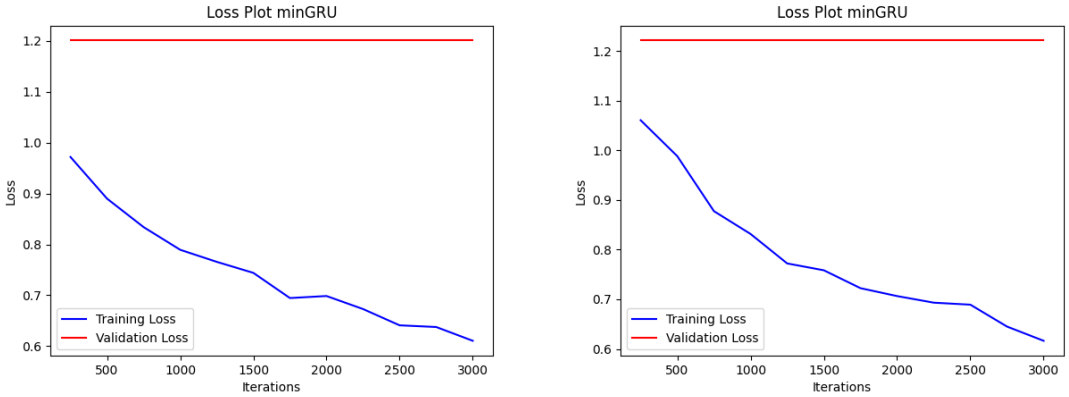
Figure 6.3: LSTM model training and validation loss plots.

Loss plots in Figure 6.3 indicate a similar trend to the GRU, except the LSTM shows a lower initial loss; however, the same trend of overfitting. The LSTM also presents a lower loss than the GRU and standard RNN.

6.2.4 MinGRU Performance

Minimal GRU improves over previous RNN architectures, with higher F1, precision and recall compared to GRUs, standard RNNs, and the LSTM, reaching 0.0391 with two experts - this is an improvement by a factor 10×. This architecture performs best with two expert networks, with 0.0115 for EM, showing that some examples were correctly predicted; further backed up by a score of 1.1494 on pass@k, indicating at least one predicted token was correct on test examples.

The MinGRU architecture achieved more consistent perplexity of around 3.36 ($\sigma =$



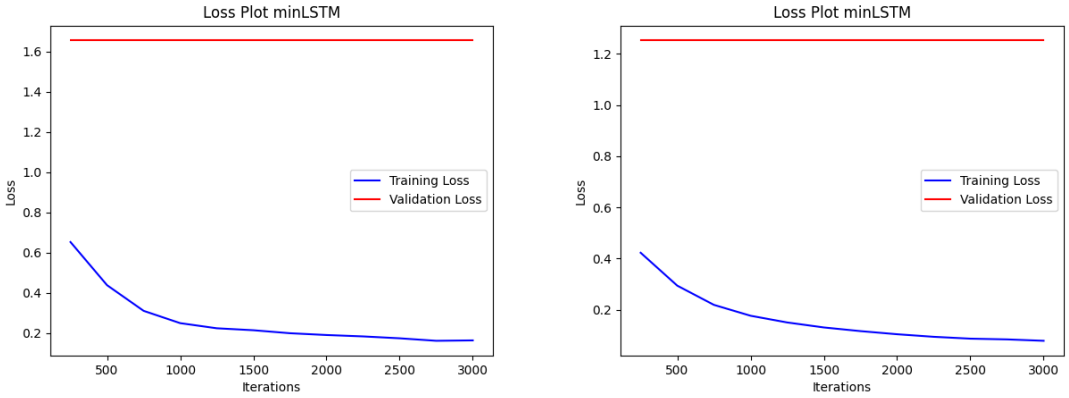
(a) No Mixtures of Experts. (b) 2 Expert Networks.

Figure 6.4: minGRU model training and validation loss plots.

0.18). Figure 6.4 shows an improved loss over other RNN architectures; however, it overfits on the validation set.

6.2.5 MinLSTM Performance

The minLSTM showed similar performance to the minGRU model, also achieving 0.0115 on EM and 1.1494 on Pass@1, however, using four experts rather than two. F1 scores improve over basic RNNs, but do not outperform the minGRU. Sentence Similarity scores are similar to those of the minGRU with a mean of 0.668 ($\sigma = 0.02$). LSTM perplexity is higher, indicating the model is less certain of its output than the minGRU.



(a) No Mixtures of Experts. (b) 2 Expert Networks.

Figure 6.5: minLSTM model training and validation loss plots.

Table 6.3: Experiment Results per model such that hidden state size (h), number of experts ($expts$), and training steps. The model was evaluated on a Exact Match (EM), F1 Score ($F1$), Sentence Similarity (SS), Pass@1 ($P@1$), Validation Perplexity ($Perp$), Recall (Rcl) and Precision (Prs).

	h	expts	train	EM	F1	SS	P@1	Perp	Rcl	Prs
base	900	0	3000							
RNN				0.0000	0.0077	0.4414	0.0000	13.4979	0.0161	0.0053
		2		0.0000	0.0051	0.4815	0.0000	9.1533	0.0094	0.0037
		4		0.0000	0.0018	0.4411	0.0000	13.2032	0.0115	0.0010
		10		0.0000	0.0044	0.4800	0.0000	6.0734	0.0077	0.0033
GRU				0.0000	0.0010	0.4725	0.0000	2.8622	0.0023	0.0006
		2		0.0000	0.0018	0.4685	0.0000	4.7202	0.0057	0.0010
		4		0.0000	0.0093	0.4532	0.0000	2.8042	0.0157	0.0068
		10		0.0000	0.0019	0.4196	0.0000	2.8754	0.0038	0.0013
LSTM				0.0000	0.0000	0.4649	0.0000	5.6946	0.0000	0.0000
		2		0.0000	0.0032	0.4352	0.0000	3.7309	0.0080	0.0020
		4		0.0000	0.0027	0.4582	0.0000	6.57226	0.0138	0.0016
		10		0.0000	0.0011	0.4553	0.0000	3.7037	0.0057	0.0006
Mini GRU				0.0000	0.0067	0.6645	0.0000	3.6344	0.0047	0.0126
		2		0.0115	0.0391	0.7030	1.1494	3.2739	0.0363	0.0447
		4		0.0000	0.0021	0.6593	0.0000	3.4013	0.0018	0.0024
		10		0.0000	0.0226	0.6793	0.0000	3.1345	0.0222	0.0253
Mini LSTM				0.0000	0.0067	0.6398	0.0000	3.3153	0.0061	0.0077
		2		0.0000	0.0084	0.6646	1.1494	3.4999	0.0095	0.0079
		4		0.0115	0.0193	0.6673	0.0000	2.9899	0.0200	0.0191
		10		0.0000	0.0109	0.6649	0.0000	5.2299	0.0073	0.0238

Loss plots for the minLSTM (Figure 6.5) indicate a reduced loss compared to the GRU equivalent.

6.2.6 Inference Times

Models and baseline models were also compared in terms of inference time - the time between passing a prompt to the model and receiving an output. This was done for baselines by timing response times from *Ollama* after the model has been cached to avoid any cold start-up latencies. Table 6.4 indicates that smaller RNN models perform significantly faster in terms of inference for both short and long code snippets.

Table 6.4: Depicts Inference Time of each model expressed in seconds, with respect to small and large code snippets.

Model	Inference Time	
	Large Code Snippet	Small Code Snippet
RNN	0.0125	0.07495
LSTM	0.009375	0.0325
GRU	0.009	0.032525
MinGRU	0.0275	0.1885
MinLSTM	0.0221	0.2171
Starcoder2:3b	5.781	4.420
qwen2.5-coder	4.387	4.236

6.2.7 Example Outputs

Figures 6.6 to 6.10 present the model outputs for a simple program that checks if a number is negative or not. All generations for this problem were incorrect however, low perplexity models such as the LSTM, GRU, and minimal version show a greater level of understanding and were able to generate a string.

```
value = -1
if value < 0 then
    print("Negative number")
else
    print("Non-negative number")
end
<PROGRAM END>
FUNCTION_CALL(max_number) STRING(30) IDENTIFIER(lastName)
STRING(odd)
```

Figure 6.6: RNN generation output, has string as the second token, however with such a high perplexity, it can be said the model is guessing.

```
value = -1
if value < 0 then
    print("Negative number")
else
    print("Non-negative number")
end
<PROGRAM END>
STRING(odd) STRING(Yearning) STRING(odd) NUMBER(6)
STRING(odd) STRING(Light) STRING(clever)
```

Figure 6.7: GRU generation, showed some improvement and generated some strings and a number

```
value = -1
if value < 0 then
    print("Negative number")
else
    print("Non-negative number")
end
<PROGRAM END>
STRING(Cacophony) STRING(Dog) NUMBER(60) NUMBER(0)
STRING(slow) NUMBER(90) STRING(back) FUNCTION_CALL(cube)
```

Figure 6.8: LSTM generation was similar to GRU except generated a function call, which aligns with the increased perplexity.

```
value = -1
if value < 0 then
    print("Negative number")
else
    print("Non-negative number")
end
<PROGRAM END>
STRING(equal) STRING(repeatuntil) FUNCTION_CALL(math.floor)
STRING(rising) STRING(Lua) IDENTIFIER(b) IDENTIFIER(add)
```

Figure 6.9: MinLSTM generation, generated strings, identifiers and a function call indicating high perplexity.

```
value = -1
if value < 0 then
    print("Negative number")
else
    print("Non-negative number")
end
<PROGRAM END>
STRING(wet) STRING(chilly) FUNCTION_CALL(string.upper)
STRING(Live) IDENTIFIER(temperature)
```

Figure 6.10: MinGRU output is very similar semantically to the MinLSTM.

6.3 Discussion

6.3.1 Dataset Collection

The first objective identified in Chapter One was curating a dataset of at least 500 samples of Lua code. During the course of this study, 677 samples of Lua were collected; most of the samples were stand-alone functions or programs demonstrating different aspects of Lua. The size of the dataset, however, was too small for the RNN models to effectively learn the necessary patterns for Lua code output prediction, and the models struggled to learn complex structures. Burgueño *et al.* (2021); Rahman, Watanobe and Keita Nakamura (2020) and Reed and Freitas (2015) found that this discrepancy could be attributed to data quality; specifically, code samples that were meaningful enough for the model to learn how to interpret them. This study found that code data available online, which was not tailored for code generation, often is not meaningful enough for a model to learn semantics, since open source code samples tend to be part of larger interconnected code bases, which could result in more noise in the model training.

An alternative approach would be the use of more recent instruct datasets for code generation, such as OpenCodeInstruct (Ahmad *et al.*, 2025). Such a method may be able to better guide the model in predicting the correct result, instructing it in the right way.

6.3.2 Model Development

This research developed five RNN architectures and carried out experiments to determine the feasibility of their use in emulating Lua execution. The findings show that the tested models struggled to generalise and predict the outputs of Lua, which is supported by previous studies that also observed similar results (Reed and Freitas, 2015).

Architecture Performance

Non-minimal RNN architectures, like the Standard RNN, GRU and LSTM, performed worse than the minimal version, which suggests that these simpler architectures have an advantage over the older, more specialised ones. This could be due to simpler architectures improving gradient flow (Tessera, Hooker and Rosman, 2021), and such reduces issues with gradients exploding or vanishing. An alternative explanation for these results is that similar architectures can often act as an implicit form of regularisation (Zhang *et al.*, 2017). They show that simpler models are less likely to overfit and, as a result, generalise better. Implicit regularisation could also explain why the minGRU model performed marginally better than the minLSTM model. Further research is required, to determine whether the performance gains suggest that the gating mechanism affected the ability to remember variables.

The RNN appeared to perform the worst out of the three non-minimal architectures. One factor which could explain this observation is that RNNs suffer from both vanishing and exploding gradients, which may have made it harder to learn longer, more complex dependencies. This could lead to the model not generalising and failing to learn the correct patterns.

Mixture of Experts

Results show that using a mixture of experts approach lowered the training and validation loss of the tested models; however, interestingly, it had little effect on how well the model generalised. This result is contrary to Chazan, Goldberger and Gannot (2017), who found that MoE enhanced their speech recognition model to

achieve better results. It seems possible that MoE increased the model sparsity, resulting in poor performance for more than two experts and therefore impeding performance.

Alternative Architectures

In future investigations, it may be possible to use different model architectures to achieve significantly better results. One approach would be to use an Encoder-Decoder RNN structure with Cross-Attention, the Encoder-Decoder (Balog *et al.*, 2017). In the cited study, the researchers proved that this architecture is effective in a code generation setting. Such an architecture could allow the model to separate the task of understanding and generation and ensure the model becomes more specialised in its task.

6.3.3 Interpreting Results

Generalisation vs Overfitting

All models showed clear signs of overfitting, evident in the disparity in training and validation loss in Figures 6.1,6.2, 6.3, 6.5, and 6.4. Extensive regularisation was attempted; however, it did not present enough improvement from regularisation alone. The validation and training loss gap was consistent across all models, but was narrowest in the minGRU, implying better generalisation, despite the use of mechanisms like Attention and Mixture of Experts. The reasons for this have already been discussed; however, most strikingly, even the minimal models failed to generalise for Lua emulation, most likely due to the availability of a large amount of quality Lua code for training.

Comparison with Baselines

Standard RNNs failed at the task of predicting any output correctly, confirming their known limitations in handling long-term dependencies and vanishing gradients. GRU and LSTM showed slight improvements with lower perplexity; however, they struggled with inference. MinGRU significantly outperformed the tested ar-

chitectures, reaching a 10× improvement in F1. The minLSTM followed closely, achieving similar Pass@1 scores, but performed slightly worse in terms of F1 and Perplexity.

Model	EM	SS	F1	Prs	Rcl
StarCoder2	0.0365	0.2376	0.1203	0.0994	0.2465
Qwen2.5Coder	0.9184	0.9744	0.9435	0.9357	0.9620

Table 6.5: Baseline Results for StarCoder and Qwen 2.5 Coder

Baseline model results are shown in Table 6.5, and show a substantial improvement on the testing set. Both upper baseline models are transformer-based and use a significantly larger amount of training data, which is one of the main reasons they outperform the RNN models in this study. StarCoder2 underperforms Qwen due to generating a lot of extra output, containing unneeded feedback, even after a large amount of prompting.

Qwen shows impressive results in generating the correct output for provided programs, only failing on very complex examples involving complex tree structures like recursion.

Errors and Limitations

All models struggled with perfect sequence prediction, indicating challenges in modelling program semantics and associating input tokens with the output tokens.

A low F1 score suggests that all RNN models had issues with over-generation, even if individual tokens are semantically plausible. Sentence similarly provided some insight into the models predicting semantically correct tokens, this is evident in Table 6.3 where cosine similarity reaches values in the range of $0.4 - 0.6$. This does not always indicate correctness, due to the semantics of the program, but can show that some semantic tokens, such as "STRING()" and "NUMBER()", were correctly identified. However, a better method, such as checking only the semantic tokens, would have been a more beneficial approach.

6.3.4 Implications for the Real world

This study has shown that RNN models have a significantly faster inference time, and therefore may show promising results for code generation and code output prediction, should they be trained with more data and the presented alternative architectures. Due to their faster inference times, RNN architectures may find use in embedded systems or devices where latency is critical.

6.3.5 Limitations of Study

This study is limited in its dataset size, 677 samples likely limit the performance and generalizability of the models. A larger, more diverse dataset may have provided some improvements and increased validity.

Additionally, RNN architectures are known to struggle with complex tasks, such that limiting the study to only single-unit models, where in each model's case, a single stand-alone unit of that model is used.

Finally, this study lacked any upper RNN baselines with which time compare against, making the upper baselines a very challenging goal due to the complexity and resources that large language models have to train.

Chapter 7

Conclusion

7.1 Findings

This study explored the viability of small models to emulate Lua compilation, with a focus on RNN variants, more specifically the LSTM, GRU, Standard RNN and their minimal variants (minGRU and minLSTM) to learn from input-output examples of Lua code. Additionally, these models used an Attention and Mixture of Experts approach to enhance their ability to learn more complex patterns. While these models demonstrated the capacity to predict some outputs, they fail to learn semantic patterns and to generalise for complex tasks. The research found that the models are too simple for such a complex task and overfit in almost all cases, despite extensive regularisation techniques such as Label Smoothing and Dropout.

Standard RNN were found to perform the worst, consistently failing to predict the outputs, and exhibiting the highest perplexity. The GRU and LSTM models performed better than the RNN with a higher F1, but still failed to predict outputs; the GRU demonstrated a lower perplexity than both the RNN and LSTM models. With regards to predicting the correct number of output tokens, the LSTM performed better than the GRU model.

Minimal architectures showed improvement over the standard models due to implicit regularisation; the minGRU achieved a ten-fold improvement in F1 using 2 experts, making some correct predictions on the validation set. The minLSTM performed worse than the minGRU in terms of F1 and Perplexity scores.

A mixture of experts approach showed promising results in lowering the loss and improved ability to generalise; however, exceeding 2 experts resulted in sparsity issues in the models. Self-attention also showed benefits in lowering the training and validation loss.

7.2 Contributions

This research contributes a novel perspective that RNNs are promising in a high-performance setting where inference latency is a key factor. They achieve fast training and inference times, which can be useful in situations like embedded software and in editor code completion tools. Additional contributions of this work are listed below.

- Systematic Evaluation of RNN models and application of Minimal RNN architectures with MoE for code generation for Lua programs.
- Contextualises the trade-off between inference efficiency and output accuracy.
- Use of semantic embeddings to allow models to learn semantic features of a language.

7.3 Limitations

This study has some limitations which may have influenced the results. The primary limitation was that the tested models struggled to generalise completely to predict Lua outputs. This may be attributed to architectural limitations of RNN models failing to capture long-range dependencies and learn complex patterns. Future work could explore more expressive architectures, such as encoder-decoder networks or transformer architectures.

Another limitation lies in the lack of strong upper RNN baselines, there are very few recent RNN models which are trained on Lua code, making the performance gap between large transformer models seem particularly vast. One way this limita-

tion may be overcome is through the use of languages which are common for code generation tasks, such as Python, where established benchmarks exist.

The dataset itself imposed constraints, consisting of 677 Lua programs, ranging from simple to more complex. Each sample was stand-alone, and so the models could not benefit from a broader context. As a result, the models only had the opportunity to learn shallow-token-level patterns. To counteract the quality and size of the dataset, an existing instruct dataset could have been used.

Finally, the choice of evaluation metrics may have limited the range of assumptions it is possible to make from the research results. The chosen metrics provided a sufficient overview of how well the models performed, however, they lacked expressiveness in terms of the semantic understanding of the models.

7.4 Future Work

Further studies should consider training and evaluating a larger range of architectures, such as the encoder-decoder architecture with cross-attention. These architectures could help separate concerns by assigning the task of understanding and output generation to distinct components, improving generalisation.

In addition, reinforcement learning (RL) provides a promising avenue for rewarding the model during loss computations for correct and incorrect outputs, aligning the model more closely to emulating compilation (Zaremba, Mikolov *et al.*, 2016), particularly in cases where token-level accuracy does not fully capture the program semantics.

The study used a limited Lua dataset, which limited its approach. Further studies should consider using a larger dataset with more semantic information, such as the Instruct dataset (Ahmad *et al.*, 2025), to more closely aid the model in reaching an output.

Further investigation and experimentation into pre-training and fine-tuning is strongly recommended. Pre-training could aid the model in learning syntax and semantics, potentially leading to faster convergence. Subsequent fine-tuning could then help

the model in adapting its knowledge of language grammar towards execution emulation. Such a method may also prevent the model from over-fitting and improve generalisation.

7.5 Self-Reflection

Throughout the course of this research, there are many things that succeeded. The project succeeded in evaluating models for code generation using attention and mixtures of experts, and gave an insight into the task of compiler emulation. It was also advantageous in terms of software architecture and organisation, creating a reusable evaluation system that effectively evaluates a range of architectures in a large-scale machine learning system.

Conversely, this study has its drawbacks, which lie in the specific architecture selection and range of data used. These drawbacks limited the study and provided avenues for future work and development for compiler emulation.

Finally, to conclude the project provided many lessons, and enhanced personal understanding in the field of deep learning and language modelling sparking further interest in the matter.

References

Afzal, Arshia *et al.* (2025). *Linear Attention for Efficient Bidirectional Sequence Modeling*. arXiv: 2502.16249 [cs.LG]. URL: https://arxiv.org/abs/2502.16249 (cit. on p. 17).

Ahmad, Wasi Uddin *et al.* (2025). *OpenCodeInstruct: A Large-scale Instruction Tuning Dataset for Code LLMs*. arXiv: 2504.04030 [cs.SE]. URL: https://arxiv.org/abs/2504.04030 (cit. on pp. 59, 66).

Akyürek, Ekin *et al.* (2024). *In-Context Language Learning: Architectures and Algorithms*. arXiv: 2401.12973 [cs.CL]. URL: https://arxiv.org/abs/2401.12973 (cit. on p. 17).

Balog, Matej *et al.* (2017). 'DeepCoder: Learning to Write Programs'. In: *Proceedings of ICLR'17*. Microsoft. URL: https://www.microsoft.com/en-us/research/publication/deepcoder-learning-write-programs/ (cit. on pp. 14, 61).

Bengio, Yoshua *et al.* (2009). 'Curriculum learning'. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: Association for Computing Machinery, pp. 41–48. ISBN: 9781605585161. DOI: 10.1145/1553374.1553380. URL: https://doi.org/10.1145/1553374.1553380 (cit. on p. 28).

Blelloch, Guy E. (1990). *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. "School of Computer Science, Carnegie Mellon University", pp. 35–60. DOI: 10.1184/R1/6608579.V1. URL: https://www.semanticscholar.org/paper/Prefix-sums-and-their-applications-Blelloch/ (cit. on pp. 9, 12).

Bui, Nghi D. Q. *et al.* (2023). 'CODETF: One-Stop Transformer Libary for State-Of-The-Art Code LLM'. In: DOI: https://doi.org/10.48550/arXiv.2306.00029. URL: https://arxiv.org/abs/2306.00029 (cit. on p. 17).

Burgueño, Loli *et al.* (May 2021). 'A generic LSTM neural network architecture to infer heterogeneous model transformations'. In: *Software and Systems Modeling* 21.1, p. 139. DOI: 10.1007/s10270-021-00893-y (cit. on pp. 12, 14, 59).

Chazan, Shlomo E., Jacob Goldberger and Sharon Gannot (2017). 'Deep recurrent mixture of experts for speech enhancement'. In: *2017 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pp. 359–363. DOI: 10.1109/WASPAA.2017.8170055 (cit. on pp. 27, 60).

Chen, Mark *et al.* (2021). 'Evaluating Large Language Models Trained on Code'. In: *ArXiv* abs/2107.03374. URL: https://api.semanticscholar.org/CorpusID:235755472 (cit. on pp. 17, 30).

Cho, Kyunghyun *et al.* (Oct. 2014). 'Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation'. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: https://aclanthology.org/D14-1179/ (cit. on p. 8).

Cummins, Chris, Dejan Grubisic, Rozière Baptiste *et al.* (2024). 'Meta Large Language Model Compiler: Foundation Models of Compiler Optimization'. In: URL: https://ai.meta.com/research/publications/meta-large-language-model-compiler-foundation-models-of-compiler-optimization/ (cit. on p. 2).

Cummins, Chris, Dejan Grubisic, Mostafa Elhoushi *et al.* (2023). 'Large Language Models for Compiler Optimization'. In: URL: https://arxiv.org/abs/2309.07062 (cit. on p. 17).

De, Soham *et al.* (2024). 'Griffin: Mixing Gated Linear Recurrences with Local Attention for Efficient Language Models'. In: *ArXiv* abs/2402.19427. URL: https://api.semanticscholar.org/CorpusID:268091246 (cit. on p. 17).

DeepSeek-AI (2025). *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948 (cit. on p. 27).

Feng, Leo, Frederick Tung, Mohamed Osama Ahmed *et al.* (Oct. 2024). *Were RNNs All We Needed?* arXiv: 2410.01201 [cs.LG]. URL: https://arxiv.org/abs/2410.01201 (cit. on pp. 9, 11, 17, 26, 44, 45).

Feng, Leo, Frederick Tung, Hossein Hajimirsadeghi *et al.* (2024). 'Attention as an RNN'. In: *arXiv preprint arXiv:2405.13956* (cit. on p. 27).

Graves, Alex *et al.* (Dec. 2014). 'Neural Turing Machines'. In: URL: https://arxiv.org/abs/1410.5401 (cit. on pp. 12, 25).

Grossberg, Stephen (1987). 'Competitive Learning: From Interactive Activation to Adaptive Resonance'. In: *Cognitive Science* 11.1, pp. 23–63. DOI: https://doi.org/10.1111/j.1551-6708.1987.tb00862.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1551-6708.1987.tb00862.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1551-6708.1987.tb00862.x (cit. on p. 10).

Grubisic, Dejan *et al.* (2024). 'Priority Sampling of Large Language Models for Compilers'. In: *EuroMLSys '24: Proceedings of the 4th Workshop on Machine Learning and Systems*. Association for Computing Machinery, pp. 91–97. DOI: 10.1145/3642970.3655831 (cit. on p. 17).

Hebb, D. O. (1950). 'The organization of behavior: A neuropsychological theory. New York: John Wiley and Sons, Inc., 1949. 335 p. $4.00'. In: *Science Education* 34.5, pp. 336–337. DOI: https://doi.org/10.1002/sce.37303405110. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sce.37303405110. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sce.37303405110 (cit. on p. 10).

Hochreiter, S. and J. Schmidhuber (1997). 'Long Short-Term Memory'. In: *Neural computation* 9.8. ID: 1, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735 (cit. on pp. 7, 11).

Hu, Zheyuan *et al.* (2024). 'State-space models are accurate and efficient neural operators for dynamical systems'. In: *arXiv preprint arXiv:2409.03231* (cit. on p. 18).

Ierusalimschy, Roberto and Luiz Henrique de Figueiredo (2009). 'Interview about Lua'. In: *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. Ed. by Federico Biancuzzi and Shane Warden. O'Reilly Media, pp. 175–190 (cit. on p. 28).

Imada, Keita and Katsuhiko Nakamura (2008). 'Towards machine learning of grammars and compilers of programming languages'. In: *Proceedings of the 2008th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part II*. ECMLPKDD'08. Antwerp, Belgium: Springer-Verlag, pp. 98–112. ISBN: 3540874801. DOI: 10.1007/978-3-540-87481-2_7. URL: https://doi.org/10.1007/978-3-540-87481-2_7 (cit. on p. 2).

Jacobs, Robert A. *et al.* (1991). 'Adaptive Mixtures of Local Experts'. In: *Neural Computation* 3.1, pp. 79–87. DOI: 10.1162/neco.1991.3.1.79 (cit. on pp. 10, 45).

Kaplan, Jared *et al.* (2020). 'Scaling Laws for Neural Language Models'. In: *CoRR* abs/2001.08361. arXiv: 2001.08361. URL: https://arxiv.org/abs/2001.08361 (cit. on p. 51).

Kolen, John F and Stefan C Kremer (2001). *A field guide to dynamical recurrent networks*. John Wiley & Sons (cit. on p. 49).

Laich, Larissa, Pavol Bielik and Martin Vechev (2020). 'Guiding Program Synthesis by Learning to Generate Examples'. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=BJl07ySKvS (cit. on pp. 14, 15).

Leather, H. and C. Cummins (2020). 'Machine Learning in Compilers: Past, Present and Future'. In: *2020 Forum for Specification and Design Languages (FDL)*. ID: 1, pp. 1–8. ISBN: 1636-9874. DOI: 10.1109/FDL50818.2020.9232934 (cit. on p. 2).

Liu, Fusheng and Qianxiao Li (2024). *From Generalization Analysis to Optimization Designs for State Space Models*. arXiv: 2405.02670 [cs.LG]. URL: https://arxiv.org/abs/2405.02670 (cit. on p. 17).

Luccioni, Alexandra Sasha, Sylvain Viguier and Anne-Laure Ligozat (2022). *Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model*. arXiv: 2211.02001 [cs.LG]. URL: https://arxiv.org/abs/2211.02001 (cit. on p. 33).

Noh, Seol-Hyun (2021). 'Analysis of Gradient Vanishing of RNNs and Performance Comparison'. In: *Information* 12.11. ISSN: 2078-2489. DOI: 10.3390/info12110442. URL: https://www.mdpi.com/2078-2489/12/11/442 (cit. on p. 26).

Parnichkun, Rom N. *et al.* (2024). *State-Free Inference of State-Space Models: The Transfer Function Approach*. arXiv: 2405.06147 [cs.LG]. URL: https://arxiv.org/abs/2405.06147 (cit. on p. 17).

Patwardhan, Narendra, Stefano Marrone and Carlo Sansone (Apr. 2023). 'Transformers in the Real World: A Survey on NLP Applications'. In: *Information* 14.4. DOI: 10.3390/info14040242 (cit. on p. 11).

Priya, Renita *et al.* (2017). 'A Deep Dive into Automatic Code Generation Using Character Based Recurrent Neural Networks'. In: *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 369–374. DOI: 10.1109/CSCI.2017.61 (cit. on pp. 12, 14, 25).

Rahman, Md. Mostafizer, Yutaka Watanobe and Keita Nakamura (2020). 'A Neural Network Based Intelligent Support Model for Program Code Completion'. In: *Sci. Program.* 2020, 7426461:1–7426461:18. URL: https://api.semanticscholar.org/CorpusID:225584181 (cit. on pp. 12, 14, 15, 25, 29, 59).

Reed, Scott and Nando De Freitas (2015). 'Neural Programmer-Interpreters'. In: URL: http://arxiv.org/abs/1511.06279 (cit. on pp. 14, 15, 25, 28, 59, 60).

Sieber, Jerome *et al.* (2024). *Understanding the differences in Foundation Models: Attention, State Space Models, and Recurrent Neural Networks*. arXiv: 2405.15731 [cs.LG]. URL: https://arxiv.org/abs/2405.15731 (cit. on p. 17).

Strubell, Emma, Ananya Ganesh and Andrew McCallum (July 2019). 'Energy and Policy Considerations for Deep Learning in NLP'. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen, David Traum and Lluís Màrquez. Florence, Italy: Association for Computational Linguistics, pp. 3645–3650. DOI: 10.18653/v1/P19-1355. URL: https://aclanthology.org/P19-1355/ (cit. on p. 33).

Tessera, Kale-ab, Sara Hooker and Benjamin Rosman (2021). 'Keep the gradients flowing: Using gradient flow to study sparse network optimization'. In: *arXiv preprint arXiv:2102.01670* (cit. on p. 60).

Vaswani, Ashish *et al.* (2017). 'Attention Is All You Need'. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Long Beach, California, USA: Curran Associates Inc., pp. 6000–6010. DOI: 10.5555/3295222.3295349. URL: http://arxiv.org/abs/1706.03762 (cit. on pp. 11, 27).

Zaremba, Wojciech, Tomas Mikolov *et al.* (June 2016). 'Learning Simple Algorithms From Examples'. In: *The 33rd International Conference on Machine Learning*, 48:421–429. DOI: 10.48550/arXiv.1511.07275 (cit. on pp. 12, 25, 26, 28, 66).

Zaremba, Wojciech and Ilya Sutskever (Oct. 2014). 'Learning to Execute'. In: *arXiv (Cornell University)*. DOI: 10.48550/arxiv.1410.4615. URL: https://arxiv.org/abs/1410.4615 (cit. on pp. 12, 15, 25).

Zhang, Chiyuan *et al.* (2017). *Understanding deep learning requires rethinking generalization*. arXiv: 1611.03530 [cs.LG]. URL: https://arxiv.org/abs/1611.03530 (cit. on p. 60).

Zhao, Yunmei *et al.* (2023). 'An interpretable LSTM deep learning model predicts the time-dependent swelling behavior in CERCER composite fuels'. In: *Materials Today Communications* 37, p. 106998. ISSN: 2352-4928. DOI: https://doi.org/10.1016/j.mtcomm.2023.106998. URL: https://www.sciencedirect.com/science/article/pii/S2352492823016896 (cit. on p. 25).

Appendix

Acronyms

ANC	Adaptive Neural Compilation
AST	Abstract Syntax Tree
BLEU	Bilingual Evaluation Understudy
BPTT	Backpropagation Through Time
CoT	Chain of Thought
CNN	Convolutional Neural Network
DSL	Domain Specific Language
GAN	Generative Adversarial Network
GPT	Generative Pre-trained Transformer
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
MinLSTM	Minimal Long Short-Term Memory
MLP	Multi-Layer Perceptron
NLP	Natural Language Processing
NPI	Neural Programmer-Interpreters
PoT	Program of Thought
RAG	Retrieval Augmented Generation
RNN	Recurrent Neural Network