

## University of Southampton Research Repository

Copyright © and Moral Rights for this thesis and, where applicable, any accompanying data are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis and the accompanying data cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content of the thesis and accompanying research data (where applicable) must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holder/s.

When referring to this thesis and any accompanying data, full bibliographic details must be given, e.g.

Thesis: Bhumika Mistry (2023) "Neural Arithmetic Logic Modules", University of Southampton, Faculty of Engineering and Physical Science, Department of Electronics and Computer Sciences, PhD Thesis, 1-214.



**UNIVERSITY OF SOUTHAMPTON**

Faculty of Engineering and the Physical Sciences  
School of Electronics and Computer Science  
Vision, Learning and Control

**An Investigation into Neural Arithmetic  
Logic Modules**

*by*

**Bhumika Mistry**

ORCID: 0000-0003-4555-0121

*A thesis for the degree of  
Doctor of Philosophy*

July 2023



University of Southampton

Abstract

Faculty of Engineering and the Physical Sciences  
School of Electronics and Computer Science

Doctor of Philosophy

**An Investigation into Neural Arithmetic Logic Modules**

by Bhumika Mistry

The human ability to learn and reuse skills in a systematic manner is critical to our daily routines. For example, having the skills for executing the basic arithmetic operations ( $+$ ,  $-$ ,  $\times$ ,  $\div$ ) allows a person to perform a variety of tasks including budgeting expenses, scaling measurements to the desired proportions when cooking/baking, and planning travel schedules. Machine Learning (ML) can reduce the manual workload for humans, inferring underlying relations within the data without the need for heavy feature engineering. However, the ability of such models to extrapolate and generalise to unseen data in an interpretable manner is challenging. With this challenge in mind, Neural Arithmetic Logic Modules (NALMs) have been developed. Such parameterised modules, specialised for arithmetic operations, are designed to guarantee generalisation if weights are correctly learned and be interpretable in what they learn. This thesis seeks to thoroughly investigate the proposition that such specialised differentiable modules with inductive biases toward arithmetic can be learned, uncovering the limitations which remain. In this work, we begin by studying the extent to which NALMs are able to learn arithmetic. We initially provide a comprehensive review of existing NALMs and take our analysis a step further with empirical results on a new benchmark with evaluation metrics specifically for measuring extrapolation performance. From this, we identify two arithmetic operations to further investigate, namely multiplication and division. For multiplication, we show how stochasticity can be applied to alleviate issues regarding falling into local minimas which cannot extrapolate. For division, we show through an extensive set of empirical results the mechanisms which can aid and hinder robustness. Factors other than the architecture are investigated including using images as the input modality, using a different loss criterion and feature scaling. In the final chapter, we draw inspiration from a human cognitive theory, the Global Workspace Theory (GWT), to develop an end-to-end architecture to combine different NALMs for compositional arithmetic.



# Contents

<b>Declaration of Authorship</b>	<b>xi</b>
<b>Acknowledgements</b>	<b>xiii</b>
<b>Definitions and Abbreviations</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 High-level Reasoning in Humans . . . . .	2
1.2 Inductive Biases . . . . .	3
1.3 Inductive Biases for System 2 . . . . .	5
1.4 A Stepping Stone towards Human Reasoning: Learning Mathematics . .	7
1.4.1 Motivating Specialist Modules over Generic MLPs . . . . .	8
1.4.1.1 Results . . . . .	10
1.4.2 Shortcomings in Mathematical Reasoning when using Transform- ers . . . . .	12
1.4.3 Representing Numbers in Machine Learning Models . . . . .	16
1.4.3.1 Static Encoding . . . . .	17
1.4.3.2 Learnable Embeddings . . . . .	19
1.5 Discovering Mathematical Expressions for Symbolic Regression . . . . .	20
1.6 Research Questions and Contributions . . . . .	21
1.6.1 Research Questions . . . . .	21
1.6.2 Contributions . . . . .	22
1.7 Thesis Structure . . . . .	25
<b>2 Review of NALMs</b>	<b>27</b>
2.1 What are NALMs and Why use them? . . . . .	27
2.1.1 What is a NALM? . . . . .	27
2.1.2 What is the Aim of a NALM? . . . . .	28
2.1.3 Why is a NALM useful? . . . . .	29
2.2 Existing NALMs Architectures . . . . .	30
2.2.1 NALU . . . . .	31
2.2.2 iNALU . . . . .	32
2.2.3 NAU and NMU . . . . .	34
2.2.4 NPU and Real NPU . . . . .	36
2.2.5 G-NALU . . . . .	37
2.2.6 NLRL . . . . .	38
2.2.7 NSR . . . . .	40
2.3 NALU's Shortcomings and Existing Solutions . . . . .	42

2.3.1	Negative Inputs and Negative Outputs . . . . .	43
2.3.2	Gating Parameter Convergence . . . . .	43
2.3.3	Bias Considerations . . . . .	44
2.3.4	Initialisation Considerations . . . . .	45
2.3.5	Division . . . . .	47
2.3.6	Compositionality . . . . .	49
2.4	Applications of the NALU . . . . .	49
2.4.1	Existing Applications . . . . .	49
2.4.2	Applications Where NALU Is Inferior . . . . .	51
2.5	Discussion: Remaining Gaps . . . . .	52
<b>3</b>	<b>Benchmarking Existing and Future Models</b>	<b>55</b>
3.1	Two Layer Arithmetic Task . . . . .	56
3.2	Evaluation Metrics . . . . .	57
3.2.1	Evaluation metrics used on the Arithmetic Dataset Task . . . . .	58
3.3	Single Module Arithmetic Task . . . . .	59
3.3.1	Evaluation Metrics . . . . .	60
3.3.1.1	Alternative Options for Generating a Success Threshold . . . . .	61
3.3.2	Results . . . . .	62
3.4	Summary . . . . .	66
<b>4</b>	<b>Multiplication - Improving Robustness via Stochasticity</b>	<b>67</b>
4.1	Robustness Issues with Multiplication Modules . . . . .	68
4.1.1	Problem: Inputs that Induce Local Optima . . . . .	68
4.2	A Stochastic Wrapper: The Stochastic NMU (sNMU) . . . . .	69
4.3	Alternate Stochastic Methods . . . . .	71
4.3.1	Stochastic Gating . . . . .	72
4.3.2	Gradient Noise . . . . .	72
4.4	Single Layer Task . . . . .	72
4.5	Arithmetic Dataset Task . . . . .	73
4.6	MNIST Arithmetic . . . . .	76
4.6.1	Static MNIST Product . . . . .	76
4.6.1.1	Isolated Digit Classification . . . . .	77
4.6.1.2	Colour Channel Concatenated Digit Classification . . . . .	79
4.6.2	Sequential MNIST Product . . . . .	82
4.7	Summary . . . . .	84
<b>5</b>	<b>Division - Understanding the Underlying Learning Mechanisms</b>	<b>87</b>
5.1	Related Work . . . . .	88
5.2	Architectures . . . . .	88
5.2.1	NRU . . . . .	89
5.2.2	NMRU . . . . .	90
5.3	Single Module Arithmetic Experiment Setup . . . . .	91
5.4	Improving the Real NPU's Robustness . . . . .	92
5.5	Uniform Range Datasets . . . . .	95
5.6	Mixed-Sign Input Datasets . . . . .	96
5.7	More Challenging Distributions: Larger Magnitudes and Mixed-Signs . . . . .	98



5.8	Division by Small Magnitudes . . . . .	100
5.8.1	Impact of the Singularity Issue on Gold Solutions . . . . .	100
5.8.2	Experimental Results . . . . .	100
5.9	Traits of Modules when Learning on the Redundancy Setting . . . . .	102
5.10	MNIST Arithmetic - Isolated Digit Classification . . . . .	103
5.10.1	Setup and Network Architecture . . . . .	104
5.10.2	Metrics and Results . . . . .	104
5.11	Discussion . . . . .	105
5.12	Summary . . . . .	107
<b>6</b>	<b>Factors to Consider when Learning NALMs</b>	<b>109</b>
6.1	Feature Scaling . . . . .	109
6.2	Uninformative MSE Loss . . . . .	113
6.3	Alternate Losses: PCC and MAPE . . . . .	115
6.3.1	Arithmetic Dataset Task . . . . .	116
6.3.2	Product of Sequential MNIST . . . . .	117
6.3.3	Division: Different Losses on the Single Module Task (with Redundancy) . . . . .	118
6.3.4	Summary . . . . .	119
<b>7</b>	<b>Compositionality - Learning Multi-Step Operations</b>	<b>121</b>
7.1	Task . . . . .	122
7.2	Methods . . . . .	124
7.2.1	MLP . . . . .	124
7.2.2	Quadratic Network . . . . .	124
7.2.3	Stacked NALMs . . . . .	124
7.2.4	Stacked Gated NALMs . . . . .	126
7.2.5	Recurrent Input Selector with Learnable NALMs . . . . .	126
7.2.6	Recurrent Input Selector with Frozen NALMs . . . . .	128
7.3	Results . . . . .	130
7.4	Summary . . . . .	133
<b>8</b>	<b>Conclusions</b>	<b>135</b>
8.1	Directions for Future Work . . . . .	138
8.1.1	Input and Module Selection for Compositional NALMs . . . . .	138
8.1.2	Learning Coefficients . . . . .	138
8.1.3	Alternative Encoding of Numbers . . . . .	139
8.1.4	Extension of the Evaluation Suite . . . . .	140
	<b>Appendix A Inductive Biases for System 2</b>	<b>141</b>
	<b>Appendix B Representation of Numbers in Humans, Animals and Computers</b>	<b>145</b>
	Appendix B.1 Humans . . . . .	145
	Appendix B.1.1 How do Humans Process Numbers? . . . . .	146
	Appendix B.2 Animals . . . . .	147
	Appendix B.3 Computers . . . . .	147
	<b>Appendix C Additional NALM Background Information</b>	<b>149</b>

Appendix C.1	Module Illustrations . . . . .	149
Appendix C.2	Step-by-step Example using the NALU . . . . .	152
Appendix C.3	Naive NPU Derivation . . . . .	154
<b>Appendix D</b>	<b>NALM Benchmarking - Comparisons of Existing Works</b>	<b>157</b>
Appendix D.1	Additional Experiments . . . . .	157
Appendix D.2	Cross Module Comparison . . . . .	158
Appendix D.3	Experiments and Findings of Modules for Logic Tasks . . . . .	159
Appendix D.3.1	NLRL . . . . .	159
Appendix D.3.2	NSR . . . . .	160
<b>Appendix E</b>	<b>Experiment Details</b>	<b>161</b>
Appendix E.1	Benchmark Synthetic Arithmetic Tasks . . . . .	161
Appendix E.1.1	Experiment Parameters . . . . .	161
Appendix E.1.2	Hardware and Runtimes . . . . .	162
Appendix E.2	Multiplication MNIST Experiments . . . . .	163
Appendix E.2.1	Experiment Parameters . . . . .	163
Appendix E.2.2	Hardware and Runtimes . . . . .	163
Appendix E.3	Division Experiments . . . . .	164
Appendix E.3.1	Parameter Initialisation . . . . .	164
Appendix E.3.2	Hardware and Runtimes . . . . .	164
Appendix E.3.3	Summary Table of the Ranges Used for the Single Layer Task . . . . .	166
Appendix E.4	MNIST Product Tasks: Architecture Details . . . . .	168
Appendix E.4.1	Isolated Digits . . . . .	168
Appendix E.4.2	Colour Channel Concatenated Digits . . . . .	168
Appendix E.4.3	Sequential MNIST . . . . .	170
<b>Appendix F</b>	<b>Multiplication: Static MNIST Analysis</b>	<b>171</b>
Appendix F.1	Class Accuracies . . . . .	171
Appendix F.2	Isolated Digits . . . . .	171
Appendix F.3	Colour Channel Concatenated Digits. . . . .	171
Appendix F.4	Digit Classification Accuracy over Epochs . . . . .	173
<b>Appendix G</b>	<b>Division: Additional Analysis</b>	<b>175</b>
Appendix G.1	Properties of a Division Module . . . . .	175
Appendix G.2	NRU; Single Module Task (without Redundancy): Tanh Scale Factor . . . . .	178
Appendix G.3	Real NPU; Single Module Task (without Redundancy) . . . . .	179
Appendix G.4	NRU; the Single Module Task (without Redundancy): Effect of Learning Rate . . . . .	180
Appendix G.5	Real NPU; Single Module Task (with Redundancy) . . . . .	181
Appendix G.6	NMRU; Single Module Task (with Redundancy): Additional Experiments . . . . .	182
Appendix G.7	NRU; Single Module Task (with Redundancy): Calculating the Sign Separately . . . . .	184
Appendix G.8	Division MNIST Arithmetic Task: Effect of Gradient Norm Clip	185

---

<b>Appendix H Gradients of the Arithmetic Dataset Task</b>	<b>187</b>
Appendix H.1 MSE Loss for the Arithmetic Dataset Task . . . . .	187
Appendix H.2 Explicit Gradients . . . . .	188
Appendix H.2.1 MSE Loss Partial Derivatives: . . . . .	189
Appendix H.3 Generalised NAU and NMU Partial Derivatives of the loss for a NAU-NMU . . . . .	190
Appendix H.4 Generalised NAU and NMU Partial Derivatives for a NAU- sNMU . . . . .	190
Appendix H.4.1 MSE Loss Definition . . . . .	191
Appendix H.4.2 Loss derivatives wrt NAU and sNMU weights . . . . .	191
<b>References</b>	<b>193</b>



## Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as: [Mistry et al. \(2022a\)](#), [Mistry et al. \(2022b\)](#), [Mistry et al. \(2022c\)](#)

Signed:.....

Date:.....



## Acknowledgements

To truly write my acknowledgements to its fullest would result in a significant increase in the length of this thesis. Therefore, I opt to take the route of having a short paragraph and instead allow my future actions to express my gratitude to those who have helped me on this marathon of a journey.

First and foremost, I want to thank my supervisors Kate Farrahi and Jon Hare for their continual support and guidance through this PhD, without which I would not have made it very far. Thank you both, for giving me this opportunity. Secondly, I want to thank the VLC research group members for all their insightful discussions and wonderful company over the years. It is an excellent working environment. Thirdly, thank you to all my friends and family for helping me when I needed that much needed step away from the work! I would also like to thank the EPSRC for its support in funding this research and the IRIDIS High-Performance Computing Facility, the ECS Alpha Cluster, and associated support services at the University of Southampton in the completion of this work. And finally, above all, thank you, Mum, Dad and Ma for your boundless love, support and blessings. Needless to say, I would have never have got to this stage without you.





# Definitions and Abbreviations

## Network architectures

CNN	Convolutional Neural Network.
DNN	Deep Neural Network.
FCNN	Fully Connected Neural Network.
GRU	Gated Recurrent Unit.
LSTM	Long Short Term Memory.
MLP	Multi-Layer Perception.
NALM	Neural Arithmetic Logic Module.
NPS	Neural Production System.
PU	Product Unit.
RIM	Recurrent Independent Mechanism.
RNN	Recurrent Neural Network.
STN	Spatial Transformer Network.

## NALMs

G-NALU	Golden - Neural Arithmetic Logic Unit.
iNALU	Improved Neural Arithmetic Logic Unit.
NAC <sub>+</sub>	Summative part of the Neural Arithmetic Logic Unit.
NAC <sub>•</sub>	Multiplicative part of the Neural Arithmetic Logic Unit.
NALU	Neural Arithmetic Logic Unit.
NAU	Neural Addition Unit.
NLRL	Neural Logic Rule Layer.
NMRU	Neural Multiplicative Reciprocal Unit.
NMU	Neural Multiplication Unit.
NPU	Neural Power Unit.
NRU	Neural Reciprocal Logic.
NSR	Neural Status Register.
RealNPU	Real Neural Power Unit.
sNMU	Stochastic Neural Multiplicative Unit.
stgNMU	Stochastic gated Neural Multiplicative Unit.

**Terms**

AI	Artificial Intelligence.
AUC	Area Under the Curve.
ANS	Approximate Number System.
CDN	Content-Delivery-Network.
CLS	Crowdsourced-Live-Streaming.
CoT	Chain of Thought.
EA	Evolutionary Algorithms.
GCD	Greatest Common Divisor.
GNW	Global Neuronal Workspace.
GP	Genetic Programming.
GPU	Graphic Processing Unit.
GWT	Global Workspace Theory.
LM	Language Model.
ML	Machine Learning.
MNIST	Handwritten digit dataset ( $28 \times 28$ pixel, greyscale) split into training (60K) and test (10K) sets.
<b>msv</b>	Mixed Sign Vector.
NLP	Natural Language Processing.
OOD	Out-of-Distribution.
OTS	Object Tracking System.
QoS	Quality of Service.
RL	Reinforcement Learning.
TPS	Thin Plate Spline.
REINFORCE	<b>REward Increment = Non-negative Factor</b> $\times$ <b>Offset Reinforcement</b> $\times$ <b>Characteristic Eligibility</b> . A class of reinforcement learning algorithms that falls under Policy Gradient methods.
RHS	Restrict Hypothesis Space.
SCOFF	SCHEMA/Object-File Factorisation.

**Losses**

MSE	Mean Squared Error.
MAPE	Mean Absolute Precision Error.
PCC	Pearson's Correlation Coefficient.
PCC-MSE	Pearson's Correlation Coefficient - Mean Squared Error. Switches from using a PCC loss to a MSE loss after a predefined number of training iterations.

# Chapter 1

## Introduction

Human-level systematic reasoning remains to be one of the holy grails of Artificial Intelligence (AI). Humans are able to take a problem, understand it, decompose it, and then plan and execute a solution. We can even learn tasks that the brain is not evolved to do (e.g., budgeting weekly expenditures). Our reasoning enables us to adapt to new situations in a timely manner despite the limited capacity of our brain. In Machine Learning (ML), this adaptation is known as out-of-domain generalisation. For AI practitioners the big question remains, what process(es) do we use for reasoning and how can we encode such a process into our learning models? However, this question is incredibly vague and broad, therefore, we instead look at answering a smaller, more well-defined question which will allow us to come a step closer to solving the question of human reasoning in machines.

In this thesis, we ask:

*How can we learn to discover basic mathematics using ML models in a generalisable manner?*

Specifically, this thesis is about understanding how to develop neural networks to do extrapolative out-of-distribution (OOD) arithmetic by using interpretable specialist modules which we term Neural Arithmetic Logic Modules (NALMs).

The aim of this chapter is to provide the reader with an intuition into the reasoning behind why this particular topic was chosen as the focal point. We begin at the broadest level, considering how humans reason and some of the ingredients required to build such neural networks. Then, we introduce mathematical reasoning in neural networks along with a case study using Multilayer Perceptrons (MLPs) to promote the need for focusing on arithmetic and the use of specialists. We identify some of the different ways to represent numbers in ML models and end by introducing symbolic regression.

## 1.1 High-level Reasoning in Humans

From the cognitive science perspective, one answer to the question *'what process(es) do we use for reasoning'* is the existence of habitual and controlled processing (Botvinick et al., 2001) representing the behaviours which are automatic and those which require mental effort (e.g., attention). This dual-process model was introduced by Posner and Snyder (2004), but the idea of having two ways of thinking can be traced back to much earlier (James, 1890). A similar line of thought comes from the System 1 and System 2 thinking processes, popularised by Kahneman (2011). System 1 encapsulates the type of thinking which is instantaneous, intuitive and requires little to no effort. System 2 refers to the logical side of the thinking process that is slower and requires effort. It is pieced together by logical judgement and a mental search for additional information acquired through past learning and experience. For example, consider the task of counting the number of dots on a page. System 1 is used when there are a small number of dots (1 to  $\sim 3$ ), where we use subitizing to instantly know the number of dots without having to count; if there are more than three dots, then our System 2 thinking is used to explicitly count the number of dots (Dehaene, 1992). The two systems are not independent, rather we combine systems in a complementary manner. For example, System 2 gets called when System 1 fails to reach an answer. Peters et al. (2006) experiment and compare participants with high and low numeracy skills on tasks requiring number processing. Participants who were more numerate were found to use their System 1 reasoning more frequently and reliably but also call upon their System 2 when tasked with more complex challenges.

There are two ways of processing problems depending on their difficulty. Simpler problems are solved simply via retrieval of the solution stored in our long-term memory (Ashcraft, 1992) like in System 1 thinking. Complex problems are solved in a procedural fashion, solving the problem in multiple steps (Van Beek et al., 2014). With practice, we can switch operations which once required procedural processing into retrieval, e.g., imagine a chess master against a novice; the novice would intensely ponder about the first counter move while a chess master can respond instantly.

Current ML systems using Deep Learning (DL) work well in solving System 1 tasks such as image classification (e.g. Szegedy et al., 2017) and object detection (e.g. Redmon et al., 2016), however the ability to do the System 2 reasoning which enables systematic generalisation (performing well on new situations in a methodological manner) and fast learning (applying new rules without practice) remains desirable (Goyal and Bengio, 2022). One may now ask *how can reasoning process(es) be encoded into our learning models?* At this point, let us assume the process of reasoning refers to System 2 thinking. But what does 'encode' refer to? In this case, encoding is considered as incorporating some form of favorability towards a particular solution for a model. This is more commonly known as an inductive bias (IB).

## 1.2 Inductive Biases

One of the most accepted definitions of an inductive bias is defined by Mitchell (1980) “to refer to any basis for choosing one generalization over another, other than strict consistency with the observed training instances”. This definition for bias was defined with respect to achieving the *inductive leap* required for generalisation. The term inductive leap stems from the philosophy of logic, referring to going from sample observations to a general conclusion. From logical reasoning, induction is also thought of as a primitive with the ability to induce general rules of inference from known facts (Peirce, 1992). In ML, an inductive leap is more commonly known as inductive or concept learning (Michalski, 1983; Utgoff, 1986). More recently, Battaglia et al. (2018) defines IB as allowing “a learning algorithm to prioritize one solution (or interpretation) over another, independent of the observed data.” Put simply, a *bias* means there exists a preference and *inductive* means the preference (bias) is towards how the learner prioritises a particular solution (from only observing samples).

A learner with no bias is called an unbiased learner or inductive system. An unbiased generaliser “makes no a priori assumptions about which classes of instances are most likely, but bases all its choices on the observed data” Mitchell (1980). However, unless all possible examples are observed, the inductive learning algorithm will not be able to find a generalisable unique solution from the data alone, making inductive learning an ill-posed problem. Such unbiased learners can only generate rules from the given samples alone so at best can memorise the observed samples (training data), meaning they will be unable to generalise to unseen data. From a perspective of loss landscapes, no bias results in a learner having preferences to the local minima of the loss surface leading to an unrobust solution easily affected by randomness from factors such as initialisation or the order of training data (Sutskever et al., 2013; Abnar et al., 2020; McCoy et al., 2020; Dodge et al., 2020). Hence, having no bias is bad as it results in a learner who memorises. But the opposite, having too strong a bias, results in a niche learner that cannot generalise to different problems. This ties in nicely with the no-free-lunch theorem (Wolpert and Macready, 1997) which states there is no universal one-fits-all learner that can outperform all other learners on every task. For generalisation to occur for a specific task, IBs need to be utilised to restrict the hypothesis space (Craven, 1996). Therefore, biases are included to reduce the hypothesis search space but too high a bias can lead to sub-optimal solutions.

At the crux of it, two types of IBs exist: the *restrict hypothesis space (RHS) biases* and the *preference biases* (Craven, 1996). The former determines the expressivity of the hypothesis space and the latter determines the preference between solutions in that space (i.e., how a learner will traverse the space). Other names used in the literature include representational bias (Gordon and Desjardins, 1995) for RHS and procedural/algorithmic bias (Rendell, 1987) for the preference bias. In both cases, we want to incorporate a bias

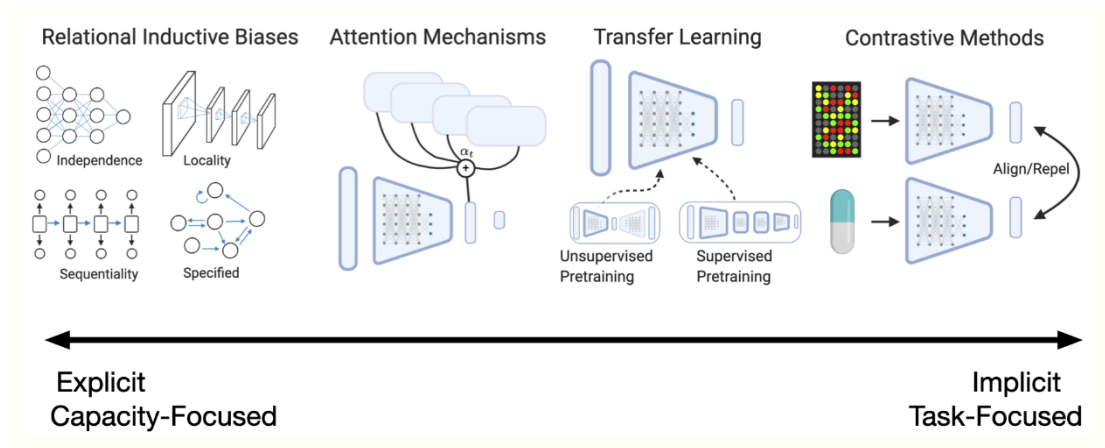


FIGURE 1.1: Example of the two types of IBs: RHS (explicit) biases and preference (implicit) biases. Image sourced from [https://sgfin.github.io/assets/thesis\\_images/knowledge\\_paradigms.png](https://sgfin.github.io/assets/thesis_images/knowledge_paradigms.png) from the post <https://sgfin.github.io/2020/06/22/Induction-Intro/>.

which enables some form of partiality toward the learnt solution. But where does this bias come from? The answer is our prior knowledge. In other words, we use biases to incorporate prior knowledge. The type of priors and the way we incorporate them is where the two categories differ (as illustrated in Figure 1.1). RHS biases are a form of *explicit biases* which are properties built into the model architecture. Such structural priors are a form of *relational biases* which are “constraints on relationships and interactions among entities in a learning process” (Battaglia et al., 2018). The type of *entity* represents an element with attributes and a property which links entities is called a *relation*.<sup>1</sup> For example:

- Convolutional Neural Networks (CNNs) (LeCun et al., 1998) have an IB for locality, referring to only applying filters to entities that are close to each other (i.e., elements in a patch);
- Recurrent Neural Networks (RNNs) (Elman, 1990; Jordan, 1997) have an IB for sequentiality, which is a result of the dependencies between the current hidden state, previous hidden state and current input entities; relations between these entities can be described as Markovian dependencies; and,
- Fully Connected Neural Networks (FCNN) (Rumelhart et al., 1986) have weak IB (of independence) because all entities (units) can interact with each other.

In contrast, preference biases are a form of *implicit biases* which are built into the task (rather than the model) and require learning. For example, the Transfer Learning

<sup>1</sup>For many of the well known DL architecture families, the structural properties can be viewed as encoding certain invariances/equivariances to certain transformations but this is outside the scope of this thesis.

paradigm shares knowledge gained from a previous task, while the Contrastive Learning paradigm learns representations by forcing similar samples together while pushing dissimilar data apart.

A bias does not need to be static and determined at the start of learning; it can be dynamic and change during learning. This is known as a bias shift (Gordon and Desjardins, 1995), where the selection of the bias occurs after training begins and can be viewed as searching through the bias space. Two examples of *dynamic biases* are the alignment of attention mechanisms (Vaswani et al., 2017) and dynamic routing of capsules in Capsule networks (Sabour et al., 2017). In both cases, the preference for solutions is learned and dependent on querying the context (Finlayson, 2020, pp. 7-20).

The type of networks we will consider (NALMs) have explicit biases and can have implicit biases. The architectures are designed to be able to model exact operations such as arithmetic expressions, which constrains the hypothesis space of available solutions. NALMs assume that particular (transformed) weight values correlate to applying exact operations on selected inputs. To induce such weights during learning, some works use a specialised regularisation (discussed in Chapter 2), which can be thought of as a type of implicit bias.

### 1.3 Inductive Biases for System 2

IBs ultimately allow for control over the possible solution search space and the traversal of the learner in the space. If we consider the logical tasks requiring System 2 processing, what types of IBs would be relevant?

One answer takes inspiration from human cognition. Specifically, the Global Workspace Theory (GWT) proposed by Baars (1993, 1997) is a widely accepted neuroscientific theory of consciousness (Michel et al., 2018), used in Chapter 7 to inspire our architecture for compositionality. The basic global workspace model from Baars (1993) is illustrated in Figure 1.2. The GWT proposes that high-level conscious processing is accomplished by allowing selected parts of the brain (via top-down attentional amplification) to update a shared communication representation (called a blackboard or workspace) which gets broadcasted to the entire brain. The parts which get selected are task-dependent and can be thought of as specialists. As only a selected few can write to the workspace, it can be considered a communication bottleneck. The use of top-down attention is similar to the biased competition theory of selective attention where areas of the brain representing visual information undergo competition to allocate resources (Desimone et al., 1995). Such competition is required due to our brains having limited processing capacity and therefore requiring a form of selectivity to filter out irrelevant information.

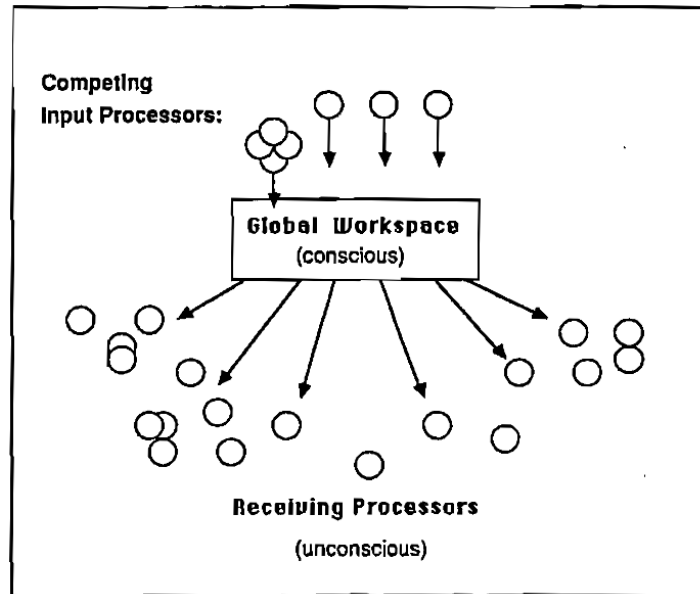


FIGURE 1.2: Example of the GW model reproduced from Baars (1993, Figure 2.3). Each circle represents a specialised (unconscious) processor. For these specialists to interact and coordinate requires exchanging information at a central point (the ‘global workspace’) only accessible through competition. This example shows four specialists accessing the workspace, where the resulting message from these specialists is broadcasted to the whole system.

The GWT was extended by Dehaene (2014) to the Global Neuronal Workspace (GNW) model, which considers the GW working with neurons. In this case, the workspace neurons connect to the (unconscious) modules by long-distance pathways (connecting prefrontal and parietal cortices) to obtain global availability (Prakash et al., 2008). When the broadcast excitation exceeds a threshold, it creates a large-scale activity pattern in the brain which is termed global ignition.

From considering the GWT, the following components need to be captured:

1. Specialists which can contribute different pieces of information to the workspace.
2. Competition between specialists for the sparse selection of relevant information.
3. Some form of global shared representation to encourage coherent knowledge.

In particular, extending the first point, we would ideally want these specialists to be independent of each other in two ways. The first is independence in the function they perform and the second is independence from each other such that no individual can inform or influence another (i.e., independent causal mechanisms (Schölkopf et al., 2012; Schölkopf et al., 2021)). Throughout this work, we will assume independence amongst the specialists. However, it is worth mentioning the possible role of redundancy. Although independence allows for separation and concise formulation, it also results in fewer ways to obtain the solution compared to an overparameterised



architecture. In contrast, introducing redundancy between specialists may improve searching the solution space. For example, imagine having a network with multiple specialists of the same type. If each specialist represented an arithmetic operation, a network with redundancy could be one which contains multiple specialists which all can do addition. As each specialist has the same expressive power, the task of addition could be split over all the specialists rather than just requiring a single specialist. Once the network has finished training it is possible to simplify the network by removing redundant units/specialists.

The specific inductive biases for System 2 include specialised mechanisms (modules), independence of mechanisms, sparsity in using the mechanisms and reuse and composition of existing knowledge for novel situations. Encoding such biases into neural networks has displayed gains in OOD performance and therefore in extrapolation ability and generalisation (Goyal et al., 2021c,b,a; Liu et al., 2021; Goyal et al., 2022).<sup>2</sup>

Although modularity can provide better systematic generalisation than monolithic architectures (Bahdanau et al., 2019), modular networks can still exhibit problems regarding module under-utilisation and lack of specialisation (Mittal et al., 2022a). Furthermore, upon inspecting pretrained generic neural architectures (such as RNNs, Transformers, FCNNs and CNNs) in their ability to specialise to and reuse modules, Csordás et al. (2021) find that module’s specialisation to learn different operations is possible to an extent, but reuse of the specialist in a composable manner is not. Therefore, a bias towards modularity does not imply the ability to specialise or compose as a human might.

In contrast to networks which have biases towards modularity with no constraints on the type of specialisation of the module, NALMs are designed to be naturally modular networks where modules can specialise to one or more operations. This specialist nature allows one to know the exact representational power of the network.

## 1.4 A Stepping Stone towards Human Reasoning: Learning Mathematics

To teach machines to reason on any task could be considered too large of a step. On top of this, reasoning and solving problems in a human-interpretable manner is an additional challenge. Therefore, in this thesis, we opt to focus on a single type of reasoning, namely, mathematical reasoning. Such reasoning is unique to human intelligence and is essential for scientific discoveries and progress. Mathematical reasoning relies on having well-defined rules and patterns and can be composed of a sequence of unambiguous concise steps. In contrast, domains such as natural language reasoning are

---

<sup>2</sup>A deeper review of such networks can be found in Appendix A.

more unstructured requiring a complex understanding of context due to subtleties such as word meaning, syntax and semantics. For a human to learn mathematics requires years of education and is incremental and compositional in learning the skills. Skills are taught in a curriculum learning fashion, where problems are ordered by difficulty, beginning with simple problems and gradually increase in difficulty. Hence, learning reliable mathematical mechanisms can provide a stepping stone toward general AI, closing the reasoning gap between humans and machines. Furthermore, the mathematical domain allows one to easily create high-quality synthetic tasks with controlled difficulty, making it easier to discover where challenges lie for models. For example, being able to associate learning difficulties with particular arithmetic operations, training ranges or compositions.

Throughout this thesis, we focus on specialist neural network modules (NALMs) that are designed to work directly on the numerical inputs rather than encodings/embeddings. To motivate why we focus on specialist neural networks over the other extreme (monolithic networks) we show how universal approximators networks (i.e., MLPs) fail to learn basic arithmetic. We also discuss the shortcomings of Transformer based Language Models (LMs), a common architecture for mathematical reasoning tasks, in learning generalisable mathematics. After which, we discuss the different types of input representations to motivate why the specialists neural networks in this thesis will not be relying on learnable embeddings or alternate forms of static encodings.

### 1.4.1 Motivating Specialist Modules over Generic MLPs

To begin, let us consider why we may want to use specialist modules over a more generic neural network such as an MLP. To demonstrate this, consider the MLP in the context of universal approximators. MLPs have high representational power enabling them (in theory) to be universal approximators. More specifically, the universal approximation theorem states that feedforward networks with at least one hidden layer using nonpolynomial activation functions, sufficient width, and a linear output layer can approximate any Borel measurable function on a compact set up to an arbitrary degree of precision (i.e., any nonzero amount of error) (Hornik et al., 1989; Leshno et al., 1993). In other words, assuming a wide enough MLP, a network will have enough representational power to approximate any continuous function up to a nonzero precision.

However, there lie three disadvantages. Firstly, at best, you only learn an approximator of the true function (Nielsen, 2015), meaning that performance on data outside the training range may be poor; an approximation implies there exists no bias towards learnt parameters being interpretable. Secondly, though bounds on the architecture size can exist (Park et al., 2021) for different function classes, an optimal architecture for a problem remains unknown. Thirdly, the ability to represent a function does not imply the ability to learn the function in an empirical setting. ReLU MLPs are found

to converge to linear functions along any direction from the origin outside the training data range, meaning that such networks cannot extrapolate to most non-linear functions (Xu et al., 2021). In other words, MLPs *linearize* to the OOD data. This holds for different network depths, widths, learning rates and batch sizes (Xu et al., 2021, Appendix C.1 and C.2). In contrast, as specialist modules have the capacity to model an exact function (rather than an approximation) they can extrapolate well.

To demonstrate the limitations of using MLPs for learning operations such as those in arithmetic, we setup a simple arithmetic task to learn a single arithmetic operation.

**Setup:** Given two inputs  $x_1$  and  $x_2$ , output the value for  $x_1 \circ x_2$ , where  $\circ \in \{+, -, \times, \div\}$ . Networks are trained on a range  $\mathcal{U}[1,2)$ , tested on an extrapolation range  $\mathcal{U}[2,6)$  and run over 25 seeds. A single hidden layer MLP network with a width of either 1 or 100 is learnt to check both extremes. Width 1 networks are trained for 50,000 epochs and width 100 are trained for 2,000,000 epochs. A ReLU activation is used and we keep bias terms to avoid constraining expressiveness. For a network to be considered interpolative/extrapolative, the errors must be below a minimum loss threshold, determined by the range of the data; for more details, see the Single Module Arithmetic Task in Section 3.3.

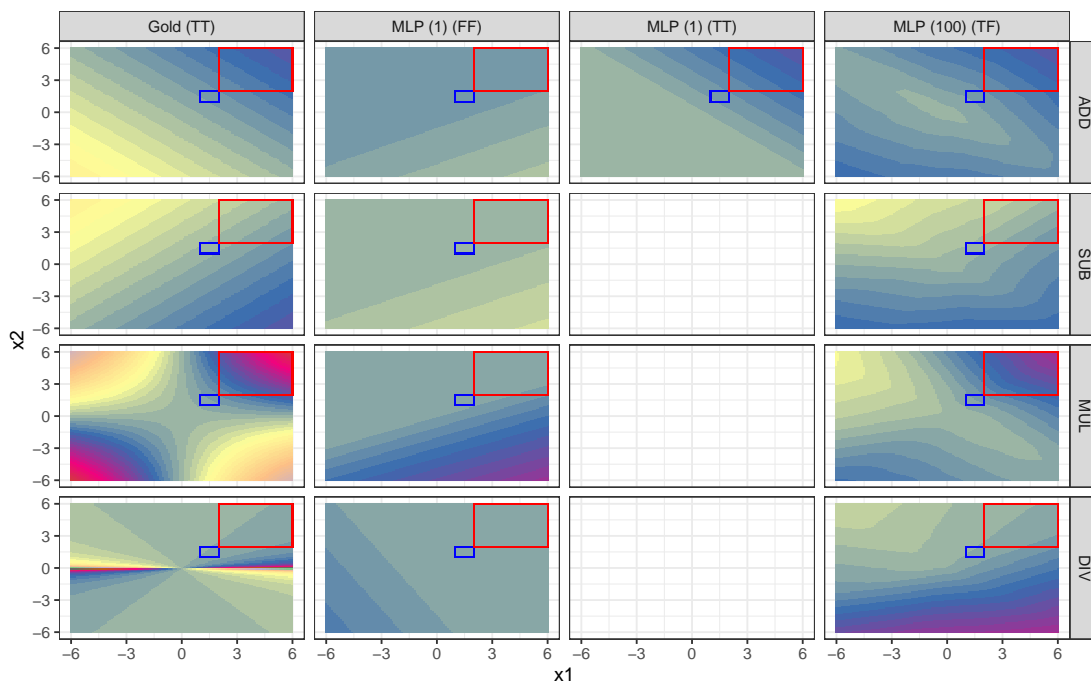


FIGURE 1.3: Surface plots for the four arithmetic operations, comparing the golden solution (left column) to learnt MLPs of either width 1 or 100. The letters in the brackets are True (T)/False (F), representing if the minimum loss threshold for the interpolation and extrapolation range have been met respectively. The blue and red squares represent the interpolation (training) and extrapolation (test) ranges respectively.

### 1.4.1.1 Results

Surface plots of the trained MLPs are given in Figure 1.3. The plots are compared to the gold solution which represents the true function that should be modelled. The division plots have a white line at  $x_2 = 0$  as division by 0 was omitted in the data collection stage. The plots are drawn over an input domain of  $[-6,6]$ , which is an extension of the extrapolation range, in order to see if the true underlying function has been learnt.

For each MLP width, there are four possible outcomes representing whether the interpolation and extrapolation ranges are modelled adequately. For a width 1 network, except for addition, there are no other cases of achieving success on the extrapolation range and for width 100 no operation is able to achieve success on the extrapolation range. However, for the one case where the extrapolation range is considered learnt (i.e., MLP (1)(TT) for ADD) the remaining portion of the surface plot which does not include the interpolation and extrapolation ranges has not been learnt correctly. For this case in particular, the MLP learns to model Figure 1.4, which represents

$$\hat{y} = \text{relu}(x_1 w + x_2 w - b) \cdot \frac{1}{w} + \frac{b}{w},$$

where  $w$  can be any positive value and  $b$  can be any value. The resulting expression learns to do addition, but only holds for positive inputs.

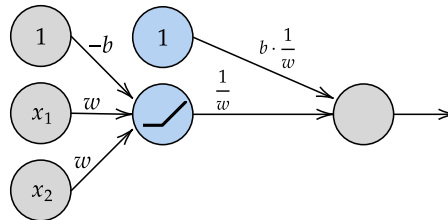


FIGURE 1.4: Example of a single hidden layer MLP with width one that can add two positive numbers.

Unlike multiplication and division, which can at best be approximated by an MLP, addition/subtraction can be learned with an MLP of width four. Rather, there exists an MLP with the capacity to either add or subtract any two floating-point numbers. Figure 1.5 presents such an MLP which consists of a two-layer neural network with one hidden layer of width four using ReLU activations. The resulting expression for addition is  $x_1 + x_2 = \text{relu}(x_1) - \text{relu}(-x_1) + \text{relu}(x_2) - \text{relu}(-x_2)$  and subtraction is  $x_1 - x_2 = \text{relu}(x_1) - \text{relu}(-x_1) - \text{relu}(x_2) + \text{relu}(-x_2)$ . These two expressions will hold for inputs sampled from any number (both positive and negative) on the real

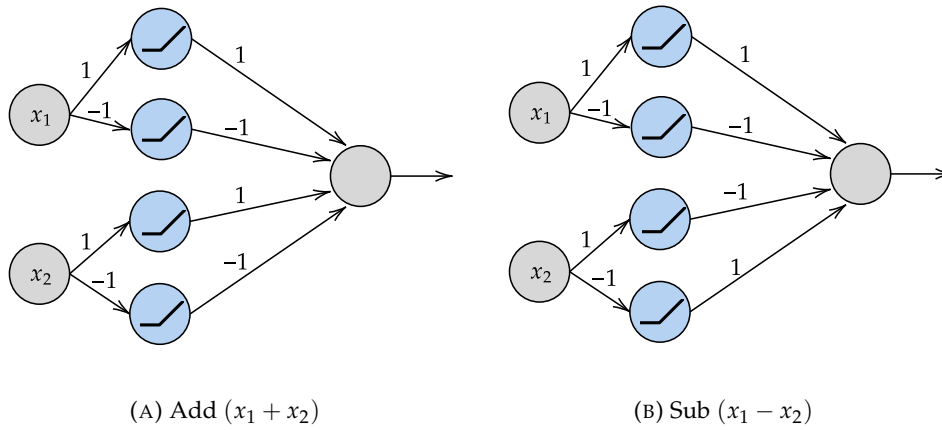


FIGURE 1.5: MLPs with 1 hidden layer and 4 hidden units which learn extrapolative addition (left) and subtraction (right). For clarity, only non-0 connections are shown.

domain. The resulting parameters for addition and subtraction respectively are

$$\begin{bmatrix} b_{1,0}^{(1)} & b_{2,0}^{(1)} & b_{3,0}^{(1)} & b_{4,0}^{(1)} \\ w_{1,1}^{(1)} & w_{2,1}^{(1)} & w_{3,1}^{(1)} & w_{4,1}^{(1)} \\ w_{1,2}^{(1)} & w_{2,2}^{(1)} & w_{3,2}^{(1)} & w_{4,2}^{(1)} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} b_{1,0}^{(2)} & w_{1,1}^{(2)} & w_{1,2}^{(2)} & w_{1,3}^{(2)} & w_{1,4}^{(2)} \end{bmatrix} = \begin{bmatrix} 0 & 1 & -1 & 1 & -1 \end{bmatrix}$$

and

$$\begin{bmatrix} b_{1,0}^{(1)} & b_{2,0}^{(1)} & b_{3,0}^{(1)} & b_{4,0}^{(1)} \\ w_{1,1}^{(1)} & w_{2,1}^{(1)} & w_{3,1}^{(1)} & w_{4,1}^{(1)} \\ w_{1,2}^{(1)} & w_{2,2}^{(1)} & w_{3,2}^{(1)} & w_{4,2}^{(1)} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} b_{1,0}^{(2)} & w_{1,1}^{(2)} & w_{1,2}^{(2)} & w_{1,3}^{(2)} & w_{1,4}^{(2)} \end{bmatrix} = \begin{bmatrix} 0 & 1 & -1 & -1 & 1 \end{bmatrix} .$$

Consider interpreting what the paths through each of the hidden units from Figure 1.5 will represent. The first and third hidden units will preserve the magnitude of  $x_1$  and  $x_2$  for positive values. The second and fourth hidden units will preserve the magnitude of  $x_1$  and  $x_2$  for negative values. The hidden-to-output layer weights will combine the magnitudes, reapplying the signs of negative inputs and applying signs for the subtraction operation to subtract  $x_2$ .

However, achieving these extrapolative MLPs is challenging as shown by surface plots shown in Figure 1.6. It is worth noting that due to symmetry, other valid solutions can be found through permutation of the above parameter matrices.

Again, notice the similar issue to the MLPs of width 1 and 100 where an MLP can learn within the interpolation range (which it was trained on) and extrapolation range (which it never saw) but fails to learn the true function. Observing the learnt weights,

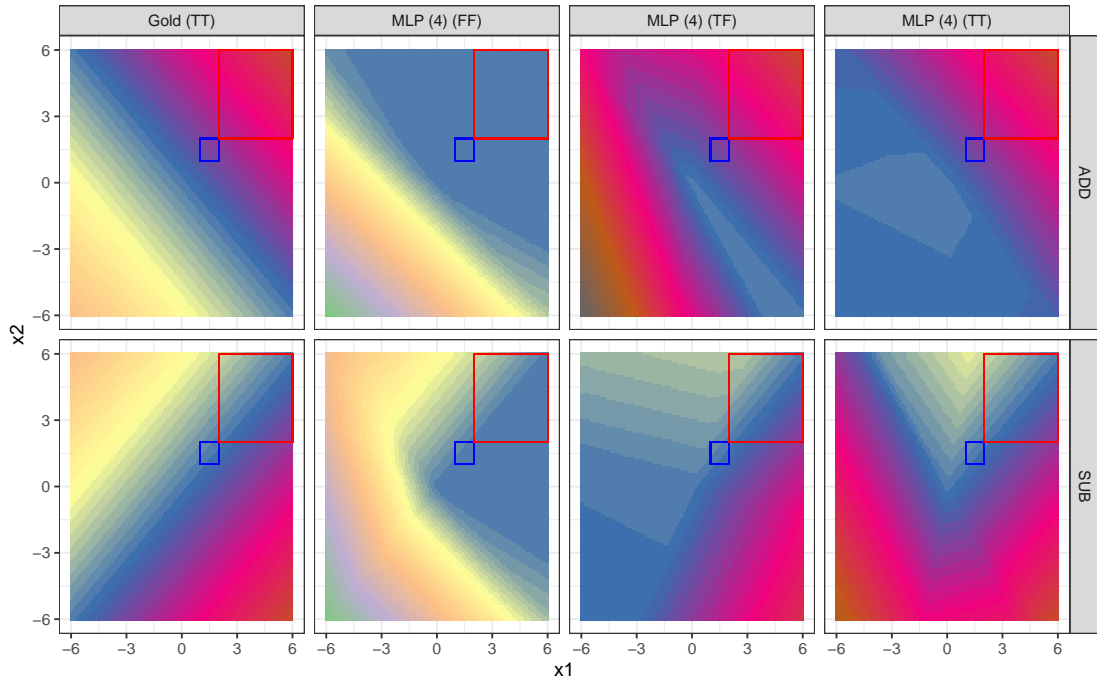


FIGURE 1.6: Surface plots for addition or subtraction, comparing the golden solution (left column) to learnt MLPs of width 4. The letters in the brackets are True (T)/False (F), representing if the minimum loss threshold for the interpolation and extrapolation range has been met respectively. The blue and red squares represent the interpolation (training) and extrapolation (test) ranges respectively.

we find the MLPs learn to use all hidden units and paths producing a complex expression which lacks interpretability.

To summarise, we have shown how generic neural networks, specifically MLPs, struggle to learn simple arithmetic operations in light of their expressive nature. Due to this, we will focus our efforts on investigating extrapolative specialist modules which are able to reproduce the gold solutions from the surface plots.

#### 1.4.2 Shortcomings in Mathematical Reasoning when using Transformers

One popular direction to explore solving mathematical tasks is via a Transformer based LM which represents problems as sequence-to-sequence translations. In other words, teaching models which can translate a sequence of tokens (problem) into another sequence of tokens (solution). Studies have suggested that such models can obtain high accuracy on tasks requiring complex arithmetic such as predicting the local stability of differential systems (Charton et al., 2021) or modular linear arithmetic used in lattice cryptography (Wenger et al., 2022; Li et al., 2023). Transformer LMs have been known to sufficiently improve performance by increasing scale (model parameters) (Brown et al., 2020; Thoppilan et al., 2022). However, the emergence of systematic reasoning cannot be obtained simply through training larger models. For example, Rae et al.

(2021) trains the LM named Gophen at different scales varying from millions to billions of parameters, with the largest model used containing 280 billion parameters.<sup>3</sup> They find that logical and mathematical reasoning sees less benefit from scaling. Hendrycks et al. (2021) explain in their study - “Accuracy also increases only modestly with model size: assuming a log-linear scaling trend, models would need around  $10^{35}$  parameters to achieve 40% accuracy on MATH<sup>4</sup>, which is impractical”.

Techniques	In-distribution	Out-of-distribution	Improves with scale
Fine-tune	✓✓	✗	✗
Prompting	✗	✗	✗
Fine-tune + Prompting	✓✓	✗	✗
Fine-tune + Scratchpad	✓✓	✗	✗
Prompting + Scratchpad	✓	✓	✓
Fine-tune + Prompting + Scratchpad	✓✓	✓✓*	✓✓

Table 1: Performance on length generalization tasks of three techniques that language models admit: (1) Finetuning, (2) Prompting (or in-context few-shot learning) and (3) Scratchpad (Chain-of-Thought reasoning). We find that each technique (and the combinations thereof) have different modes of failure and present different trade-offs regarding in and out-of-distribution coverage. ✗ signifies poor ✓ signifies nontrivial, ✓✓ signifies near-perfect performance. (\*) Refers to task-dependency.

FIGURE 1.7: Copy of Anil et al. (2022, Table 1) showing performance in length generalisation on the parity and variable assignment tasks, comparing fine-tuning, prompting and scratchpads.

Rather than scaling by training/finetuning larger models, alternate methods to encourage the emergence of reasoning are required (see Figure 1.7). These techniques include scratchpad training, chain of thought (CoT) reasoning, in-context learning and majority voting (Lewkowycz et al., 2022). *Scratchpads* require finetuning models to output the intermediate calculations, rather than the direct output (Nye et al., 2021). For example, having state tracking as the intermediary outputs for code execution questions. Using scratchpads allows a model to explicitly reference intermediary calculations rather than relying on internal representations of the calculations, encouraging better multi-step reasoning. *CoT reasoning* requires prompting the model to output an explanation on how to calculate the answer before giving the final output (Wei et al., 2022; Chowdhery et al., 2022). Unlike scratchpads which output execution steps, CoT reasoning will give a natural language explanation using sentences. It also better matches how humans use logical steps when solving tasks rather than having a single final output. *In-context* learning requires providing the model with a few ( $\sim 2 - 3$ ) examples of in-domain questions containing the expected outputs (sometimes with CoT reasoning) before prompting the desired question to answer (Radford et al., 2019; Brown et al., 2020). This technique enables skill acquisition without sacrificing model generalisability. Finally, the *majority voting* strategy (also known as self-consistency) is a decoding strategy in which multiple sequences of rationales and answers are given on which a voting scheme is used to select a final answer (Wang et al., 2023).

<sup>3</sup>Which is  $\times 1.6$  the size of GPT3 (Brown et al., 2020)

<sup>4</sup>The Mathematics Aptitude Test of Heuristics (MATH) dataset consists of 12,500 math problems used in competitions requiring strategy to be solved. Each problem comes with a step-by-step solution.

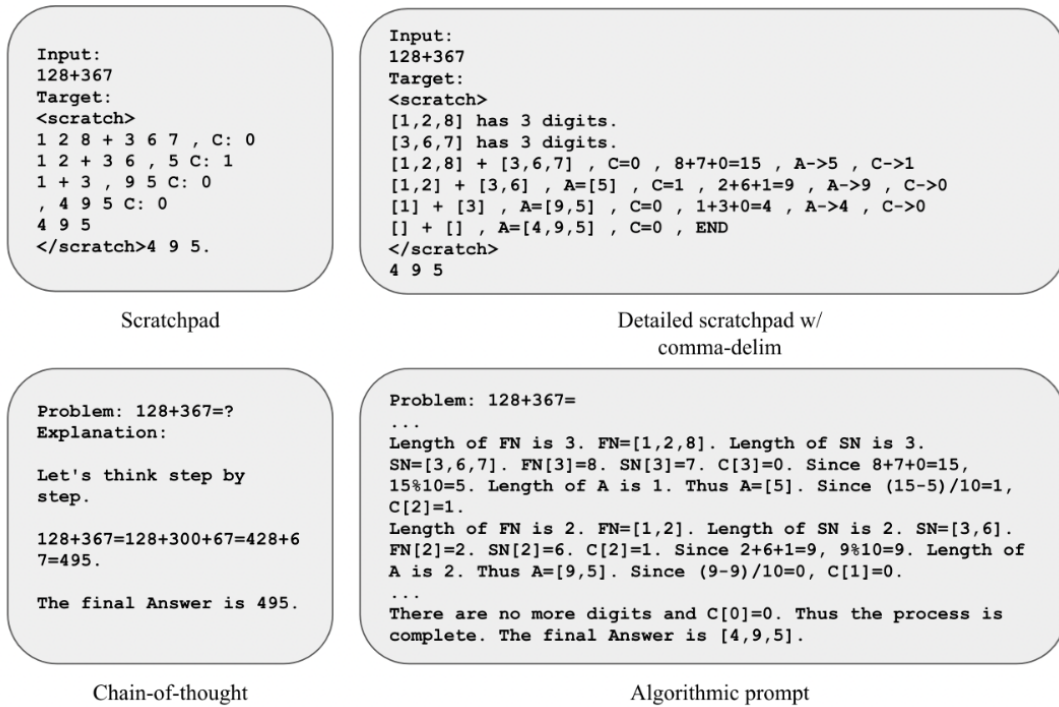


FIGURE 1.8: Copy of Zhou et al. (2022, Figure 8) showing different examples of in-context prompting styles.

Zhou et al. (2022) uses scratchpad/CoT reasoning for detailed in-context prompts of each step of the calculation. This includes steps which may be taken for granted such as identifying the number of digits in a number before using it. An example of the different prompting styles is given in Figure 1.8. Such *algorithmic prompting* improves OOD generalisation towards reasoning problems. In particular, this work identifies four learning stages for the LM to exhibit, inspired by how children are taught skills. The stages include teaching an algorithm (e.g., two input addition) as a skill, accumulating different skills (e.g., learning to do either addition or subtraction), composing skills (e.g., doing multi-number addition to complete a multiplication) and using skills as tools (e.g., using addition in a wider context problem which can require wider reasoning). Constraining the algorithmic prompts to be these comprehensive yet structured contexts allows the model to use the context as a human would expect. As a result, if the context was modified to include systematic errors the accuracy would also drop substantially. Due to the limitation of the allowed input context length in Transformers, for the skill composition tasks such as two number addition, gaining reasonable accuracy for OOD scenarios requires splitting the context and passing them to multiple models. However, using such a technique for the ‘skills as tools’ stage results in degrading the performance of informal logical reasoning.

Minerva (a large LM which is fine-tuned on high-quality mathematic datasets) which can also use in-context learning, CoT reasoning and majority voting was found to still make the most mistakes based on incorrect reasoning for the CoT (Lewkowycz et al.,



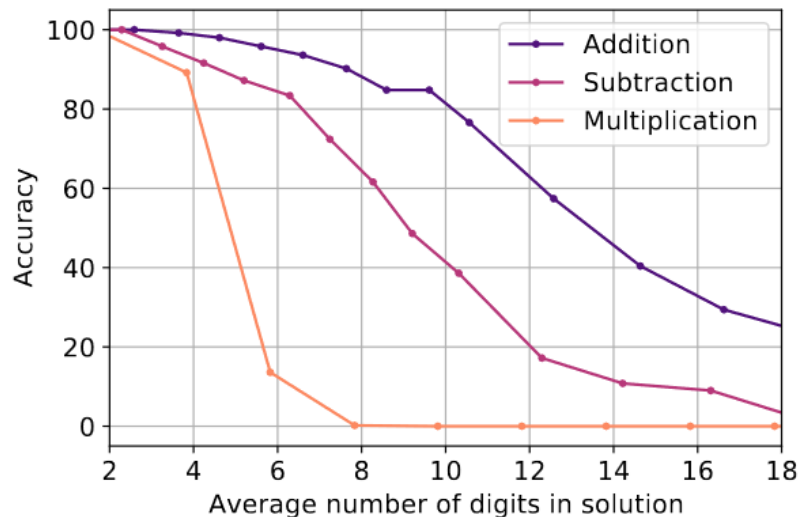


FIGURE 1.9: Copy of Lewkowycz et al. (2022, Appendix I; Figure 7). Accuracy on the single arithmetic operation of two numbers using Minerva. 500 arithmetic questions are sampled at random for each operation and choice of digits.

2022). Interestingly, Minerva exhibits only a few hallucinations meaning mathematical concepts do not get fabricated regularly. Failures were intuitive and predictable, suggesting some mathematical properties may be learnt. Another example of predictable failure includes observing the failure modes for a small Transformer (up to 6 layers) trained in matrix diagonalisation. Such models would consistently learn the eigenvalues and unit norms of rows and columns but fail on having orthogonal rows and columns (Charton, 2022a). This would remain the case even if the training would be prematurely ended or if larger training matrices were given.

Much progress is still required to be made for reasoning on tasks as simple as two input OOD arithmetic as shown by Figure 1.9. For example, comparing the performance of Minerva on multiplying two numbers with 4 vs 6 digit outputs shows a drop of accuracy from  $\sim 90\%$  to  $\sim 15\%$ . Transformer LMs are data-hungry, but many real-world datasets have tailed distributions where the longer the solution the lower the frequency (Anil et al., 2022). It is those long solutions which tend to correspond to the types of questions that LMs struggle with. Therefore, the LMs must learn to generalise from short-length solutions to longer ones. Furthermore, issues regarding robustness, compositionality and general OOD ability have been observed even though test accuracies are high (Welleck et al., 2022).

From comparing learning on in-domain data to OOD data, Transformers have shown a preference for parallel strategies which favour using shortcuts and spurious correlations rather than (sequential) serial ones (Anil et al., 2022). This can be observed for the binary parity task which determines if the number of set bits (1s) is odd or even. The learnt parallel strategy resulted in counting the frequency of ones and using a threshold to determine the output. Such a strategy is unable to generalise to longer

sequences. Another example of shortcuts can be found when learning the greatest common divisor (GCD) of two integers (e.g., for integers 18 and 24 the GCD would be 6) (Charton, 2022b). GCD can perform well using a base 30 encoding rather than the traditional base two or ten.<sup>5</sup> Larger bases have the advantage of shorter input and output sequences. However, the accuracy of the model becomes dependent on the base the model is trained on; a factor which should not affect the ability to do GCD. This suggests the model learns to cheat rather than learn the underlying mechanisms to solve the task, otherwise all bases would expect to have similar performance. The model learns shortcuts for easy cases, exploiting the representation of the given base representation, but as a result, fails to generalise. For example, in base two, by counting the number of rightmost 0's in a token it is possible to determine the GCD of values  $2^n$ .

To summarise, techniques such as in-context prompting, scratchpads, CoT reasoning and majority voting improve reasoning and generalisation of Transformers over scaling. However, the reasoning is not perfect and can still fail on even the basic arithmetic operations such as multiplication. There exists reliance on how inputs are encoded and habits towards shortcutting. Furthermore, there due to the lack of transparency of the parameters of the architecture, there is no guarantee of the network's ability to execute systematic solutions for mathematical tasks. In contrast, if instead specialists networks designed for systematic generalisation are used, it would be possible to have confidence in what the network learns without relying on large numbers of parameters that require tricks to infer from.

### 1.4.3 Representing Numbers in Machine Learning Models

Artificial neural networks with human cognitive limitations have been developed, such as simulating the distance effect where the time to discriminate two numbers increases as the distance between the two numbers decreases (Anderson et al., 1994).<sup>6</sup> But should we build networks bounded by such biases for learning arithmetic? If a machine can add, surely it should be able to add no matter the quantity with the same confidence each time. Such reliability is essential for applications such as equation discovery in physical and financial modelling. In other words, we want the representation and processing of the numbers to guarantee extrapolation; working on OOD data. Existing works in ML and DL have explored different types of encodings of numbers for arithmetic tasks (see Figure 1.10) including *static encodings* and *learnable vector embeddings*.

<sup>5</sup>An example of a base 10 symbolic representation of the number 24 would be tokens '+', '2', '4'.

<sup>6</sup>See Appendix B for further coverage of number representations in humans, animals and computers.

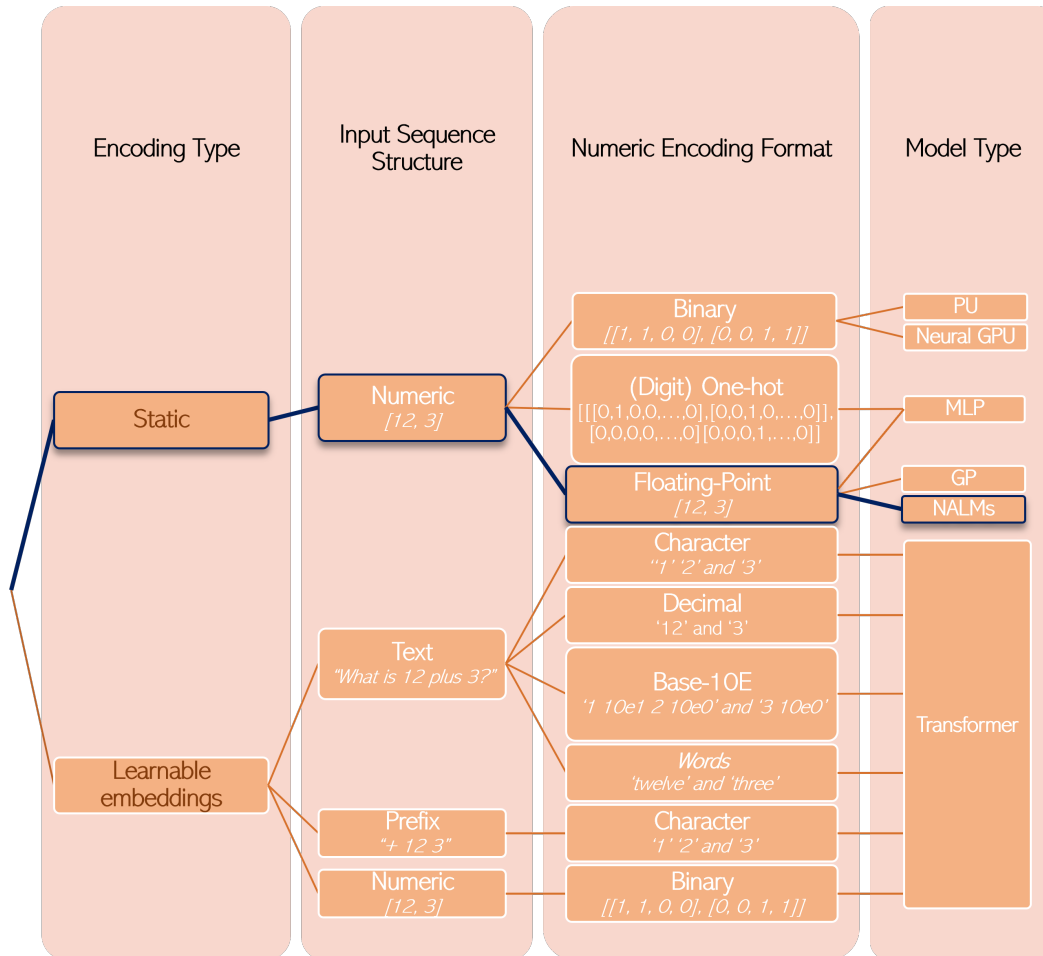


FIGURE 1.10: Breakdown of the types of static and learnable number encoding formats used in ML/DL. This example considers an input of adding 12 and 3. The *input sequence structure* represents how the input is given. The *numeric encoding format* is the strategy used to represent the numbers in the input. The *model type* is the type of architecture that can use such an encoding format. The blue path leads to how NALMs represent numbers which is the type of representation used in this thesis.

### 1.4.3.1 Static Encoding

Static encodings are one-to-one mappings that take the numeric inputs and convert them into the relevant encoded format to be used as input to the network. This can include converting the numeric input to either a binary, one-hot, or floating-point format.

**Binary Encoding.** Binary inputs were a popular encoding choice with earlier networks that designed multilayered neural networks to perform operations such as n-bit addition of  $N$  numbers, n-bit multiplication, or division of two numbers (Hajnal et al., 1987; Siu and Bruck, 1990; Siu et al., 1993; Cannas, 1995; Franco and Cannas, 1998). Durbin and Rumelhart (1989) attempted to create more expressible units by creating a Product Unit (PU). The PU is similar to a standard neuron with no activation but with two differences: (1) the cumulative summation is replaced with a cumulative product

and (2) the weights are no longer multiplied but are exponents to the input. The resulting PU unit therefore can express polynomials while requiring fewer neurons than a typical network to learn. However, [Martínez-Estudillo et al. \(2008\)](#) find the PUs are prone to falling into local minima when using gradient descent and therefore rely on non-differentiable evolutionary optimisation strategies such as crossover and mutation to converge.

Other, more recent architectures which process binary inputs are Neural GPUs ([Kaiser and Sutskever, 2016](#)). The Neural GPU is constructed from convolutional gated recurrent units and requires various training techniques such as curriculum learning, relaxed parameter sharing and dropout to learn. Neural GPUs can extrapolate to long sequence lengths (2000) from being trained on length 20 inputs for operations such as binary multiplication, however, the models are not robust as only a few Neural GPU models generalise to such a long sequences. [Freivalds and Liepins \(2017\)](#) offer an improvement by simplifying the architecture and training procedure by introducing diagonal gating and hard nonlinearities but require implementing additional cost functions to enforce their constraints.

**One-hot Encoding.** Integer inputs could also be represented using one-hot encodings of each digit. For example, [Nollet et al. \(2020\)](#) uses such an encoding on MLPs to learn long multiplication and addition for up to 7 digits. The network learns to break the task into processing steps representing sub-operations allowing for the (intermediary) input to act as an external memory. Similar to the Neural GPU's need for curriculum learning ([Kaiser and Sutskever, 2016](#)), active learning was required to control the difficulty of the dataset to learn long multi-digit multiplication. Certain neurons in the MLP were found to encode digit operations for some operands, however, extrapolation performance to longer digits remained untested.

**Floating-Point Encoding.** Alternatively, we can use numerical inputs directly but using a floating-point representation. Some approaches that can process raw numerical inputs include using MLPs, Genetic Programming (GP), and specialised arithmetic units (NALMs).

In Section 1.4.1, we have shown how MLPs can process inputs using floating-point representations when training to perform arithmetic operations. Our findings show that the MLPs could not learn operations such as multiplication or division; when the network has the capacity to model operations exactly (i.e., addition/subtraction) the ideal solutions were unable to be learned.

GP is a class of Genetic Algorithms that search over the space of computer programs using operations for natural selection ([Koza, 1994](#)). GP initialises a set of random candidate mathematical expressions (called a population) which gets iteratively evolved.

The evolution process involves selecting the best candidates (parents) via a fitness function (e.g., lowest error), creating offspring of the parents using crossover and mutation, and replacing the population with the new offspring. At the end of the iterations or when a stopping criterion is met, the fittest candidate is selected as the final solution. Though GP is able to explore vast search spaces, it is slow to converge and non-differentiable so gradient-based optimisation is difficult to use (Landajuela et al., 2022; Kamienny et al., 2022).

Specialised arithmetic units, commonly known as NALMs (Mistry et al., 2022a), have specially designed architectures that can select and process a vector of inputs represented as floating-point values to learn arithmetic expressions. Therefore, no form of input preprocessing or learning of embeddings is required. They are specially designed for extrapolative arithmetic meaning that they can generalise to OOD data.

### 1.4.3.2 Learnable Embeddings

Rather than static encodings of the input, learning vector embeddings, which take advantage of high-dimensional representations would be considered the most common approach for DL architectures such as Transformers. Nogueira et al. (2021) showed that the symbolic encoding of the representation of the numbers (numeric encoding format) in the input question matters for performance and data sample efficiency. For example, an addition/subtraction task given as a text based question for up to 60 digit numbers (completed using a pretrained T5 Transformer (Raffel et al., 2020)) found a base ten scientific encoding (e.g., 32 is represented as “3 10e1 2 10e0”) to outperform all other encodings including decimal, characters and words. Although better symbolic representations can positively impact performance, they were unable to find a representation which is generalisable to any input length. Russin et al. (2021) found pretrained Transformers on character-level mathematical expressions can, to some extent, break down maths expressions to their sub-expressions if given enough training data. However, rather than full compositionality, there is a mix between composition and memorisation (table lookup). The vector representations of the operators would store the aggregate results, while the digit representations were spatially organised according to their natural order in the mental number line (Dehaene, 1999), e.g., vectors representing one are closer to two than nine.

If mathematical questions are given in text form as input sequences (parsed as tokens) without any preprocessing (e.g., tree-parsing) then models struggle in tasks requiring multiple intermediate steps (Saxton et al., 2019). Processing inputs as expression trees which get converted into prefix sequences to aid in encoding structure can solve complex integral problems but has trouble with extrapolation over different input/output sequence length (Davis, 2019; Lample and Charton, 2020). For example, if trained on data which generates short derivative inputs then the model would perform poorly

on test data where the input data consists of long derivative inputs. This inability to generalise to different data generators also occurs in other mathematical tasks such as matrix diagonalisation where the train and test distributions of the eigenvalues would differ (Charton, 2022a).<sup>7</sup>

Rather than encoding token-based inputs, Yan et al. (2020) shows that using a binary encoding format for the inputs results in learning embeddings which if visualised show a systematic number system. Such emergent internal number systems contained in the learnt embedding space have been observed by many (Kondapaneni and Perona, 2020; Cognolato and Testolin, 2022; D’Ascoli et al., 2022). Probing popular pretrained embedding representations for numeracy finds fine-grained information about number magnitude and order, however weak OOD extrapolative performance when tested on numerical inputs whose magnitudes lie outside of the training range (Wallace et al., 2019). Naik et al. (2019) also find embeddings currently cannot capture precise magnitude ( $a < b$ ) and numeration ( $3 = \text{‘three’}$ ).

Overall, many recent efforts for training neural networks to learn extrapolative arithmetic focus on learning and manipulating high-dimensional representations which can be difficult to interpret and struggle with extrapolation. We on the other hand will use this thesis as an opportunity to explore the extent to which we can learn extrapolative arithmetic using modular architectures (i.e., NALMs) which work directly on real-valued inputs encoded using a floating point format.

## 1.5 Discovering Mathematical Expressions for Symbolic Regression

Throughout this thesis, we will investigate models which can model mathematical expressions. The focus will be on learning simple arithmetic, but if extended, the models explored can also be used to model rich expressions which occur in symbolic regression. Symbolic regression is the task of discovering the underlying relationship between the input variables ( $X$ ) and the target output variable ( $Y$ ). The relation is composed of a combination of mathematical operations and coefficients. Ideally, models will discover expressions that have a balance between accuracy and simplicity. That is, the expression is correct while also being composed with a minimal number of terms. The ability to discover such expressions from data makes symbolic regression models especially applicable to physics (Udrescu and Tegmark, 2020). For example, recently in astrophysics, Wadekar et al. (2023) use symbolic regression to discover a new equation (by adding a new term to the existing equation) that is able to reduce the scatter

---

<sup>7</sup>Some distributions such as Gaussian and Laplace (with heavy tails characteristics) are able to generalise to OOD distributions, however all other tested distributions could not.

in mass estimates of a galaxy cluster. These findings mean more accurate mass estimations can be given of existing/upcoming X-ray surveys such as eROSITA, therefore improving scientific contributions. Furthermore, the speed of equation discovery is another benefit of SR. For example, Brunton et al. (2016)'s ML symbolic regression model can discover the underlying equations for fluid dynamics which had taken experts 30 years to discover.

Examples of existing approaches for symbolic regression include GP (Genetic Programming), sparse regression models and DNNs. GP uses Evolutionary Algorithms (EA) to learn mathematical expressions (Koza, 1994; Schmidt and Lipson, 2009). EAs maintain a population of expressions where individuals of the population get selected based on a fitness function and modified via techniques such as crossover and mutation. Hybrid methods which combine GP with local search can also be used to further improve generalisation (Kommenda et al., 2020). The Sparse Identification of Nonlinear Dynamics (SINDy) model is a data-driven sparse regression approach for symbolic regression (Brunton et al., 2016). Given a library of pre-defined functions (e.g., selection, negation, squaring, etc.), SINDy will discover a subset of relevant functions along with a scaling coefficient for each selected function. Sparsity is promoted by using either the LASSO regression or a least squares-based criteria. Finally, DNNs using Transformers are another data-driven approach. Transformers treat the problem as a sequence-to-sequence translation (Biggio et al., 2021; Kamienny et al., 2022).

## 1.6 Research Questions and Contributions

This section gives the research questions investigated throughout this work and the novel contributions made. As NALMs are a fairly new research field, our work focuses on strengthening the foundational understanding regarding NALMs including how to benchmark, how to understand causes of failure, and how to compose.

### 1.6.1 Research Questions

#### **RQ 1: To what extent are NALMs robust to learning basic arithmetic operations?**

The first Research Question (RQ) focuses on assessing existing NALMs' ability to learn the basic arithmetic operations which they are built to do. To answer this question, we need to consider how to measure the performance of NALMs, motivating RQ 1.1, and in what ways to consider robustness, motivating RQ 1.2.

RQ 1.1: What evaluation strategies can be used to best analyse the different characteristics of a NALM?

RQ 1.2: Which NALMs are robust to learning on different training distributions?

**RQ 2: How can the failures in learning extrapolative models using NALMs be characterised?**

The second RQ focuses on investigating the causes of the observed failure cases of NALMs. Two directions are considered. The first (RQ 2.1) is exploring failures caused due to architecture design. The second (RQ 2.2) is exploring failures caused due to the nature of the data and task.

RQ 2.1: What are significant architecture choices which influence the ability to learn?

RQ 2.2: Can modifying the ML pipeline provide performance gains in robust extrapolation?

**RQ 3: Which strategies are most promising for combining different NALMs?**

The final RQ focuses on NALM compositionality for learning multi-step arithmetic. Similar to how a single operation can best be performed by a certain type of NALM, we investigate how the architectures used to combine NALMs influence learning in RQ 3.1. In RQ 3.2, we investigate designing a compositional architecture for NALMs.

RQ 3.1: To what extent does the way in which NALMs are combined affect their ability to compose?

RQ 3.2: How can we build an architecture to compose NALMs for learning different combinations?

## 1.6.2 Contributions

The novelty of this thesis lies in its findings on the feasibility of using NALMs as ML models to reliably learn basic arithmetic in an extrapolative manner. An overview of the thesis is given in Figure 1.11 and the contributions are summarised below.

**Evaluating and benchmarking NALMs.** In the context of RQ 1.1, we analyse existing evaluation methodologies for NALMs concluding that there is a lack of consistency in the research field. In an attempt to alleviate this issue, we propose a new synthetic arithmetic benchmark to measure the capabilities of NALMs in learning single-step operations. We extend an existing set of evaluation metrics based on the work of (Madsen and Johansen, 2019) for our benchmark (Chapter 3), which focuses on measuring performance in relation to extrapolation success, convergence speed, and weight interpretability for the desired operation. Our benchmark measures against various ranges



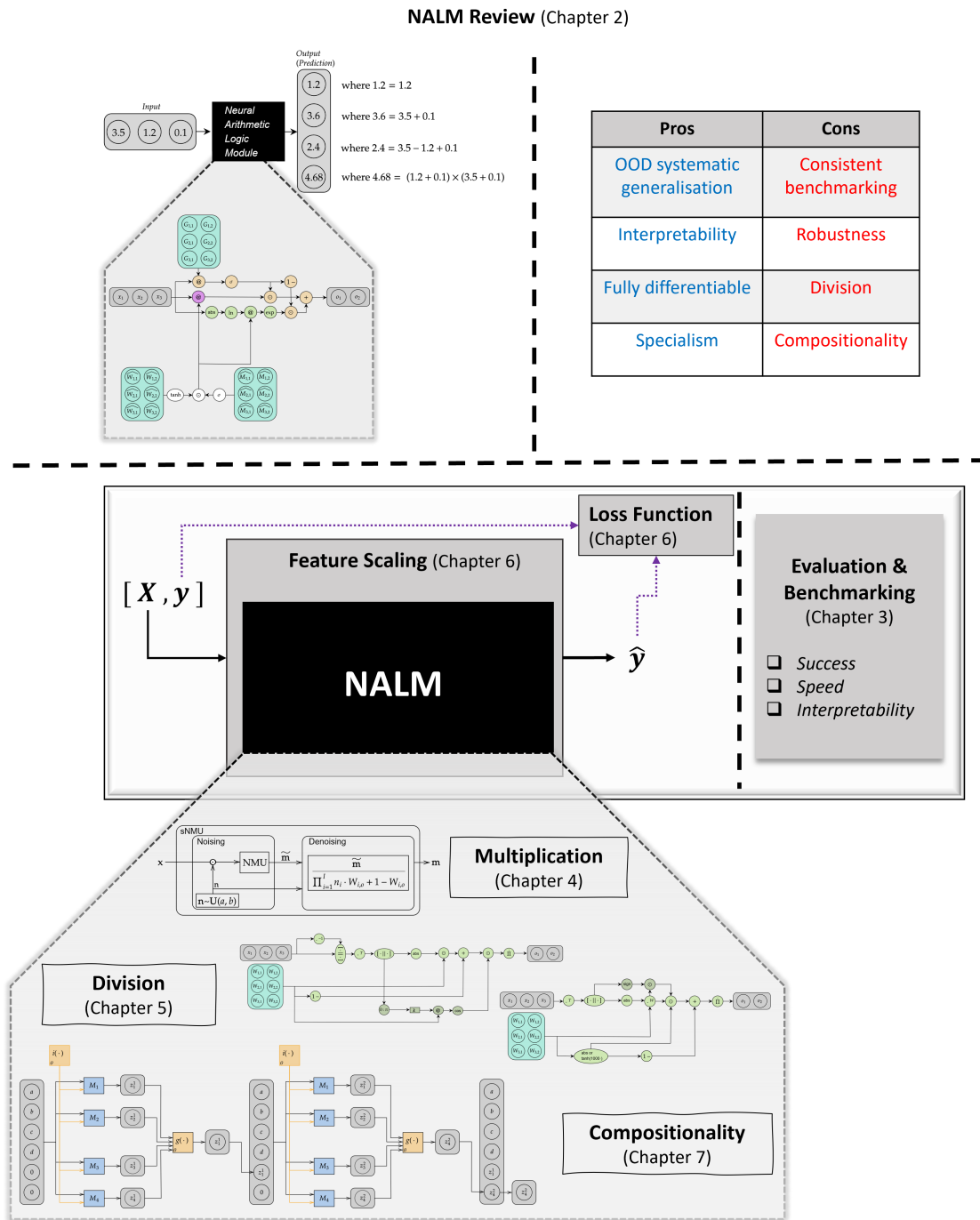


FIGURE 1.11: Overview of the different components of this thesis. Chapter 2 is a review of existing research on NALMs including areas of weakness. The remaining chapters consider the components of the ML pipeline. Chapter 3 focuses on ways to benchmark NALMs and have evaluation metrics suited for extrapolative arithmetic. Chapters 4, 5 and 7 investigate ways to build better NALMs for multiplication, division, and composition. Chapter 6 considers the effect of techniques such as feature scaling and using different loss functions for training.

and distributions throughout the thesis (Chapters 3-6) to answer RQ 1.2 where we show issues regarding robustness occurring in a range of NALMs.

The contributions for these RQs have been presented at the NeurIPS 2022 Journal Showcase Track and published in

- Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. A primer for neural arithmetic logic modules. *Journal of Machine Learning Research (JMLR)*, 23(185):1–58, 2022a. doi:10.48550/ARXIV.2101.09530.

**Challenges in Robustly Learning Multiplication and Division.** A comprehensive qualitative analysis of existing NALMs along with our findings from RQ 1, allows us to identify the key areas of weaknesses in NALMs which leads us to RQ 2. Our contributions for RQ 2.1 include designing a novel stochastic wrapper for reducing the chance of falling into local optima in multiplication in Chapter 4 and two novel NALMs for learning division in Chapter 5. We provide extensive empirical results for our division modules, comparing against an existing division NALM (the Real Neural Power Unit (NPU) (Heim et al., 2020)) as a case study. Our findings identify the types of data which hinder learning division for each module, including training on mixed-sign inputs, negative ranges, extremely small values and different distributions. For RQ 2.2, in Chapter 6, we contribute findings on how factors outside of the NALM architecture can significantly impact learning. We show how feature scaling impedes the learning of extrapolative weights by disturbing their ability to discretise. Furthermore, by evaluating three different losses we show the importance of the choice of loss function used when solving different arithmetic tasks for both numerical and image inputs.

A majority of the experiments for these RQs have been published in:

- Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. Exploring the learning mechanisms of neural division modules. *Transactions on Machine Learning Research (TMLR)*, 2022b. ISSN 2835-8856.
- Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. Improving the robustness of neural multiplication units with reversible stochasticity. *CoRR*, 2022c. doi:10.48550/ARXIV.2211.05624.

**Composition of NALMs.** In Chapter 7, we investigate compositionality to answer RQ 3. In relation to RQ 3.1, our findings indicate that deep stacking of NALMs does not create effective learners. We argue that the sparse and discrete nature of NALM parameters for the task results in gradient challenges when learning. To answer RQ 3.2, we show that performance can be improved if the compositional network has access to intermediate steps of working (a scratchpad) by designing an architecture influenced by the GWT. The resulting architecture forces competition between modules to update the scratchpad while broadcasting the information at every step.

## 1.7 Thesis Structure

We begin in Chapter 2 by providing the background knowledge required to understand NALMs. This includes providing motivation for using NALMs, explaining existing NALMs and reviewing the current shortcomings of the field. We also provide existing examples of applications where NALMs have been utilised in larger end-to-end networks showing the possible breadth of this research field. Chapter 3 begins by understanding how previous studies have evaluated NALMs, discovering a lack of consistent evaluation. In attempts to alleviate this, we further build on one work by adding a new benchmark with empirical results on the four main arithmetic operations over multiple well-known NALMs. Chapter 4 and Chapter 5 build on observations made in Chapter 3 which identify difficulty in learning multiplication and division reliably. Chapter 4 explore using a new technique of reversible stochasticity to reduce chances of falling into a local optima, while Chapter 5 discovers reasons as to why division is so difficult to learn and in the process introduces two new NALMs. Rather than looking at NALM architectures, Chapter 6 focuses on understanding the effects of changing different parts of the ML training pipeline on NALMs including feature scaling and losses. Chapter 7 investigates how NALMs can be combined to do compositional arithmetic. In particular, we show how building a globally accessible scratchpad containing intermediate workings can improve the chances of learning. Chapter 8 discusses the overall findings of the thesis and potential areas of future work for the field.



## Chapter 2

# Review of NALMs

In this chapter, we introduce and review the family of neural networks which we call NALMs. Our review begins with a high-level overview clarifying the what and whys and then delves into the formal definitions of existing NALMs. Once this foundation is laid, we use the first NALM, the NALU, as a case study to provide a comprehensive explanation of the existing pros and cons of the field as well as existing applications. We conclude with the remaining gaps in the field which become the focal points for the remainder of this thesis.

### 2.1 What are NALMs and Why use them?

We first answer three high-level yet important questions: 1. What is a NALM? 2. What is the aim of a NALM? 3. Why is a NALM useful? From answering these questions, we shall arrive at the following definition: *A NALM is a neural network that performs arithmetic and/or logic based expressions which can extrapolate to OOD data when parameters are appropriately learnt whilst expressing an interpretable solution.*

#### 2.1.1 What is a NALM?

NALM stands for Neural Arithmetic Logic Module. *Neural* refers to neural networks. *Arithmetic* refers to the ability to learn arithmetic operations such as addition. *Logic* refers to the ability to learn operations such as selection, comparison based logic (e.g., greater than) and boolean based logic (e.g., conjunction). *Module* refers to the architecture's of the neural units which learn arithmetic and/or logic operations. The term module encompasses both a single (sub-)unit and multiple (sub-)units combined together.

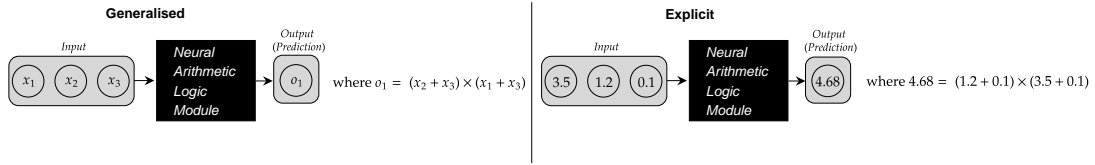


FIGURE 2.1: High-level example of the input-output structure into a NALM. Both networks are the same. The generalised network defines the notation of each element in the input and output. The explicit network is an example of valid input and output values.

**What kind of operations can be learnt?** Existing work has tried to model arithmetic operations including addition, subtraction, multiplication, division and powers. Logic based operations include logic rules (e.g., conjunction) (Reimann and Schwung, 2019), control logic (e.g.,  $\leq$ ) (Faber and Wattenhofer, 2020) and selection of relevant inputs.

**How are operations learnt?** Because a NALM is a neural network, a module can model the relation between input and output vectors via supervised learning which trains parameters through backpropagation. To learn the relation between input and output requires learning to select relevant elements of the input and apply the relevant operation/s to the selected input to create the output.

**How is data represented?** The input and outputs are both vectors. Each vector element is a real-valued number which is implemented as a floating-point number. An example of a single data sample is illustrated in Figure 2.1 where we assume that the NALM used (made from two stacked sub-units) can learn addition, subtraction and multiplication. In practice, data would be given in batch form for training. Furthermore, there can be multiple output elements which can learn a different arithmetic expressions.

### 2.1.2 What is the Aim of a NALM?

The aim of NALMs is to provide systematic generalisation in learning arithmetic and/or logic expressions. Once the learning state (training) has ended, if the correct weights are learnt, the NALM can also extrapolate to unseen data (i.e., OOD data).

**What does interpretability mean for NALMs?** Imagine modelling the relation between input  $\mathbf{x}$  and output  $\mathbf{y}$  with a module  $f$  parameterised by  $\theta$ , i.e.,  $\mathbf{y} = f_{\theta}(\mathbf{x})$ . We say a NALM is interpretable if you can set the module's parameters ( $f_{\theta}$ ) to express the underlying relation between  $\mathbf{x}$  and  $\mathbf{y}$  in a provable way. Simply put, the parameters of a NALM can be set such that, if the expression which the NALM calculates is written out, it will hold for all valid inputs. For example, for modelling the addition of two inputs ( $x_1$  and  $x_2$ ), having a model which takes in the two inputs and applies a dot product with a weight vector set to ones results in  $\mathbf{y} = f_{\theta}(\mathbf{x}) = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$  which will always result in the output being  $x_1 + x_2$  no matter the values of  $x_1$  and  $x_2$ .

More broadly speaking, the type of interpretability we want from a NALM is *decomposable transparency* (Lipton, 2016). Transparency means to understand how the model works. Decomposability is transparency at the component level defined by Lipton (2016) as ‘each part of the model - each input, parameter, and calculation - admits an intuitive explanation’. For example, for modelling  $\text{force} = \text{mass} \times \text{acceleration}$ , there are: the two inputs into the NALM representing mass and acceleration, the (multiplication) operation parameters which are set to 1, and the calculation that multiplies the two inputs resulting in the value for the force.

**What does extrapolation on OOD data mean for NALMs?** *OOD data* refers to data which is sampled outside the training distribution. For example, if trained on a range [0,10] a valid OOD range could be [11,20]. *Extrapolation* is the ability to correctly predict the output when given OOD data. In the context of NALMs, extrapolation means that the model successfully learns the underlying principles for modelling the (arithmetic/logic) operations it is designed for. From a practical viewpoint, a NALM with successful extrapolative capabilities can be considered as a module where the loss in predictive accuracy only occurs due to numerical imprecisions of hardware limitations.

### 2.1.3 Why is a NALM useful?

The ability to learn arithmetic seems trivial in comparison to other architectures such as LSTMs, CNNs or Transformers which can be used as standalone networks to learn tasks such as arithmetic, object recognition and language modelling. So, why care about NALMs? Learning arithmetic, though it may seem a simple task, remains unsolved for neural networks. Solving this problem requires precisely learning the underlying rules of arithmetic such that failure cases will not occur in cases of OOD data. Therefore, before considering more complex tasks, solving simple tasks seems reasonable. Furthermore, even though NALMs specialise in arithmetic there is no restriction in using them as part of larger end-to-end neural networks. For example, attaching a NALM to a CNN as residual connections (Rana et al., 2020) to improve counting in images. In Section 2.4, we discuss a vast array of applications in which NALMs are being utilised. The extrapolative and interpretable properties of NALMs would also be desirable when looking at alternate disciplines. Interpretability and transparency for ML models have been acknowledged as necessary ingredients for obtaining a scientific outcome in the natural science (Roscher et al., 2020), materials science and chemistry domains (Oviedo et al., 2022). For example, interpretable models such as Generalised Additive Models have been used to model and interpret the key features of chemical adsorption of subsurface alloys (Esterhuizen et al., 2020).

TABLE 2.1: A summary regarding the key properties of different NALMs. Properties include the *operations* which can be modelled, if a form of explicit *gating* is used, if the NALM can process *negative inputs* and if the NALM uses a form of *discretisation regularisation* on its parameters. \* indicates the NALM can only learn arbitrary powers (i.e.,  $x_i^p$  where p is a real number) if the power value is between -1 to 1.

	NALU	iNALU	NAU	NMU	NPU	G-NALU	NLRL	NSR
<b>Addition</b>	✓	✓	✓			✓		
<b>Subtraction</b>	✓	✓	✓			✓		
<b>Multiplication</b>	✓	✓		✓	✓	✓		
<b>Division</b>	✓	✓			✓	✓		
<b>Arbitrary Power</b>	✓*				✓	✓*		
<b>Boolean Logic</b>							✓	
<b>Control Logic</b>								✓
<b>Gating</b>	✓	✓			✓	✓	✓	
<b>Negative Input</b>		✓	✓	✓	✓			✓
<b>Discretisation</b>		✓	✓	✓				

## 2.2 Existing NALMs Architectures

This section introduces existing NALM architectures along with their mathematical definitions. The mathematical notation will be in an element-wise form which provides how to calculate an output element  $y_o$  indexed at  $o$  given a single data sample (input vector  $\mathbf{x}$ ). For completeness, we also provide illustrations for each module<sup>1</sup> using the matrix/vector with symbols and colouring following the key in Figure 2.2.

We begin with the first NALM, the NALU (Trask et al., 2018) followed by the: Improved NALU (iNALU) (Schlör et al., 2020), Neural Addition Unit (NAU) / Neural Multiplication Unit (NMU) (Madsen and Johansen, 2020), Neural Power Unit (NPU) (Heim et al., 2020), Golden Ratio NALU (G-NALU) (Rana et al., 2019), Neural Logic Rule Layer (NLRL) (Reimann and Schwung, 2019) and Neural Status Register (NSR) (Faber and Wattenhofer, 2020). A summary of the NALM properties is given in Table 2.1.

→	Data flow	⊙	Element-wise multiplication	○	Input/output
□	Intermediary result	+	Element-wise addition	⊖	Learnable parameter
@	Matrix Multiplication	−	Element-wise subtraction	⊙	Part of gating calculation
⊗	Kronecker product	1 −	Element-wise 1-<value>	⊕	Part of summative path calculation
$\cdot^T$	Transpose input	$\cdot^{-1}$	Element-wise reciprocal of input	⊗	Part of multiplicative path calculation
$[\cdot \parallel \cdot]$	Concatenate input (column-wise)	$\cdot^W$	Element-wise power	⊗	Part of multiplicative sign retrieval calculation
$\begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$	Concatenate input (row-wise)				

FIGURE 2.2: Key of the symbols and colouring system for architecture illustrations.

<sup>1</sup>Module illustrations from the original papers are provided in Appendix C.1.



## 2.2.1 NALU

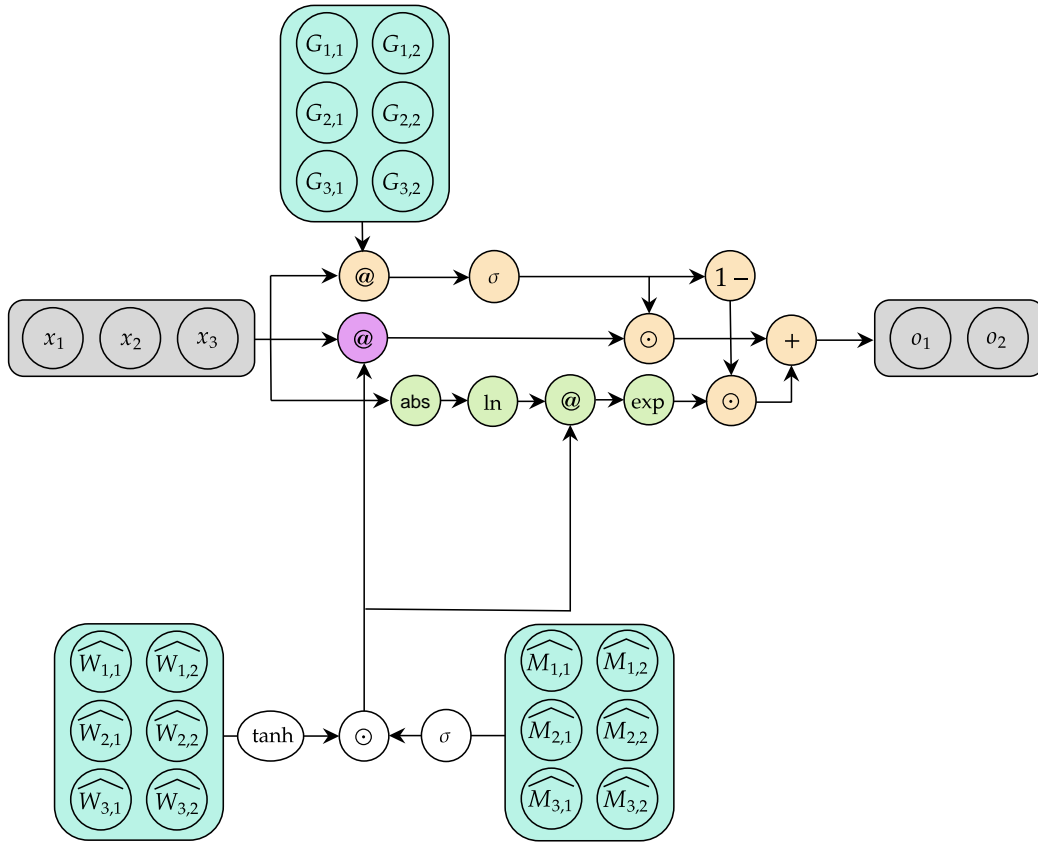


FIGURE 2.3: NALU architecture. Example of a 3-feature input and 2-feature output model.

The NALU, illustrated in Figure 2.3, provides the ability to model basic arithmetic operations, specifically: addition, subtraction, multiplication and division. The NALU requires no indication of which operation to apply and aims to learn weights that provide extrapolation capabilities if correctly converged. The NALU comprises of two sub-units, a summative unit that models  $\{+, -\}$  and a multiplicative unit that models  $\{\times, \div\}$ . Following the notation of Madsen and Johansen (2020) we denote the sub-units as  $\text{NAC}_+$  and  $\text{NAC}_\bullet$ , respectively. Formally, for calculating a specific output value, the

NALU is expressed as:

$$W_{i,o} = \tanh(\widehat{W}_{i,o}) \odot \text{sigmoid}(\widehat{M}_{i,o}) \quad (2.1)$$

$$\text{NAC}_+ : a_o = \sum_{i=1}^I (W_{i,o} \cdot x_i) \quad (2.2)$$

$$\text{NAC}_\bullet : m_o = \exp \left( \sum_{i=1}^I (W_{i,o} \cdot \ln(|x_i| + \epsilon)) \right) \quad (2.3)$$

$$g_o = \text{sigmoid} \left( \sum_{i=1}^I (G_{i,o} \cdot x_i) \right) \quad (2.4)$$

$$\text{NALU} : \hat{y}_o = g_o \cdot a_o + (1 - g_o) \cdot m_o \quad (2.5)$$

where  $\widehat{W}, \widehat{M} \in \mathbb{R}^{I \times O}$  are learnt matrices ( $I$  and  $O$  represent input and output dimension sizes). A non-linear transformation is applied to each matrix and then both are combined via element-wise multiplication to form  $W$  (Equation 2.1). Due to the range values of  $\tanh$  and  $\text{sigmoid}$ ,  $W$  aims to have an inductive bias towards values  $\{-1, 0, 1\}$  which can be interpreted as selecting a particular operation *within* a sub-unit (i.e., intra-sub-unit selection). For example, in the  $\text{NAC}_+$  +1 is addition and -1 is subtraction, and in the  $\text{NAC}_\bullet$  +1 is multiplication and -1 is division. In both sub-units, 0 represents not selecting/ignoring an input element. A sigmoidal gating mechanism (Equation 2.4) enables selection *between* the sub-units (i.e., inter-sub-unit selection), where an open gate, 1, selects the  $\text{NAC}_+$  and closed gate, 0, selects the  $\text{NAC}_\bullet$ . Once trained, the gating should ideally select a single sub-unit.  $G$  is learnt and the gating vector  $g$  represents which sub-unit to use for each element in the output vector. Finally, Equation 2.5 gates the sub-units and sums the result to give the output. NALU's gating only allows for each output element to have a mixture of operations from the same sub-unit. Therefore, each output element is an expression of a combination of operations from either  $\{+, -\}$  or  $\{\times, \div\}$  but not  $\{+, -, \times, \div\}$ . This issue is fixed by stacking NALUs such that the output of one NALU is the input of another. For a step-by-step example of the NALU, see Appendix C.2. Next, we overview the architectures of some recent modules.

## 2.2.2 iNALU

The iNALU identifies key issues in the NALU and modifies the unit to incorporate solutions (detailed in Section 2.3). In particular, they use:

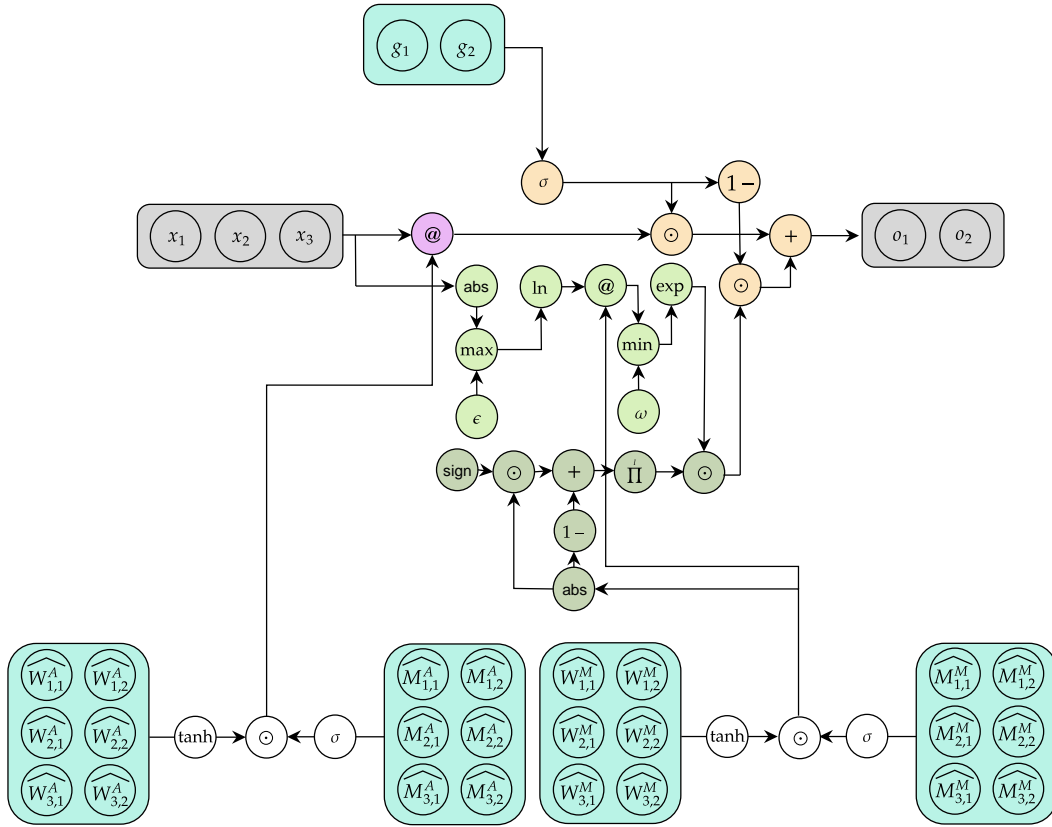


FIGURE 2.4: iNALU architecture. Example of a 3-feature input and 2-feature output model.

- **Independent weight matrices.** To allow the multiplicative and summative paths to learn their own set of  $\widehat{W}$  and  $\widehat{M}$  weights to be used in calculating  $a$  for the  $\text{NAC}_+$  and  $m$  for the  $\text{NAC}_\bullet$ .

$$W_{i,o}^A = \tanh(\widehat{W}_{i,o}^A) \cdot \text{sigmoid}(\widehat{M}_{i,o}^A), \quad (2.6)$$

$$W_{i,o}^M = \tanh(\widehat{W}_{i,o}^M) \cdot \text{sigmoid}(\widehat{M}_{i,o}^M). \quad (2.7)$$

- **Clipping.** Clipping the multiplicative weights using the equation below (with  $\omega = 20$ ) and clipping the gradient of learnable parameters between  $[-0.1, 0.1]$ .

$$m_o = \exp(\min(\ln(\max(|x_i|, \epsilon)) \cdot W_{i,o}^M, \omega)). \quad (2.8)$$

- **Multiplicative sign correction.** Retrieve the output sign of the multiplicative path,

$$msv_o = \prod_{i=1}^I \left( \text{sign}(x_i) \cdot |W_{i,o}^M| + 1 - |W_{i,o}^M| \right). \quad (2.9)$$

- **Regularisation.** Include a regularisation loss term that avoids having near-zero learnable parameters,

$$\mathcal{R}_{\text{sparse}} = \frac{1}{t} \sum_{\theta \in \{\widehat{W}^A, \widehat{M}^A, \widehat{W}^M, \widehat{M}^M, g\}} \frac{\sum_o^O \sum_i^I \max(\min(-\theta_{i,o}, \theta_{i,o}) + t, 0)}{O \cdot I}, \quad (2.10)$$

where  $t = 20$ . This activates if the loss is under 1 and there have been over 10 iterations of training data.

- **Reinitialisation.** Reinitialise the model weights if the average loss collected over a number of consecutive iterations has not improved. More specifically, reinitialisation occurs for every 10<sup>th</sup> iteration, if over 10,000 iterations have occurred and the average loss of the first half of those iterations of the errors is less than the average loss of the second half plus its standard deviation, and the average loss of the latter half of errors is larger than 1.
- **Independent gating.** Remove the dependence of the input values when learning the gate parameters,

$$g_o = \text{sigmoid}(g_o). \quad (2.11)$$

The iNALU expression for calculating a single output element indexed at  $o$  is

$$\text{iNALU} : \hat{y}_o = g_o \cdot a_o + (1 - g_o) \cdot m_o \cdot msv_o. \quad (2.12)$$

**Applications.** Schlör et al. (2020) create a hybrid layer consisting of iNALU and ReLU neurons for fraud detection, under the assumption that financial data inherently has some underlying mathematical relations. If data is fraudulent then the underlying patterns would not be followed hence making it identifiable.

### 2.2.3 NAU and NMU

The NAU and NMU are sub-units for addition/subtraction and multiplication respectively. The NAU and NMU definitions for calculating an output element indexed at  $o$  are:

$$\text{NAU} : a_o = \sum_{i=1}^I (W_{i,o} \cdot x_i), \quad (2.13)$$

$$\text{NMU} : m_o = \prod_{i=1}^I (W_{i,o} \cdot x_i + 1 - W_{i,o}), \quad (2.14)$$

where the  $W$  is unique for each sub-unit. Prior to applying the weights of a sub-unit to the input vector, each element of  $W$  is clamped between  $[-1,1]$  if using the NAU, or  $[0,1]$

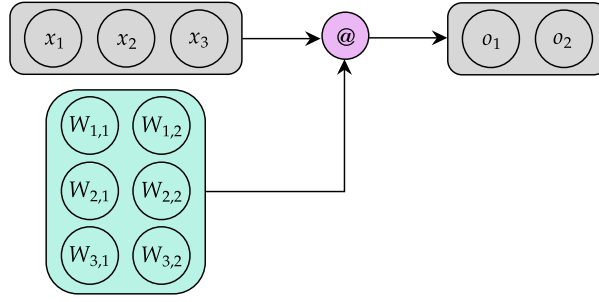


FIGURE 2.5: NAU architecture. Example of a 3-feature input and 2-feature output model.

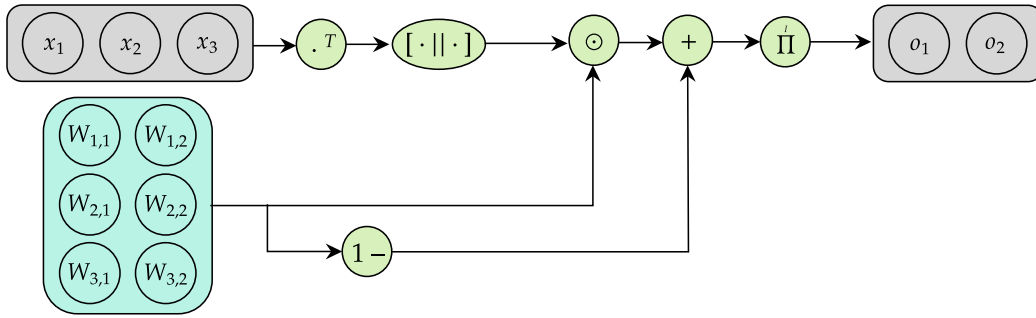


FIGURE 2.6: NMU architecture. Example of a 3-feature input and 2-feature output model.

if using the NMU. Therefore, considering discrete weights  $\{-1, 0, 1\}$ , Equation 2.13 will do the summation of the inputs where each input is either added ( $W_{i,o} = 1$ ), ignored ( $W_{i,o} = 0$ ), or subtracted ( $W_{i,o} = -1$ ). When considering the discrete weight values of the NMU  $\{0, 1\}$ , the result is the product of the inputs where each input is either multiplied ( $W_{i,o} = 1$ ) or not selected ( $W_{i,o} = 0$ ). Rather than allowing the product of the inputs to be multiplied by 0 whenever an irrelevant input (i.e., with a weight of 0) is processed, Equation 2.14 will also convert the input to be 1 (the multiplicative identity value) resulting in the irrelevant input not having any effect towards the final output.

**Regularisation.** To enforce the module weights to become discrete values, the following regularisation loss term is also used,

$$\mathcal{R}_{sparse} = \frac{1}{I \cdot O} \sum_{o=1}^O \sum_{i=1}^I \min(|W_{i,o}|, 1 - |W_{i,o}|). \quad (2.15)$$

A scaling factor for regularisation strength

$$\lambda = \hat{\lambda} \max \left( \min \left( \frac{iteration_i - \lambda_{start}}{\lambda_{end} - \lambda_{start}}, 1 \right), 0 \right), \quad (2.16)$$

is multiplied to  $\mathcal{R}_{sparse}$ , where the regularisation strength is scaled by a predefined  $\hat{\lambda}$ . The regularisation will grow from 0 to  $\hat{\lambda}$ , between iterations  $\lambda_{start}$  and  $\lambda_{end}$ , after which it plateaus and remains at  $\hat{\lambda}$ .

**Applications.** As well as modelling physical equations, the NAU and NMU have been integrated with Variational Autoencoders to model non-linear Ordinary Differential Equations (ODEs) (Heim et al., 2019). Pei et al. (2021) uses the NAU as part of their Stateformer architecture which learns to recover types from binaries. The NAU is used to learn the data flow for assignment (e.g., ‘mov’) and arithmetic (e.g., ‘add’) instructions by representing their numerical values (in decimal or hexadecimal formats) as embeddings. The NAU acts on a high dimensional embedding of a byte of the partial data state resulting in aggregate data state embeddings. From visualising learnt NAU representations via a t-SNE representation, they find that the NAU learns to represent the embeddings in an ordered ring-shaped cluster (Pei et al., 2021, Appendix 1.7).

## 2.2.4 NPU and Real NPU

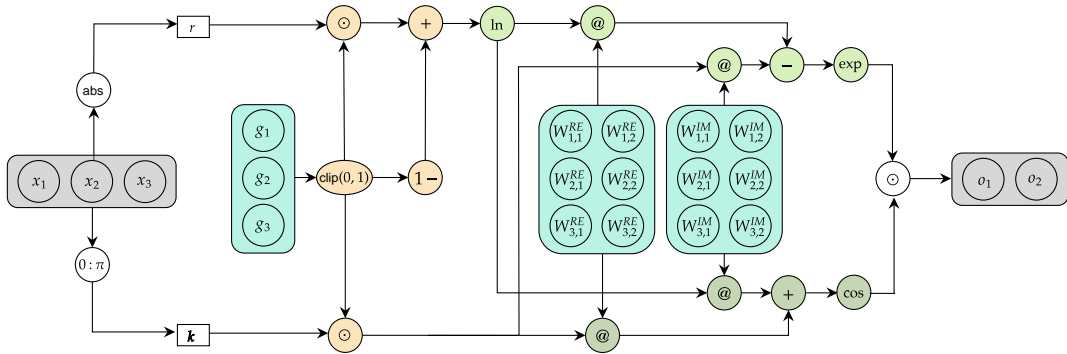


FIGURE 2.7: NPU architecture. Example of a 3-feature input and 2-feature output model.

The NPU (Equation 2.17) focuses on improving the division ability of the NAC<sub>•</sub> by applying a complex log transformation and using real and complex weight matrices ( $W^{RE}$  and  $W^{IM}$  respectively).<sup>2</sup>

NPU based modules can model products of arbitrary powers ( $\prod x_i^{w_i}$ ), therefore the learnable weight parameters do not require to be discrete. For example, modelling the square-root operation requires  $W_{i,0}^{RE} = 0.5$ . The  $r$  (Equation 2.18) converts values close to 0 into 1 to avoid the output multiplication becoming 0. To do this, a relevance gate ( $g$ ) is learnt representing if an input element is relevant and should be used as part of an output expression ( $g_i = 1$ ) or not be selected ( $g_i = 0$ ). Furthermore, each element of

<sup>2</sup>See Appendix C.3 for the derivation of converting the NAC<sub>•</sub> to the NPU.

$g$  is clipped between the range  $[0,1]$  (Equation 2.20).

$$\begin{aligned} \text{NPU} : y_o = & \exp \left( \sum_{i=1}^I (W_{i,o}^{\text{RE}} \cdot \ln(r_i)) - \sum_{i=1}^I (W_{i,o}^{\text{IM}} \cdot k_i) \right) \\ & \cdot \cos \left( \sum_{i=1}^I (W_{i,o}^{\text{IM}} \cdot \ln(r_i)) + \sum_{i=1}^I (W_{i,o}^{\text{RE}} \cdot k_i) \right) \end{aligned} \quad (2.17)$$

where

$$r_i = g_i \odot (|x_i| + \epsilon) + (1 - g_i), \quad (2.18)$$

$$k_i = \begin{cases} 0 & x_i \geq 0 \\ \pi g_i & x_i < 0 \end{cases}, \quad (2.19)$$

and

$$g_i = \min(\max(g_i, 0), 1). \quad (2.20)$$

Additionally a simplified version of the NPU exists, named the **Real NPU**, considering only real values of Equation 2.17:

$$\text{Real NPU} := \exp \left( \sum_{i=1}^I (W_{i,o}^{\text{RE}} \cdot \ln(r_i)) \right) \cdot \cos \left( \sum_{i=1}^I (W_{i,o}^{\text{RE}} \cdot k_i) \right). \quad (2.21)$$

As the NPU and Real NPU can express arbitrary powers, using a regulariser to enforce discrete parameters like in the iNALU, NAU or NMU would restrict the expressiveness. Therefore, Heim et al. (2020) use a scaled  $L_1$  penalty, where the scaling value  $\beta$  grows between predefined values  $\beta_{start}$  to  $\beta_{end}$  and is increased every  $\beta_{step} = 10,000$  iterations by a growth factor  $\beta_{growth} = 10$ .

### 2.2.5 G-NALU

The **G-NALU** replaces the exponent base in the tanh and sigmoid operations when calculating NALU's weight matrix with a golden ratio base value:

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \quad (2.22)$$

$$\text{sigmoid}_{\text{gr}} = \frac{1}{(1 + \phi^{-x})} \quad (2.23)$$

$$\text{tanh}_{\text{gr}} = \frac{\phi^{2x} - 1}{\phi^{2x} + 1} \quad (2.24)$$

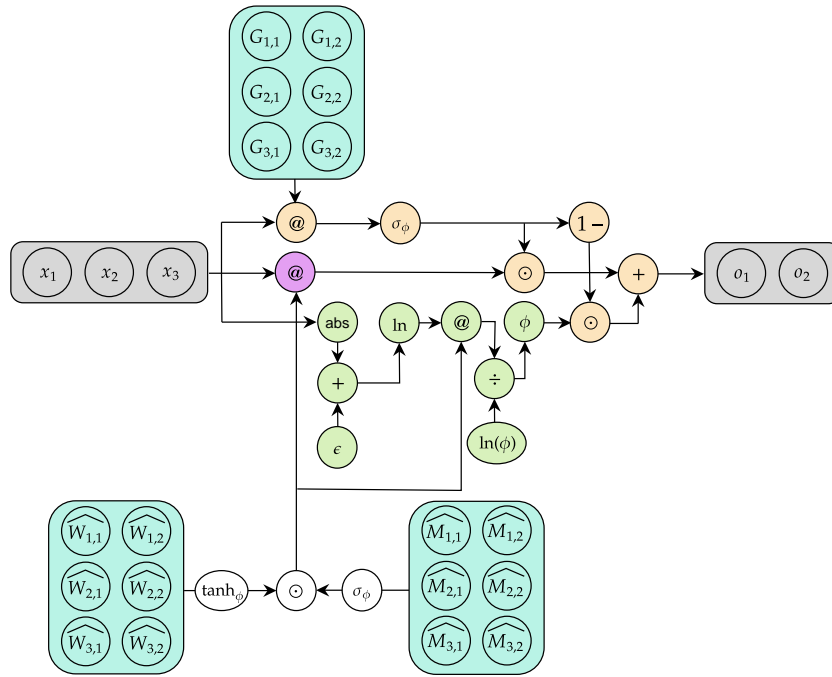


FIGURE 2.8: G-NALU architecture. Example of a 3-feature input and 2-feature output model.

The use of a golden ratio base also requires the  $\text{NAC}_\bullet$  definition (Equation 2.3) to be modified into Equation 2.25 to allow for the  $\ln$ -exp transformation to work.

$$\text{NAC}_\bullet : m_o = \phi \left( \frac{\sum_{i=1}^I (W_{i,o} \ln(|x_i| + \epsilon))}{\ln(\phi)} \right) \quad (2.25)$$

## 2.2.6 NLRL

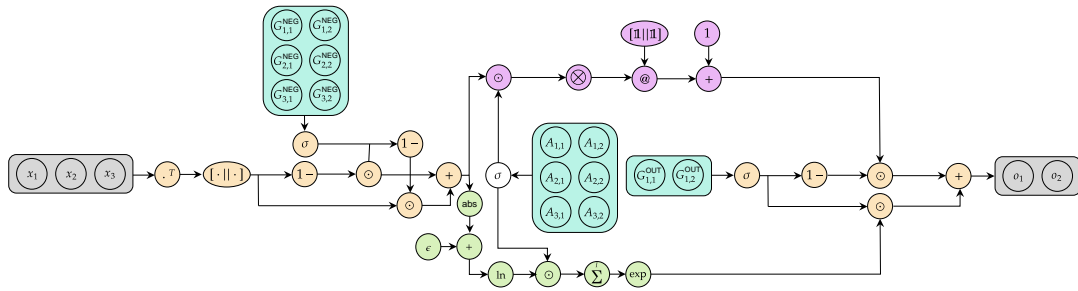


FIGURE 2.9: NLRL architecture. Example of a 3-feature input and 2-feature output model.

The **NLRL**, Figure 2.9, is a module to express boolean logic rules via modelling AND (conjunction), OR (disjunction) and NOT (negation). By stacking NLRLs together, it



is also possible to represent more complex relations including implication, exclusive OR and equivalence. The architecture is designed under the assumption of modelling the logic rules on booleans, therefore the input values must be booleans. The default NLRL architecture consists of the following four parts in which the three base operators (negation, conjunction and disjunction) are modelled:

- **Negation gating**, which models the negation operator. The (negation) gate determines if an input element should be negated (gate value of 1) or simply passed along (gate value of 0).

$$\hat{x}_{i,o} = (1 - \text{sigmoid}(G_{i,o}^{\text{NEG}})) \cdot x_{i,o} + \text{sigmoid}(G_{i,o}^{\text{NEG}}) \cdot (1 - x_{i,o}) \quad (2.26)$$

- **OR calculation**, which applies disjunctions (weight value of 1) for the output of the input gating.

$$z_o^{\text{OR}} = \bigotimes_{i=2}^I \left( \left[ 1 \quad -A_{i,o} \cdot \hat{x}_{i,o} \right] \otimes \left[ -1 \quad A_{1,o} \cdot \hat{x}_{1,o} \right] \right) \mathbf{1} + 1 \quad (2.27)$$

- **AND calculation**, which applies conjunctions (weight value of 1) over the output of the input gating. The definition is the same as the NAC $\bullet$  (Equation 2.3) used in the NALU.

$$z_o^{\text{AND}} = \exp \left( \sum_{i=1}^I (A_{i,o} \cdot \ln(|\hat{x}_{i,o}| + \epsilon)) \right) \quad (2.28)$$

- **Output gating**, which determines whether an output value should use the OR calculation (gate value 0) or AND calculation (gate value 1).

$$\hat{y}_o = (1 - \text{sigmoid}(G_{i,o}^{\text{OUT}})) \cdot z_o^{\text{AND}} + \text{sigmoid}(G_{i,o}^{\text{OUT}}) \cdot z_o^{\text{OR}} \quad (2.29)$$

Three parameter matrices require to be learnt. One for learning the gate values for negation ( $G^{\text{NEG}}$ ), another for learning the (shared) weight values for the AND and OR calculations ( $A$ ) and one for learning the gate values for the output ( $G^{\text{OUT}}$ ).

Optionally, the application of the De-Morgan laws enables representing a conjunction using only negation and disjunction and representing disjunction using only negation and conjunction, making it possible to modify the architecture to only need either the AND or OR calculation block. The changes require removing the unwanted calculation block and replacing the output gate with a negation gate. Using only the negation and conjunction operators is favoured as the implementation of disjunction requires using the Kronecker product which scales poorly with input size.

## 2.2.7 NSR

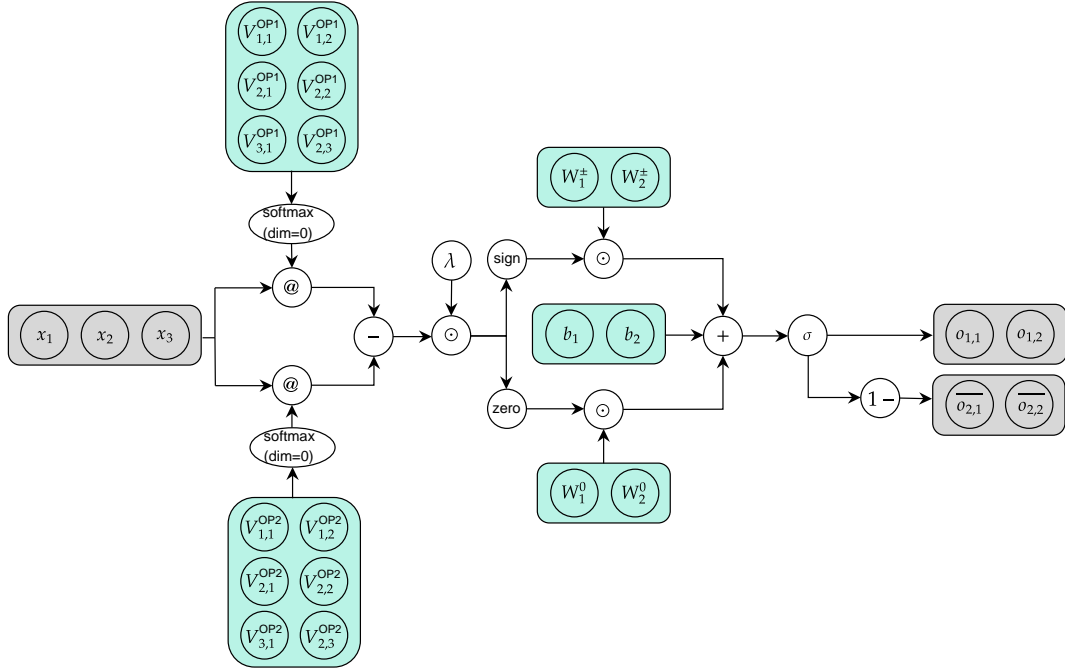


FIGURE 2.10: NSR architecture. Example of a 3-feature input and 2-feature output model.

The **NSR** (inspired by physical status registers found in the Arithmetic Logic Unit's of computers), models comparison based control logic:  $<$ ,  $>$ ,  $!$ ,  $=$ ,  $>=$ ,  $<=$ . Simply put, the NSR does quantitative reasoning by learning what input elements to compare and how to compare them. A NSR will output two elements. The first represents if the comparison is true (or false) and the second is the negation of the first output (i.e.,  $1 - o_1$ ). The negation is given such that when the NSR is used in a downstream task, the other layers can have access to either branch of the comparison.

The NSR architecture does the following:

1. Learns two matrices ( $V^{\text{OP1}}$  and  $V^{\text{OP2}}$ ) whose purpose is to select two inputs to be operands ( $\widehat{x}^{\text{OP1}}$  and  $\widehat{x}^{\text{OP2}}$ ) of the comparison function.

$$\widehat{x}_o^{\text{OP1}} = \sum_i^I (x_i \cdot \text{softmax}(V_{i,o}^{\text{OP1}})) \quad (2.30)$$

$$\widehat{x}_o^{\text{OP2}} = \sum_i^I (x_i \cdot \text{softmax}(V_{i,o}^{\text{OP2}})) \quad (2.31)$$

2. Takes the difference of the two selected operands.

$$\widehat{x}_o = \widehat{x}_o^{\text{OP1}} - \widehat{x}_o^{\text{OP2}} \quad (2.32)$$

3. Scales the difference with a hyperparameter ( $\lambda$ ) to avoid vanishing gradients. Authors indicate an inverse relation between  $\lambda$  and the difference of the input values which can be used to set the  $\lambda$  value (Faber and Wattenhofer, 2020, Figure 5).

$$\widehat{x}_o = \lambda \cdot \widehat{x}_o \quad (2.33)$$

4. Calculates the sign bit ( $\widehat{x}^\pm$ ) and zero bit ( $\widehat{x}^0$ ) of the difference value by using smooth continuous functions.

$$\widehat{x}_o^\pm = \tanh(\widehat{x}_o) \quad (2.34)$$

$$\widehat{x}_o^0 = 1 - 2 \tanh(\widehat{x}_o)^2 \quad (2.35)$$

5. Learns a scale value (for each bit) and shared shift value.
6. Applies the scale and shift to the bit values, takes the sum of the results and passes the result through a sigmoid. The resulting value represents the probability of the comparison being true/false.

$$z_o = \widehat{x}_o^\pm \cdot W_{i,o}^\pm + \widehat{x}_o^0 \cdot W_{i,o}^0 + b_o \quad (2.36)$$

$$y_o = \text{sigmoid}(z_o) \quad (2.37)$$

7. Returns as output the comparison value and its negation value ( $1 - y_o$ ).

Given two inputs (relevant for the comparison), the NSR will compute the sign and zero bit of the difference of the two operands. The sign and zero bit definitions are continuous relaxations of the discrete definitions which rescale the bounds to avoid the gradients of partial derivatives becoming zero. That is

$$\widehat{x}_o^\pm = \begin{cases} 1 & \text{if } \widehat{x}_o > 0 \\ 0 & \text{if } \widehat{x}_o = 0 \\ -1 & \text{if } \widehat{x}_o < 0 \end{cases} \quad \text{and} \quad \widehat{x}_o^0 = \begin{cases} 1 & \text{if } \widehat{x}_o = 0 \\ -1 & \text{if } \widehat{x}_o \neq 0 \end{cases}.$$

To improve robustness to different initialisations, the NSR also implements redundancy which learns multiple independent operand pairs ( $\widehat{x}_o^{\text{OP1}}, \widehat{x}_o^{\text{OP2}}$ ) in parallel for each output element. Each pair will have its own sign and zero bit, hence learning its own set of scale and shift values. These independent paths get aggregated together by summing the different  $z_o$ 's together.

## 2.3 NALU’s Shortcomings and Existing Solutions

The development of NALMs after the NALU has focused on improving its various shortcomings. In this section, we detail the weaknesses of NALU and explain existing solutions. We focus on the arithmetic NALMs: iNALU, NAU, NMU and NPU when looking at solutions. Table 2.2 summarises the discussed issues and proposed solutions.

TABLE 2.2: Summarised NALU shortcomings and existing proposed solutions.

Short-coming \ NALM	NAU/NMU	iNALU	NPU/Real NPU	CalcNet (G-NALU)
<b>NAC<sub>•</sub> cannot have negative inputs/targets</b>	NMU: Remove log-exponent transformation	Sign correction (mixed sign vector)	Sign retrieval	
<b>Convergence of gate parameters</b>	Stacking instead of gating	Independent gating, separate weights per sub-unit and regularisation loss	-	-
<b>Fragile initialisation</b>	Theoretically valid initialisation scheme	Reinitialise model	-	-
<b>Weight inductive bias of <math>\{-1,0,1\}</math> not met (non-discrete solutions)</b>	Regularisation loss term and clipping	Regularisation loss term	(see below)*	-
<b>Gradient propagation</b>	Linear weight matrix	NAC <sub>•</sub> clip and gradient clip	Relevance gating	Replace sigmoid and tanh exponent’s with golden ratio
<b>Singularity (values close to 0)</b>	NMU: Remove log-exponent transformation	NAC <sub>•</sub> clip	Complex space transformation and relevance gating	-
<b>Compositionality</b>	-	-	-	Parsing algorithm

\* The NPU and Real NPU supports fractional weights (e.g., 0.5 representing square-root) and therefore do not enforce discretisation.

### 2.3.1 Negative Inputs and Negative Outputs

The multiplicative module of the NALU (the  $\text{NAC}_\bullet$ ) is unable to process negative inputs and produce negative outputs. The issue lies with Equation 2.3, which requires converting negative inputs into their positive counterparts because the log transformation cannot evaluate negatives. The use of an exponent also causes the inability to have negative outputs, as the range of an exponent is  $\mathbb{R}_{>0}$ . As the sign of the input is ignored and never recovered, the  $\text{NAC}_\bullet$  is unable to have negative output values. To allow for negative outputs, a module can incorporate logic to deal with assigning the correct sign to the output such as the iNALU's sign correction mechanism (Schlör et al., 2020) or the NPU's inherent sign retrieval (Heim et al., 2020).

The iNALU's sign correction mechanism creates a mixed sign vector ( $\mathbf{msv}$ )  $\in \mathbb{R}^{O \times 1}$ , consisting of elements  $\{-1, 1\}$  (assuming  $\mathbf{W}$  has converged to integers  $\{-1, 0, 1\}$ ), where each element represents the correct sign for each output element.<sup>3</sup> The  $\mathbf{msv}$  is element-wise multiplied to Equation 2.3 resulting in applying the relevant sign to the outputs of the multiplicative sub-unit. The  $+1 - |W_{i,o}|$  means unselected inputs ( $W_{i,o} = 0$ ) will avoid affecting the final sign value, as they will only multiply the  $\mathbf{msv}_o$  value by 1. Therefore, an alternate way to view the  $\mathbf{msv}$  is as a gating mechanism,  $\text{sign}(x_i) \cdot |W_{i,o}| + 1 \cdot (1 - |W_{i,o}|)$ , where a **on gate** ( $W_{i,o} = -1/1$ ) gives the sign and an **off gate** ( $W_{i,o} = 0$ ) returns 1.

In the case of the Real NPU, the latter half of its definition (in matrix form)  $\odot \cos(\mathbf{W}^{\text{RE}} \mathbf{k})$  can be interpreted as a sign retrieval mechanism.  $\mathbf{k}$  represents positive inputs as 0 and negative inputs as  $\pi$  assuming the gate value has converged to select the input. Assuming this convergence, the  $\mathbf{W}^{\text{RE}}$  values are  $\{1, -1\}$  representing  $\{\times, \div\}$ . Hence, the two possible outcomes from evaluating the cosine expression for a single input value are  $\cos(\pm\pi) = -1$  or  $\cos(0) = 1$  which represents the sign of the input value.

Alternatively, rather than requiring an additional step for recovering the signs, we can rethink the multiplicative architecture. It is possible to remove the log-exponent transformations from Equation 2.3 as Madsen and Johansen (2020) does for defining the NMU (Equation 2.14). This means negative targets can be expressed because the sign is no longer removed from the input but at the cost of no longer being able to model division.

### 2.3.2 Gating Parameter Convergence

The NALU gate, responsible for selection between the  $\text{NAC}_+$  and  $\text{NAC}_\bullet$  modules, is unable to converge reliably. For example, converging towards 1 when the true gate

<sup>3</sup>Notice the similarity in calculation between the NMU (Equation 2.14) and iNALU's  $\mathbf{msv}$  (Equation 2.9).

value should be 0 and visa-versa. In cases where the correct gate is selected, the NALU still fails to converge consistently to the expected discrete value (Madsen and Johansen, 2020, Appendix C.5). Another issue from partial convergence of gate (and weight) parameters is the leaky gate effect (Schlör et al., 2020), where non-discretised parameters result in the network learning non-optimal solutions. Such solutions may perform well on the interpolation data used during training but will not generalise to OOD data. This issue is also amplified when additional NALU modules are stacked.

Even when using the improved NAU and NMU networks instead of the  $\text{NAC}_+$  and  $\text{NAC}_\bullet$ , the NALU-style gating still leads to sub-optimal results. Therefore, Madsen and Johansen (2020) replace module gating with module stacking. As NALMs can learn selection logic, it is possible to stack two NALMs and have the output be the result of only one of the NALMs, similar to how gating can only select one network. Rather than removing the gating, Schlör et al. (2020) attempt to tackle the issue by using separate weights for the iNALU sub-units (Equations 2.6 and 2.7) to improve convergence, and independent gating (removing  $x$  from Equation 2.4) so learning  $g$  is no longer influenced by the input (see Equation 2.11). Removing the dependence on the input removes contradictory constraints on the gating that would lead to nonoptimal solutions. Taking the example given by Schlör et al. (2020, Section 3.3.6), imagine two different input samples  $x_1 = [2, 2]$  and  $x_2 = -x_1 = [-2, -2]$  and the task of adding, i.e., calculating  $2 + 2 = 4$  for the first input and  $-2 - 2 = -4$  for the second. Assuming we use the NALU gating method, it implies that  $g = \text{sigmoid}(x_1 G) = \text{sigmoid}(x_2 G) = \text{sigmoid}(-x_1 G)$ . However, as  $x_1 \neq x_2$ , the previous statement is invalid.

### 2.3.3 Bias Considerations

The NALU architecture assumes that exact arithmetic operations only occur when parameter values are either -1, 0 or 1. However, other than the sigmoid and tanh transformations which constrain the ranges to be either  $\{0, 1\}$  or  $\{-1, 1\}$  respectively there are no biases put in place to enforce the weights to converge to the discrete values mentioned above.

The most common way of allowing these weight biases to be achieved is by adding a regularisation term for sparsity and using weight clamping (Madsen and Johansen, 2020; Schlör et al., 2020). An illustrative example of the Madsen and Johansen (2020); Schlör et al. (2020) regularisation functions are found in Figure 2.11. Madsen and Johansen (2020) use sparsity (discretisation) regularisation (Equation 2.15) to enforce the relevant biases for both the NAU  $\{-1, 0, 1\}$  and the NMU  $\{0, 1\}$ . Note that the absolute of  $W_{i,0}$ , is not necessary when using NMU. The regularisation activates and warms up over a predefined period of time to avoid overpowering the main mean squared error loss term (Equation 2.16). Clamping is also applied to the weights beforehand to the ranges of the desired biases. The iNALU uses a piece-wise function (Equation 2.10) for

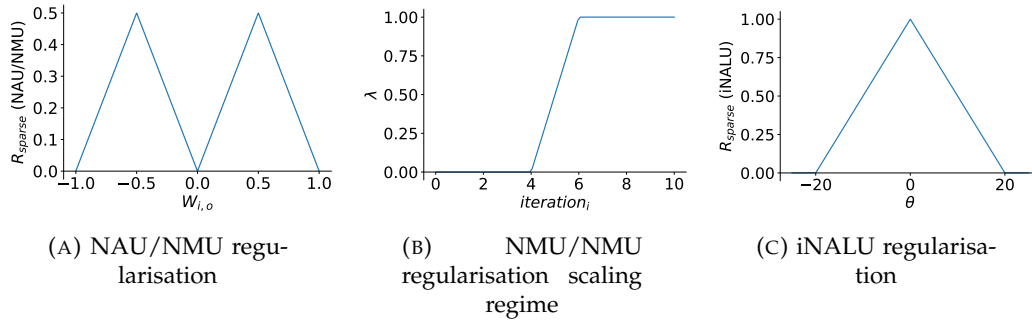


FIGURE 2.11: Regularisation functions used to induce sparsity in learnable parameters. Left: Sparsity regularisation used on the NAU and NMU (see Equation 2.15), forcing values towards  $\{-1, 0, 1\}$ . Middle: Scaling function (Equation 2.16) to control the importance of the sparsity regularisation for the NAU and NMU. For this example,  $\hat{\lambda}$  is set to 1 and the scale factor will grow between iterations 4 to 6. Right: Sparsity regularisation for a single parameter used on the iNALU (see Equation 2.10).

regularisation on weight ( $\widehat{W}^A$ ,  $\widehat{M}^A$ ,  $\widehat{W}^M$ ,  $\widehat{M}^M$ ) and gate ( $g$ ) parameters to encourage discrete values that do not converge to near-zero values. Intuitively, this regularisation penalises the parameter to encourage it to move towards  $-t$  or  $t$ . Therefore, by having a large positive/negative value, when the parameter goes through a sigmoid or tanh activation (see Equations 2.6, 2.7 and 2.11), the resulting value will be close to  $\{-1, 0, 1\}$ . Rather than a warmup period, regularisation occurs only once the loss is under a predefined threshold and stops once a discretisation threshold  $t = 20$  is met.

The NALMs studied in this thesis are designed to compute precise arithmetic and have been used in tasks requiring parameters to be discrete (and in many cases sparse) which is encouraged via regularisation. However, this may be considered a harsh bias and masking issues regarding robustness during learning. Relaxing the need for precise calculations and introducing architectures which do not expect discrete weights may provide a tradeoff between reducing precision of computation and improving robustness of results to different training data and initialisations. Investigating this tradeoff is left as future work.

### 2.3.4 Initialisation Considerations

Good initialisations are crucial for convergence. However, the NALU weight initialisations are unable to meet the desired criteria for typical neural networks as proven by Madsen and Johansen (2020) who show both the  $NAC_+$  and  $NAC_\bullet$  are unable to satisfy the guarantees of good learning.

Following Glorot and Bengio (2010), we want the expected output of the NALU's subunit  $a_o$  at initialisation to be zero i.e.,  $E[a_o] = 0$  in order to prevent exploding outputs/-gradients. Taking the definition of the  $NAC_+$  :  $a_o = \sum_{i=1}^I (W_{i,o} \cdot x_i)$ , notice how in order for  $E[a_o] = 0$  we must set the  $E[W_{i,o}] = 0$ . Since  $W_{i,o} = \tanh(\widehat{W}_{i,o}) \odot \text{sigmoid}(\widehat{M}_{i,o})$ , we

want  $E[\tanh(\widehat{W}_{i,o})] = 0$  so that  $E[W_{i,o}] = 0$ . Doing so has the additional benefit that the selection of operations within a sub-unit is unbiased. In other words, there is no preference in selecting  $+$  and  $-$  or  $\times$  and  $\div$ . However, with such an initialisation scheme, the expectation for the weight's gradient becomes zero since

$$\begin{aligned}
 E\left[\frac{\partial \mathcal{L}}{\partial \widehat{M}_{i,o}}\right] &= E\left[\frac{\partial \mathcal{L}}{\partial W_{i,o}}\right] E\left[\frac{\partial W_{i,o}}{\partial \widehat{M}_{i,o}}\right] \\
 &= E\left[\frac{\partial \mathcal{L}}{\partial W_{i,o}}\right] E\left[\tanh(\widehat{W}_{i,o})\right] E\left[\sigma'(\widehat{M}_{i,o})\right] \\
 &= E\left[\frac{\partial \mathcal{L}}{\partial W_{i,o}}\right] \times 0 \times E\left[\sigma'(\widehat{M}_{i,o})\right] \\
 &= 0.
 \end{aligned} \tag{2.38}$$

As for the NALU's other sub-unit, the NAC $\bullet$ , begin by assuming that weights can be initialised such that  $E[m_o] = 0$  (Glorot and Bengio, 2010) and that inputs are uncorrelated. Using the second order Taylor approximation the expectation of NAC $\bullet$  is estimated as:

$$E[m_o] \approx \left(1 + \frac{1}{2} \text{Var}[W_{i,o}] \log(|E[x_i]| + \epsilon)^2\right)^I \Rightarrow E[m_o] > 1, \tag{2.39}$$

implying that satisfying  $E[m_o] = 0$  and therefore well initialised weights cannot be achieved for the NAC $\bullet$  (Madsen and Johansen, 2020, Appendix B.3.1).

Empirical results support these claims, showing difficulty in both optimisation and robustness for the NALU. In such cases, we assume the Madsen and Johansen (2020) implementation of NALU is used for initialisation, where weight matrices are from a Uniform distribution with the range calculated from the fan values,<sup>4</sup> and the gate matrix from a Xavier Uniform initialisation with a sigmoid gain.<sup>5</sup> Such fragility in optimisation results in convergence to the expected parameter value being difficult to achieve, especially when redundant inputs that require sparse solutions exist. When such redundancy exists, non-sparse solutions are not extrapolative, lacking transparency. In contrast to the NALU's sub-units, the NAU (which is a linear layer) can be initialised using the Xavier Uniform initialisation (Glorot and Bengio, 2010) and the NMU can be initialised such that  $E[m_o] \approx (\frac{1}{2})^I$  which becomes zero as the number of inputs (I) approaches infinity.

To ease optimisation, Madsen and Johansen (2020) use a linear weight matrix construction (removing the need for non-linear transformations), while Schlör et al. (2020) use clipping on the NAC $\bullet$  calculation (see Equation 2.8). The minimum of the input is

<sup>4</sup>[https://github.com/AndreasMadsen/stable-nalu/blob/2db888bf2dfcb1bba8d8065b94b7dab9d178332/stable\\_nalu/layer/nac.py#L22](https://github.com/AndreasMadsen/stable-nalu/blob/2db888bf2dfcb1bba8d8065b94b7dab9d178332/stable_nalu/layer/nac.py#L22)

<sup>5</sup>[https://github.com/AndreasMadsen/stable-nalu/blob/2db888bf2dfcb1bba8d8065b94b7dab9d178332/stable\\_nalu/layer/\\_abstract\\_nalu.py#L90](https://github.com/AndreasMadsen/stable-nalu/blob/2db888bf2dfcb1bba8d8065b94b7dab9d178332/stable_nalu/layer/_abstract_nalu.py#L90)



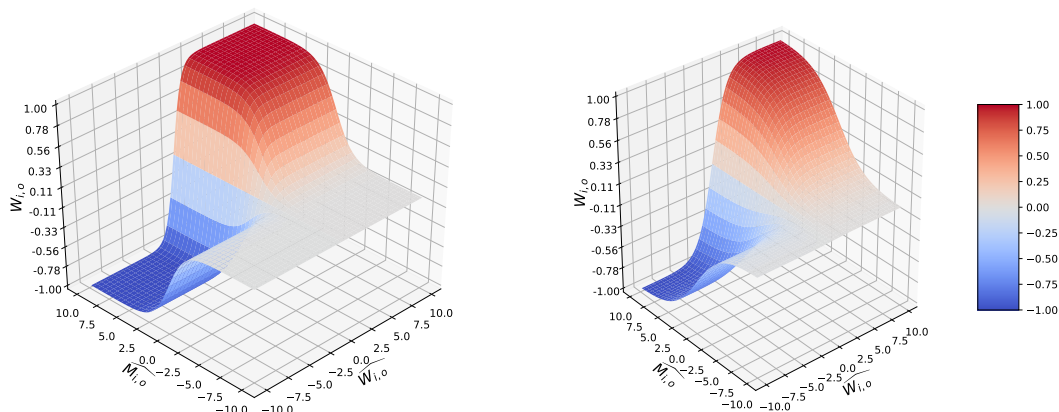


FIGURE 2.12: Adapted from Rana et al. (2019, Figure 2 and Figure 4) for showing the values for  $W$  used in NALU calculated over the domain of  $\widehat{W}$  and  $\widehat{M}$ . Left: Using NALU's calculation of  $W$  where tanh and sigmoid are calculated with base e. Right: G-NALU's calculation of  $W$  where tanh and sigmoid are calculated with a golden ratio base resulting in smoother value transition.

clipped to  $\epsilon = 10^{-7}$  and the result of the  $\ln$  operation is clipped to be at most  $\omega = 20$ . Using this clipping allows avoiding exploding intermediary results. Additionally, gradient clipping is used to avoid exploding gradients.

Rana et al. (2019) modify the non-linear activations of the weight matrices in the NALU for smoother gradient propagation, as shown by Figure 2.12, to try to avoid sub-optimal convergence. In contrast, in attempts to avoid falling/remaining in a local optima, the iNALU allows multiple reinitialisations of a model during training to counteract the non-optimal initialisation in NALU which contribute to vanishing gradients and convergence to local minima. Reinitialisation occurs every  $m^{th}$  epoch if the following two conditions are met: (1) the loss has not improved in the last  $n$  steps, (2) the loss is larger than a pre-defined threshold. The main disadvantage with reinitialising multiple times during training is that it can require running more iterations which may be infeasible. For example, for a standalone NALM it is possible to keep reinitialising until a reasonable solution is found, however if the NALM is used as a subcomponent in a larger neural network then reinitialisation can be too costly. Through a grid search, they find having the mean of the gate and NALU weight matrices  $\widehat{M}$ ,  $\widehat{W}$  initialised to be 0, -1 and 1 respectively, results in the most stable modules. However, even when using such initialisations, the stability problem remains for division (Schlör et al., 2020, Table 1).

### 2.3.5 Division

Division is the NALU's weakest operation (Trask et al., 2018). Having both division and multiplication in the same module causes optimisation difficulties. Madsen and Johansen (2020) highlight the singularity issue (caused by division by 0 or values close to 0 bounded by an epsilon value) in the NAC, which causes exploding outputs (see

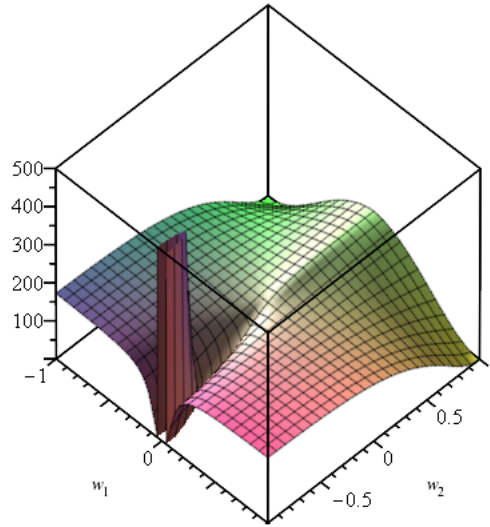


FIGURE 2.13: Taken from [Madsen and Johansen \(2020, Figure 2b\)](#). An example illustration of the unstable optimisation issue arising when using a stacked  $\text{NAC}_+$   $\text{NAC}_\bullet$  with  $\epsilon = 0.1$ . The plot represents the root mean squared loss surface when modelling  $(x_1 + x_2) \cdot (x_1 + x_2 + x_3 + x_4)$  for the input  $[1, 1.2, 1.8, 2]$ .  $w_1$  and  $w_2$  represent the weight values to use for the  $\text{NAC}_+$  and  $\text{NAC}_\bullet$  weight matrices such that

$$\mathbf{W}_1 = \begin{bmatrix} w_1 & w_1 & 0 & 0 \\ w_1 & w_1 & w_1 & w_1 \end{bmatrix} \text{ and } \mathbf{W}_2 = [w_2 \ w_2].$$

Figure 2.13). The NMU removes the use of log, therefore is not epsilon bound but is only designed for multiplication. The NPU takes [Madsen and Johansen \(2020\)](#)'s interpretation of multiplication (using products of power functions) but applies it in a complex space enabling division and multiplication ([Heim et al., 2020](#)). Though the NPU cannot fully solve the singularity issue as a log transformation is still applied to the inputs, the relevance gating (see Equation 2.18) aids in smoothing the loss surface to

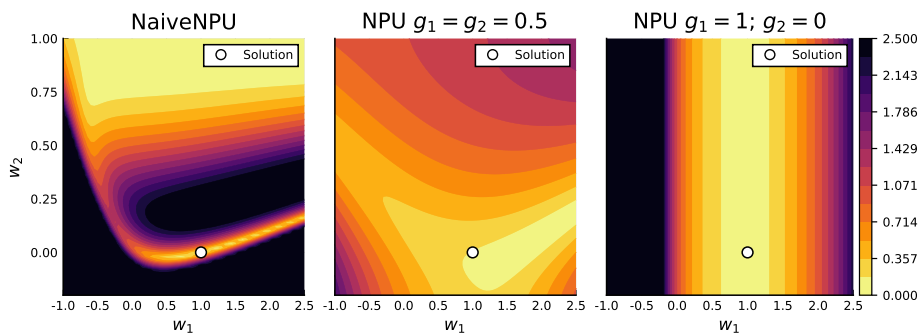


FIGURE 2.14: Taken from [Heim et al. \(2020, Figure 3\)](#). Gradient norm  $\|\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{\text{RE}}}\|$  of the NaiveNPU (NPU without relevance gating) and the NPU for the task of selecting the first element of a two element input. The correct solution is  $w_1 = 1$  and  $w_2 = 0$ . The NaiveNPU has a large zero gradient region for  $w_2 > 0.75$  unlike the NPU. The gates are initialised to  $g_1 = g_2 = 0.5$  (middle plot) which after training converge to  $g_1 = 1$  and  $g_2 = 0$  (right plot).

provide better convergence (see Figure 2.14). [Schlör et al. \(2020\)](#) observe that reinitialising modules numerous times during training can still lead to failure, implying that the issue lies in unit architecture as well as initialisation. Hence, division remains an open issue.

### 2.3.6 Compositionality

A single NALU is unable to output expressions whose operations are from both  $\{+, -\}$  and  $\{\times, \div\}$ , for example  $x_1 + x_2 \times x_3$ . Though it is possible to stack the NALUs to increase the expressibility, having each module learn reliably remains a challenge as currently even a one layer NALU cannot learn a single operation well. [Rana et al. \(2019\)](#) develop CalcNet, a parsing algorithm, such that the expression to learn is decomposed into its intermediary sub-expressions which obey the rules of precedence (i.e., BIDMAS) and then is solved in a compositional manner. However, decomposition requires fixed rules and pre-trained sub-units which are undesirable because in order to decompose, the input must also contain the operations used, meaning that the model is exposed to a priori relating to the underlying function.

## 2.4 Applications of the NALU

This section describes the uses of NALU as a sub-component in architectures to tackle practical problems outside the domain of solving arithmetic on numeric inputs. Success and failure cases are mentioned. We choose to focus on NALU applications on the basis that the improved modules discussed above can be applied in place of NALU to provide additional performance gains to the mentioned applications.

### 2.4.1 Existing Applications

[Xiao et al. \(2020\)](#) insert a NALU layer between a two-layer Gated Recurrent Unit (GRU) and dense layer to predict vehicle trajectory of complex road sections (containing constantly changing directions). NALU improves extrapolation capabilities to deal with abnormal input cases outside the range of the GRU hidden states output.

[Xiao et al. \(2021\)](#) use the NALU to improve the numerical extrapolation ability in predicting the aggregation of (stationary) private cars on weekends. During weekends the patterns for car aggregation are much less predictable, therefore having the ability to generalise out of the training range is useful. The input into the NALU is the combined representations of the spatial, temporal and external features from which linear relations can be inferred. [Zhang et al. \(2022\)](#) also use the NALU's ability to capture linear relationships to help with extrapolation in predicting the stay time for private cars.

Such predictions are useful in helping sales by recommending points of interest, prediction of traffic conditions and road congestion. The full architecture is a MLP encoder, a NALU ‘exception’ module and a MLP decoder (with dropout to avoid overfitting from data imbalance). Stay locations are random and sparse which makes prediction challenging, however some amount of linear relationship exists between a person’s stay behaviour and their stay time implying the NALU can be useful. The features are both temporal and spatial (since spatial alone is too sparse and random). They find for the short-stay settings, the improvement from using their NALU based model is small but for longer-stay prediction, where extrapolative abilities are required, the improvement is more substantial. [Jiang et al. \(2021\)](#) create a framework for location based services for user destination prediction where they can trade-off between preserving location privacy (by adding injected noise) and utility (usefulness of their prediction). The framework combines the NALU, an MLP and GRUs to extract features to predict the area under the curve (AUC) of the privacy and utility metrics/losses. In particular, the NALU is used to capture linear relationships between the model and privacy AUC.

[Raj et al. \(2020\)](#) combine  $NAC_+$  modules before LSTM cells for *fast* training in the extraction of temporal features to classify videos for badminton strokes. They further experiment in using  $NAC_+$  modules with a dense layer to learn temporal transformations, finding better performance than the LSTM based module and the dense modules being quicker to train. They justify the use of the  $NAC_+$  as a way to produce sparse representations of frames, as non-relevant pixels would not be selected by the  $NAC_+$  resulting in 0 values, while relevant pixels accumulate.

[Zhang et al. \(2019a\)](#) use deep Reinforcement Learning (RL) to learn to schedule views on content-delivery-networks (CDNs) for crowdsourced-live-streaming (CLS). NALU’s extrapolative ability alleviates the issue of data bias (which is the failure of models outside the training range) by using the NALU to build an offline simulator to train the agent when learning to choose actions. The simulator is composed of a two layer LSTM with a NALU layer attached to the end. [Zhang et al. \(2019b\)](#) propose a novel framework (named Livesmart) for cost-efficient CLS scheduling on CDNs with a quality-of-service (QoS) guarantee. Both components required in Livesmart contain models using NALU. The first component (named new viewer predictor) uses a stacked LSTM-NALU to predict workloads from new viewers. The second component (named QoS characterizer) predicts the QoS of a CDN provider. This component uses a stack of CNNs, a LSTM and a NALU. Both components use the NALU’s ability to capture OOD data to aid in dealing with rare events/unexpected data.

[Wu et al. \(2020\)](#) combines layers of  $NAC_+$  to learn to do addition and subtraction on vector embeddings to form novel compositions for creating analogies. Modules are applied to the output of an attention module (scoring candidate analogies) that is passed through a MLP. The output of the  $NAC_+$  modules is passed to a LSTM producing the final analogy encoding.

The NALU has also been used with CNNs. [Rajaa and Sahoo \(2019\)](#) applies stacked NALUs to the end of convolution units to predict future stock prices. [Rana et al. \(2020\)](#) utilises the  $NAC_+$ /NALU as residual connection modules to larger convolutional networks such as the U-Net and a fully convolutional regression network for cell counting in images. Such connections enable better generalisation when transitioning to data with higher cell counts to the training data. However, no observations are made to what the units learn to cause an improvement in cell counting over the baseline models.

[Chennupati et al. \(2020\)](#) uses the NALU as part of a larger architecture to predict the runtime of code on different hardware devices configured using hyperparameters. The NALU predicts the reuse profile of the program, keeping track of the count of memory references accessed in the execution trace. The NALU outperforms a genetic programming approach for doing such a prediction.

[Teitelman et al. \(2020\)](#) explores the problem domain of cloning black-box functionality in a generalisable and interpretable way. A decision tree is trained to differentiate between different tasks of the black box. Each leaf of the tree is assigned a neural network comprising of stacked dense layers with a NALU layer between them. Each neural network is able to learn the black-box behaviour for a particular task. Like [Xiao et al. \(2020\)](#), results showed that NALU is required to learn the more complex tasks.

Finally, [Sestili et al. \(2018\)](#) suggests the NALU has potential use in networks that predict security defects in code. This is due to the module's ability to work with numerical inputs in a generalisable manner, instead of limiting the application to be bound to a fixed token vocabulary requiring lookups.

## 2.4.2 Applications Where NALU Is Inferior

We now discuss examples of situations in which NALU modules are a sub-optimal architecture choice for applicational settings. [Madsen and Johansen \(2020\)](#) show that the NAU/NMU outperforms the NALU in the MNIST sequence task for both addition and multiplication. [Dai and Muggleton \(2020\)](#) show the arithmetic ability (named background knowledge) of the NALU is incapable of performing the MNIST task for addition or products when combined with an LSTM. Instead, they show a neural model for symbolic learning which learns logic programs using pre-defined rules as background knowledge can perform with over 95% accuracy. However, we question whether the failure is a result of the NALU or due to the misuse of its abilities by combining it with an LSTM. For example, as the inputs are images, unless the LSTM converts each image into a numerical value that can be processed by the NALU in an arithmetic way it can be suggested that the LSTM is completing the task without the numerical capabilities of the NALM. [Jacovi et al. \(2019\)](#) show that in black box cloning for the [Trask et al. \(2018\)](#)

MNIST addition task, their EstiNet model which captures non-differentiable models outperforms NALU. Though it can be argued that a more relevant comparison would test the  $NAC_+$  or the NAU which are solely designed for addition. Joseph-Rivlin et al. (2019) show that although the  $NAC_\bullet$  can learn the order for a polynomial transformation to high accuracy, it is still outperformed by a pre-defined order two polynomial model. Results suggest that the  $NAC_\bullet$  may not have fully converged to express integer orders. Dobbels et al. (2020) found the NALU was unable to extrapolate for the task of predicting far-infrared radiation fluxes from ultraviolet-mid-infrared fluxes. Though no clear reason was stated, the lack of extrapolation could be attributed to the co-dependence of features because of applying fully connected layers prior to the module. Jia et al. (2020) considers the NALU as a hardware component concluding that the NALU has too high an area and power cost to be feasible for practical use. Implementing for addition costs 17 times the area of a digital adder, and the memory requirements for weight storage are energy inefficient for doing CPU operations.

## 2.5 Discussion: Remaining Gaps

This section discusses areas that remain to be fully addressed for NALMs. The identified areas are a compilation of current weaknesses in the field which we have observed in this chapter. The areas include *benchmarks*, *division*, *robustness*, *compositionality*, and *interpretability of more complex architectures*, which will become the focus of the remaining chapters of this thesis.

Having **benchmarks** is important in allowing for reliable comparison between modules. Such benchmarks should include a simple synthetic dataset which we detail in this work and a real-world data benchmark (which remains to be created) with a systematic evaluation.

One goal of these modules is to be able to extrapolate. To achieve this, a module should be **robust** to being trained on any input range. Madsen and Johansen (2020) show that modules are unable to achieve full success of all tested ranges (with the stacked NAU-NMU failing on a training range of [1.1,1.2], being unable to obtain a single success). Reinitialisation of weights (Schlör et al., 2020) during training could provide a solution, however this seems to be unlikely given Madsen and Johansen (2020) tests against 100 model initialisations and using reinitialisation for a NALM that is part of a large end-to-end network may not be economical.

**Division** remains a challenge. From reviewing existing NALMs, no module has been able to reliably solve division. Currently, the NPU by Heim et al. (2020) is the best module to use, though it would struggle with input values close to zero. Madsen and Johansen (2020) argues modelling division is not possible due to the singularity issue. One suggestion for dealing with the zero case is to take influence from Reimann and

Schwung (2019) which can have an option for showing an output that is invalid (or in their case all off values).

**Compositionality** is desirable. A model should be flexible, having the option to select different types of operations and model complex mathematical expressions. Currently, the two popular approaches are gating and stacking. Gating has been found to not work as expected and gives convergence issues. Stacking, though more reliable, has fewer options in operation selection than gating. Deep stacking of modules (in a non-recurrent fashion) remains untested.

Finally, it remains to be understood **how modules influence learning of other modules** (such as recurrent networks and CNNs) in their representations. Such architectures which require combining NALMs with other families of neural networks can be considered as another form of compositionality. This is an area for future work and is not addressed as part of this thesis. For example, this can involve understanding if representations are more interpretable because of being trained with a NALM, or whether the emergence of any useful biases occurs in the encoding of the generic modules.





## Chapter 3

# Benchmarking Existing and Future Models

Having access to well-defined benchmarks is an important step in unifying the NALM field. Standardised benchmarks with consistent model implementations allow for reliable cross-comparisons of different works and improve the speed of progress in their respective fields. For ML in general, the Hugging Face platform provides a plethora of open source implementations with 60K models and 6K datasets<sup>1</sup> for domains including audio, text, vision and RL (HuggingFace, 2022). To encourage common knowledge for models, they also provide model cards (Mitchell et al., 2019) which contain metadata regarding the model, its uses, ethical considerations, reproducibility, and results. For symbolic regression,<sup>2</sup> there exists SRBench, a reproducible benchmark that encourages cross-pollination from the ML and GP research fields that consist of 252 (122 black-box and 130 synthetic) datasets (Cava et al., 2021). The SRBench is described as a “living benchmark” (SRBench, 2023), encouraging new models to be evaluated on (Kamienny et al., 2022) and ways to be evaluated (Dick, 2022).

In contrast, currently, no de facto method exists for measuring arithmetic extrapolation performance on NALMs. Therefore, we dedicate this chapter to setting what we believe is a good foundation for evaluating NALMs. We begin by focusing on a popular two-layer arithmetic task setup. This task is used as a case study to understand the existing evaluation techniques used in the field and to highlight the existing inconsistencies across papers, hence encouraging the need for task standardisation. We end this section by introducing our *Single Module Arithmetic Task* to act as a standardised benchmark for NALMs. We provide empirical results, comparing existing arithmetic NALMs and provide open source code (MIT license) at <https://github.com/bmistry4/nalm-benchmark>.

---

<sup>1</sup><https://huggingface.co/docs/hub/index>

<sup>2</sup>Symbolic Regression is the task of discovering the underlying mathematical expression from data.

As NALMs are arithmetic specialists which can extrapolate, we consider synthetic arithmetic baselines allowing for control over the input data to easily create OOD data sets for testing extrapolation. One example of a baseline task is a single-layer task which determines how well a NALM can perform a single operation. To increase complexity of the task, a two-layer task can be considered, which introduces redundant inputs and multistep operations. The two-layer task tests the ability to select from a large sized input containing multiple irrelevant inputs and apply exact operations in a particular order to relevant data items. The chosen evaluation metrics take inspiration from the NALM definition given in Chapter 2 which states *NALM expressions extrapolate to OOD data when parameters are appropriately learnt whilst expressing an interpretable solution*. Specifically, the metrics are designed specifically to measure a NALMs success, speed of convergence, and interpretability of the solution. The success metric measures the NALMs ability to learn an extrapolative solution which is determined based on an error threshold. As NALMs are specialised to learning arithmetic we expect finding the expected solution to be fast. Therefore, we also evaluate NALMs based on their speed to discover an extrapolative solution. Finally, as NALMs have transparent architectures, we want to confirm that the solution is interpretable. As the proposed baselines require exact arithmetic (without scaling coefficients) we know that the required parameters for extrapolative solutions have discrete weights. Therefore, we can measure how discrete weights are via a sparsity error to measure interpretability.

### 3.1 Two Layer Arithmetic Task

A task consistently used to test NALMs is the ability of a module to learn a two-operation function.<sup>3</sup> This was first introduced as the ‘*Static Simple Function Learning*’ task by Trask et al. (2018) but has been developed since due to the lack of details towards reproducing the experiment. Madsen and Johansen (2019) introduce their own version of the experiment setup (including details for reproducibility) which they use in their later work (Madsen and Johansen, 2020) under the name ‘*Arithmetic Datasets*’ task. Specifically, given an input vector of 100 floating-point numbers, the first (addition) layer should learn to output two values (denoted a and b) which are the sums of two different partially overlapping slices (i.e., subsets) of the input, and the second layer should perform an operation on a and b. Figure 3.1 illustrates such an example, where the second operation is a multiplication. Solving this task requires stacking NALMs to learn the compositional expression. Due to the rigorous setup, evaluation metrics, and available code, we strongly suggest the Madsen and Johansen (2019) experiment be used to test and compare new modules for the Two Layer Arithmetic task.

---

<sup>3</sup>An overview of other types of experiments are given in Appendix D.

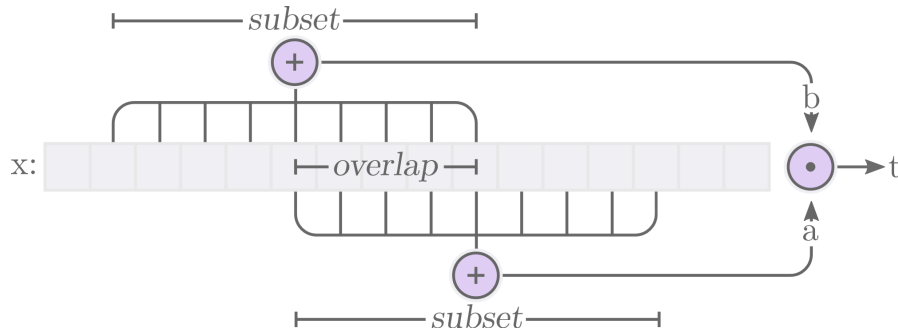


FIGURE 3.1: Taken from Madsen and Johansen (2020, Figure 6). Illustration on how to get from input vector to the target scalar for the Arithmetic Dataset Task. This setup is solved using a stacked addition-multiplication module.

Other works such as the iNALU’s experiment 4 (*‘Influence of Initialization’*) and 5 (*‘Simple Function Learning Task’*) also use this task but have a different setup to Madsen and Johansen (2020). The experiments calculate  $a$  and  $b$  differently by not allowing for overlap between  $a$  and  $b$ , and allowing  $a$  and  $b$  to be made up of random (instead of consecutive) elements of the input. Also, the iNALU uses different interpolation and extrapolation ranges. Heim et al. (2020)’s claims that their *‘Large Scale Arithmetic’* task is equivalent to the *Arithmetic Dataset Task*, however, there are critical distinctions between the two meaning the results from the two papers are not directly comparable. In Table 3.1 we highlight the differences between the three experiment setups.<sup>4</sup>

## 3.2 Evaluation Metrics

The purpose of evaluation metrics is to reflect whether a model solution is the true solution and be able to rank different model solutions against each other. Trask et al. (2018) calculates a score for each model using

$$\frac{\text{MSE loss of the model}}{\text{MSE loss of a randomly initialised model (with no training)}}$$

A score of 0 reflects perfect accuracy while a score larger than 1 means the solution is worse than the baseline model. Though this method is good for relative rankings between different models, there is no indication to the relative performance against the gold solution (Schlör et al., 2020). Furthermore, a randomly initialised model will most likely have poor performance, so the scaled errors of the other models seem better than they are. Heim et al. (2020) measures the median of the MSE with confidence intervals using the median absolute deviation. Compared to the mean, the median is less sensitive to outliers and skewed results, however as a result it discards information about individual errors which can be helpful when considering factors such as the extent of

<sup>4</sup>We do not compare Trask et al. (2018) as no details on the experiment setup is given. We do not compare Rana et al. (2019) as they do not include this experiment.

TABLE 3.1: Differences in the ‘Large Scale Arithmetic’ task used in the papers Madsen and Johansen (2020) and Heim et al. (2020). ‘a’ and ‘b’ represent summed slices of the input, and are the expected output values for the addition module. \*U=Uniform, S=Sobol and TN=Truncated Normal. (The Sobol generates a quasi-random sequence of points which are distributed to approximate the Uniform distribution.)

Property	Madsen and Johansen (2020)	Heim et al. (2020)	Schlör et al. (2020)
Hidden size	2	100	2
Iterations for one run	5,000,000	50,000	100,000
Number of seeds	100	10	10
Learning rates	$10^{-3}$	$10^{-2}$ for addition and $5 \times 10^{-3}$ for all other operations	$10^{-3}$
Subset and overlap ratios	0.25 and 0.5	0.5 and 0.25 (for addition, subtraction, and multiplication)	0.33 and 0
Division	a/b	1/a	a/b
Interpolation and extrapolation ranges*	Train: U[1,2) for all operations. Test: U[2,6).	Train: S(-1,1) for addition, subtraction, and multiplication, S(0,0.5) for division. Test: S(-4,4) for addition, subtraction and multiplication, S(-0.5,0.5) for division.	Train: U[-3,3] and TN $_{(\mu=0,\sigma=1)}$ [-3,3] Test: U[-5,-5] and TN $_{(\mu=3.5,\sigma=\frac{1}{6})}$ [3,4] respectively.
Programming framework	Pytorch (Python)	Flux (Julia)	Tensorflow (Python)

robustness against different initialisations. Both Madsen and Johansen (2019) and Schlör et al. (2020) measure the MSE but also compare if the MSE is within a threshold value representing the error of an ideal solution to a given precision. This threshold comparison produces a success metric in which each seed can be compared in a pass/fail situation which is averaged to a success rate.

### 3.2.1 Evaluation metrics used on the Arithmetic Dataset Task

Madsen and Johansen (2019) extends the use of threshold-based success by using configuration-sensitive success thresholds, two additional metrics to measure the speed of convergence and sparsity, and confidence intervals for each metric where each interval calculated using a different distribution family to best match the metric. Specifically, there are three evaluation metrics: (1) the success on the extrapolation dataset against a near optimal solution (*success rate*), (2) the first iteration in which the task is considered

solved (*speed of convergence*), and (3) the extent of discretisation towards the weights' inductive biases (*sparsity error*).

A success means the MSE of the trained model is lower than a threshold value (i.e., the MSE of a near optimal solution). For the Arithmetic Dataset Task, the threshold is a simulated MSE on 1,000,000 data samples using a model where each weight of the addition is off the optimal weight value by  $\epsilon = 10^{-5}$ . A near optimal solution is used over an optimal solution as it considers accumulated numerical precision errors (a limitation of hardware rather than module architecture). The sparsity error calculated by  $\max_{i,o}(\min(|W_{i,o}|, 1 - |W_{i,o}|))$ , represents the NALM weight element which is the furthest away from the acceptable discrete weights for a NALM. For example, for the NMU, if a weight was at 0.7 it would get a sparsity error of 0.3.

Each metric is calculated over different seeds where the total number of seeds should be enough to demonstrate issues on robustness while keeping computation time reasonable. 95% confidence intervals are calculated for each metric. The success rate uses a Binomial distribution because trials (i.e., run on a single seed) are either pass/fail situations. The convergence metric uses a Gamma distribution and sparsity error uses a Beta distribution following the evaluation scheme in [Madsen and Johansen \(2019\)](#).

### 3.3 Single Module Arithmetic Task

Having a standardised benchmark is essential for fair comparison of modules. As stated previously, so far, no such benchmark exists. Therefore, we provide results on a *Single Module Arithmetic Task*, training modules on their respective operations over a range of different interpolation distributions and testing over a range of extrapolation distributions.

**Why not use the two-layered Arithmetic Dataset Task?** The Arithmetic Dataset Task requires modules to perform three sub-tasks: *selection*, *operation*, *stacking*. Selection is the ability to deal with input redundancy for both modules (though more-so for the first layer addition module). Operation is the ability to carry out the correct operation/s (i.e., addition and multiplication). Stacking sees if training can propagate through two layers. Even with only two layers, there are already multiple components being assessed in a single task, making it difficult to analyse where issues lie. Therefore, to gain a better understanding of individual NALMs, we propose an experiment that evaluates if the operation/s the module specialises in can be learned.

**Setup.** A single module is used. The input size is two and the output size is one, hence there is no input redundancy. Hence, the objective is to model:  $y = x_1 \circ x_2$  where  $\circ \in \{+, -, \times, \div\}$ . We test the: NALU, iNALU, G-NALU, NAC<sub>+</sub>, NAC<sub>•</sub>, NAU, NMU, NPU, and Real NPU. Each run trains for 50,000 iterations to allow for enough

iterations until convergence. A MSE loss is used with an Adam optimiser. Interpolation (training/validation) and extrapolation (test) ranges are presented in Table 3.2. The ranges are influenced by the ranges from Madsen and Johansen (2020) as they provide good coverage. Early stopping is applied using a validation dataset sampled from the interpolation range. A summary of experiment parameters is shown in Table 3.3. Additional module specific hyper-parameters, hardware and runtimes are found in Appendix E.1.

INTERPOLATION	EXTRAPOLATION
[-20, -10)	[-40, -20)
[-2, -1)	[-6, -2)
[-1.2, -1.1)	[-6.1, -1.2)
[-0.2, -0.1)	[-2, -0.2)
[-2, 2)	[-6, -2) $\cup$ [2, 6)
[0.1, 0.2)	[0.2, 2)
[1, 2)	[2, 6)
[1.1, 1.2)	[1.2, 6)
[10, 20)	[20, 40)

TABLE 3.2: Interpolation (train/validation) and extrapolation (test) ranges used for the Single Module Arithmetic Task. Data (as floats) is drawn from a Uniform distribution with the range values as the lower and upper bounds.

TABLE 3.3: Parameters for the Single Module Task which are applied to all modules. \*Validation and test datasets generate one batch of samples at the start which gets used for evaluation for all iterations.

Parameter	Single Module Task
<b>Layers</b>	1
<b>Input size</b>	2
<b>Subset ratio</b>	0.5
<b>Overlap ratio</b>	0
<b>Total iterations</b>	50000
<b>Train samples</b>	128 per batch
<b>Validation samples*</b>	10000
<b>Test samples*</b>	10000
<b>Seeds</b>	25
<b>Optimiser</b>	Adam (betas=(0.9, 0.999))

### 3.3.1 Evaluation Metrics

We adopt the Madsen and Johansen (2019)’s evaluation scheme used for the Arithmetic Dataset Task (explained in Section 3.2.1), but adapt the expression used to generate the predictions of an  $\epsilon$ -perfect model  $y_0^\epsilon$ . To reiterate, the purpose of using an epsilon based evaluation is to take into consideration the numerical precision errors which can

occur as well as the effect of the input ranges on the magnitude of those errors. The expression for  $y_o^\epsilon$  varies per operation and is shown below:

$$\begin{aligned} \text{Addition: } y_o^\epsilon &= (x_1 + x_2) - \left( \sum_{i=1}^I |x_i| \right) \epsilon \\ \text{Subtraction: } y_o^\epsilon &= (x_1 - x_2) - \left( \sum_{i=1}^I |x_i| \right) \epsilon \\ \text{Multiplication: } y_o^\epsilon &= (x_1 x_2) (1 - \epsilon)^2 \times \prod_{x_i \in X_{irr}} (1 - |x_i| \epsilon) \\ \text{Division: } y_o^\epsilon &= \frac{x_1 (1 - \epsilon)}{x_2 (1 + \epsilon)} \times \prod_{x_i \in X_{irr}} (1 - |x_i| \epsilon) \end{aligned}$$

Assume  $x_1$  and  $x_2$  are the relevant operands for the operation. Any remaining features ( $x_3, \dots, x_n$ ) are irrelevant to the calculation and part of the set  $X_{irr}$ . Let  $I$  denote the total number of input features. In each case, the  $\epsilon$  for each feature will contribute some error towards the prediction. A simulated MSE is then generated with an  $\epsilon = 10^{-5}$  like in Madsen and Johansen (2019) and used as the threshold value to determine if a NALM converges successfully for a particular range by comparing the NALMs extrapolation error against the threshold value. The above expressions provide the most tolerant scenario (assuming epsilon-feature error) of the error threshold to be considered a success, regardless of if the inputs are positive, negative or mixed-signs.

### 3.3.1.1 Alternative Options for Generating a Success Threshold

Though not used for this experiment, other methods can be used to generate the  $\epsilon$ -threshold. The factors which can be changed include:

- The  $\epsilon$ -perfect model, e.g., we could use a  $\epsilon$ -perfect NALU expression which uses log space.
- The comparison metric against the perfect model. A MSE is used but other metrics such as PCC or MAPE are also valid.
- The value of  $\epsilon$  to control the tolerance of the threshold. Larger values would be more tolerant while smaller values are tighter.

All these can be modified and should be considered if creating a new threshold evaluation scheme. However, the three points to be consistent on no matter the chosen evaluation method are:

1. Being task and range dependant.

2. Using the same threshold when comparing models on the same task.
3. Not making the generation of the threshold dependent on the benchmarked model.

### 3.3.2 Results

We present the NALMs' performances on the four main arithmetic operations. Each figure consists of plots for each evaluation metric (success rate, speed of convergence and sparsity error), with confidence intervals calculated over 25 seeds.

**Addition (Figure 3.2).** The NAU has full success for all ranges correlating to the sparsity errors around 0 meaning that weights successfully converge to the expected value of 1. The iNALU also has full success but takes longer to solve and has a slightly larger sparsity error than the NAU. The NALU struggles with consistent performance especially for the small positive range ( $\mathcal{U}[0.1,0.2)$ ), large positive range ( $\mathcal{U}[10,20)$ ) and range with both positive and negative inputs ( $\mathcal{U}[-2,2)$ ). The low sparsity error implies that discrete values are being converged to, though not to the correct ones. The  $\text{NAC}_+$  also struggles to obtain consistent results over different ranges like the NALU. The G-NALU performs the worst of all the modules obtaining non-zero success on only 4 of the 9 ranges.

**Subtraction (Figure 3.3).** The NAU has full success for all ranges. The solved at iterations does remain low, similar to addition, with perfect sparsity when converged. However, ranges  $\mathcal{U}[-1.2,-1.1)$  and  $\mathcal{U}[1.1,1.2)$  require over double the number of iterations to be solved compared to the rest of the ranges implying that small ranges can cause more challenging loss landscapes. The difficulty of these two ranges also holds for all other modules which have near 0 success (except the iNALU which has at least 40% success). The iNALU has full success on all ranges excluding  $\mathcal{U}[-1.2,-1.1)$  and  $\mathcal{U}[1.1,1.2)$ . Like addition, the solve speed and sparsity error of iNALU remain larger than the NAU. The

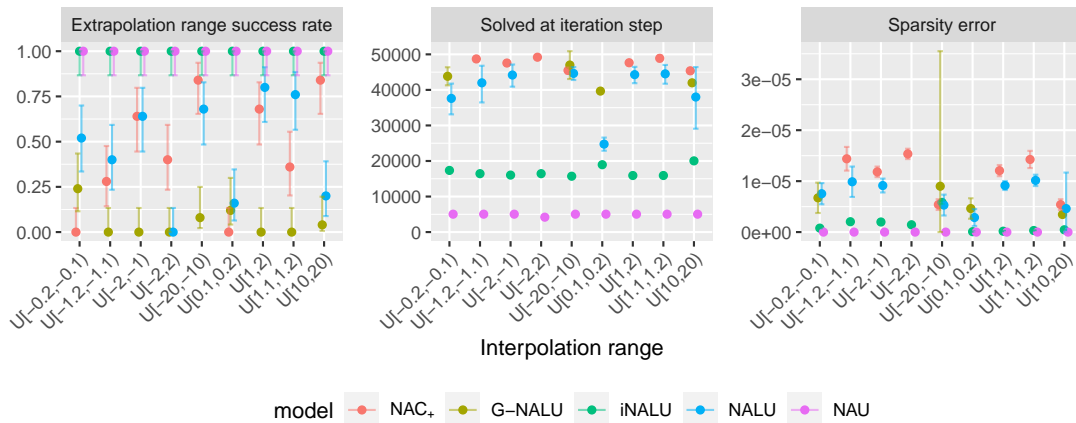


FIGURE 3.2: Performance on Single Module Task for addition.



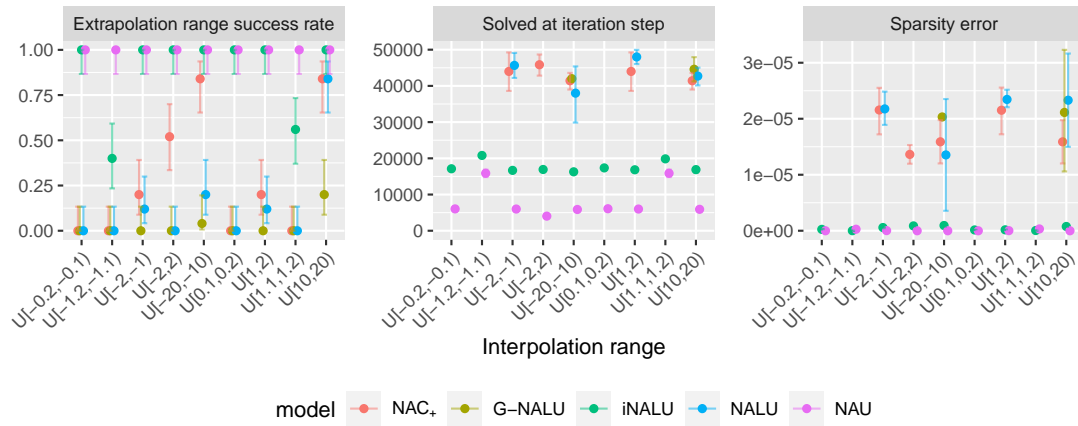


FIGURE 3.3: Performance on Single Module Task for subtraction.

NALU struggles much more with subtraction than with addition (except for  $\mathcal{U}[10,20]$ ). The NAC<sub>+</sub> outperforms NALU on 4 of the 9 ranges. The G-NALU does not outperform the NALU on any ranges.

**Multiplication (Figure 3.4).** The NMU, iNALU, NPU and Real NPU have full success on range  $\mathcal{U}[1,2]$ . The NMU struggles with some negative input ranges, i.e.,  $\mathcal{U}[-1.2,-1.1]$  and  $\mathcal{U}[-2,-1]$ . Though NPUs can process negative inputs, empirical results suggest the modules struggle to learn. The NPU and Real NPU perform the same for all ranges except one, suggesting that the problem is not complex enough to require the use of the imaginary weight matrix. However,  $\mathcal{U}[-2,2]$  is an example in which  $W_{im}$  is utilised (achieving 32% more success than the Real NPU). Even though this range allows either of the input values to be positive or negative values, the learned weights should be [1,1] for the real weights and [0,0] for the imaginary weights. The NALU can solve some ranges but no range with full success. The NAC<sub>•</sub> outperforms the NALU on the 2 ranges it has success but fails to achieve any success on the remaining 7 ranges. The iNALU outperforms the NALU on 7 ranges where it gains full success on 5 of those

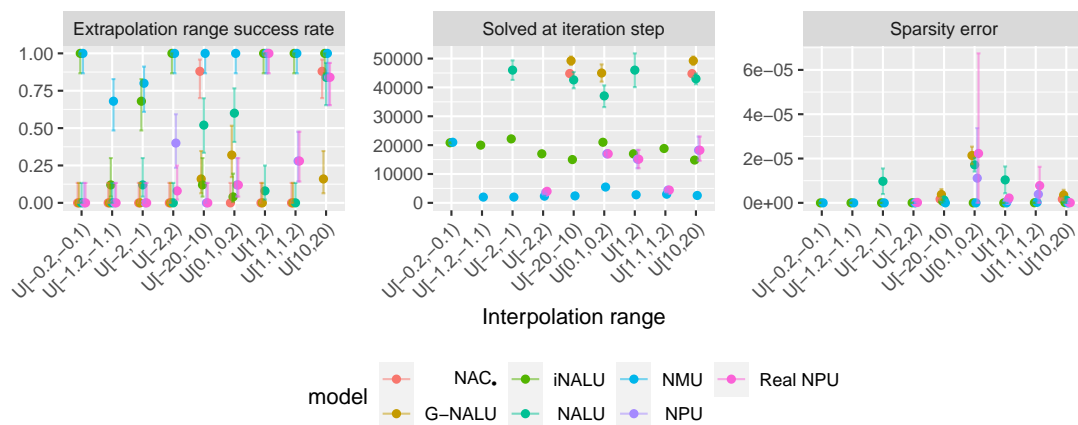


FIGURE 3.4: Performance on Single Module Task for multiplication.

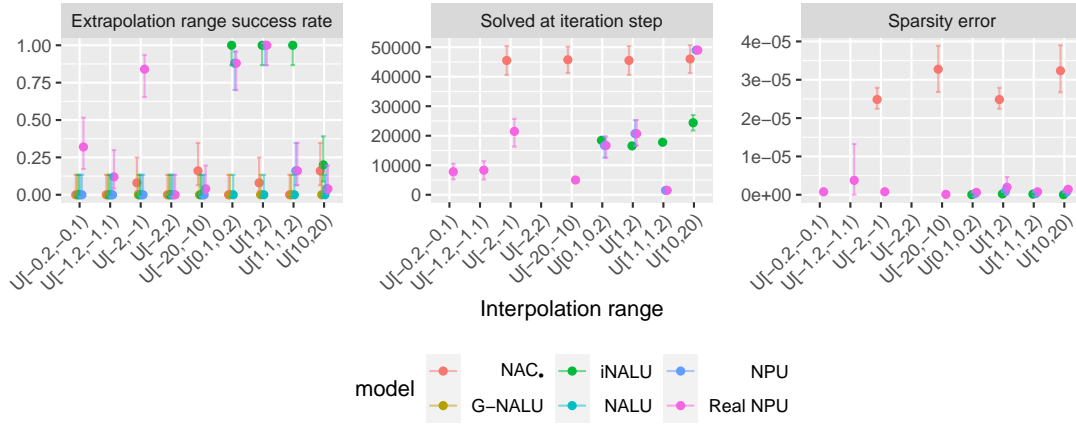


FIGURE 3.5: Performance on Single Module Task for division.

ranges. Similar to the addition and subtraction results, the G-NALU does not outperform the NALU on any ranges suggesting that golden ratio based tanh and sigmoid are not effective for learning.

**Division (Figure 3.5).** No model solves division for all ranges. The iNALU is the only module to have a success rate of 1 on any range, fully solving 3 of the 9 ranges. This highlights the difficulty in modelling division even for the simplest case, aligning with prior claims (Madsen and Johansen, 2020). The NPU and Real NPU perform perfectly for  $\mathcal{U}[1,2]$ . The Real NPU has better performance than the NPU for negative input ranges. The NAC $\bullet$  is able to achieve some success on 4 ranges while the NALU and the G-NALU cannot achieve any success on all 9 ranges. The failure on  $\mathcal{U}[-2,2]$  for the NALU and G-NALU is expected due to the inability to process mixed sign inputs caused by the modules' log-exponent transformation.

### Why is multiplication/division more challenging to learn than addition/subtraction?

A trend observed from our qualitative and quantitative analysis is that learning multiplication or division operations are significantly more difficult than addition or subtraction. As NALMs require gradient based learning, parameter updates are operator dependant, therefore, if the partial derivatives of the operations are considered (see Table 3.4) the following point becomes apparent. Addition and subtraction have constant derivatives whereas multiplication and division derivatives are reliant on the inputs (including elements you are not differentiating with respect to). This gives intuition

TABLE 3.4: Partial derivatives for a two-input single arithmetic operation task.

Expression	$\frac{\partial y}{\partial a}$	$\frac{\partial y}{\partial b}$
$y = a + b$	1	1
$y = a - b$	1	-1
$y = a \times b$	b	a
$y = \frac{a}{b}$	$\frac{1}{b}$	$-\frac{a}{b^2}$

as to why multiplication and division have difficulties in robustness to different training ranges. Even though architectures such as the NALU use a log-exponent transformation to convert multiplication/division into addition/subtraction operations in logarithmic space, the expression is still equivalent to a multiplication and division, meaning the gradients will still be input dependent.

**Testing limits of full success modules.** Achieving full success on all the ranges for an operation only occurred three times - the iNALU for addition and the NAU in addition and subtraction. To determine to what extent this holds, we experiment with introducing redundant units to the input, resulting in an increase in the task difficulty.

The NAU fails at 10 inputs (8 redundant inputs) for addition (Figure 3.6) but remains successful for subtraction. To discover failures for subtraction, we significantly increase the number of inputs to 100 (98 redundant inputs) (Figure 3.7). This results in failure cases for  $\mathcal{U}[-1.2,-1.1)$  and  $\mathcal{U}[1.1,1.2)$  which match the failure ranges for addition. The

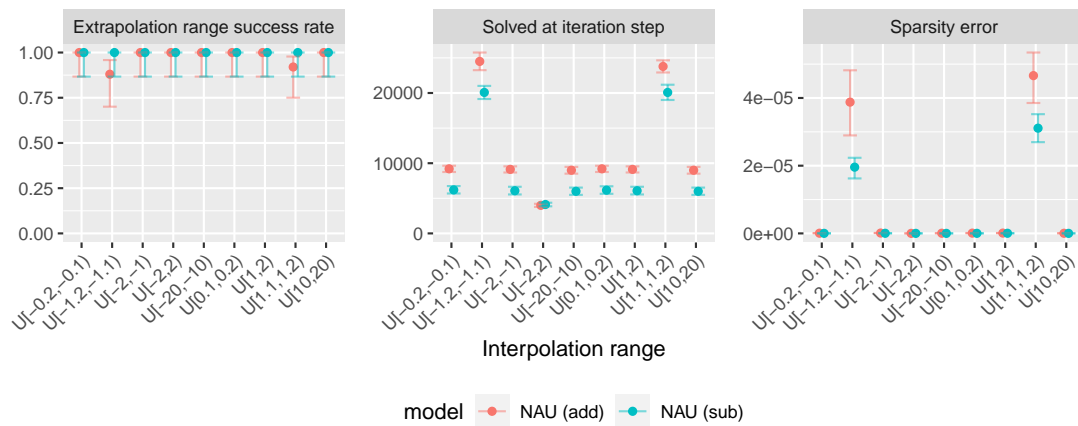


FIGURE 3.6: NAU failures on the Single Module Task for addition and subtraction with 10 inputs (8 redundant inputs).

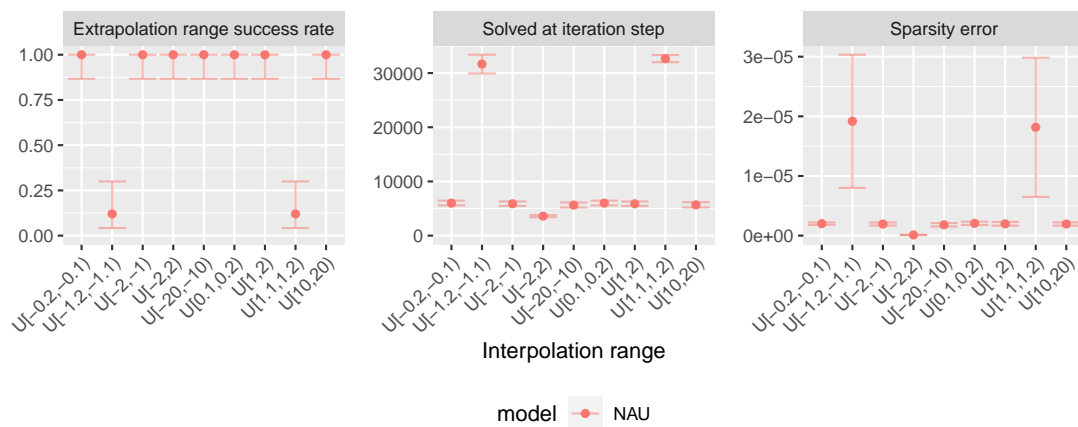


FIGURE 3.7: NAU failures on the Single Module Task for subtraction with 100 inputs (98 redundant inputs).

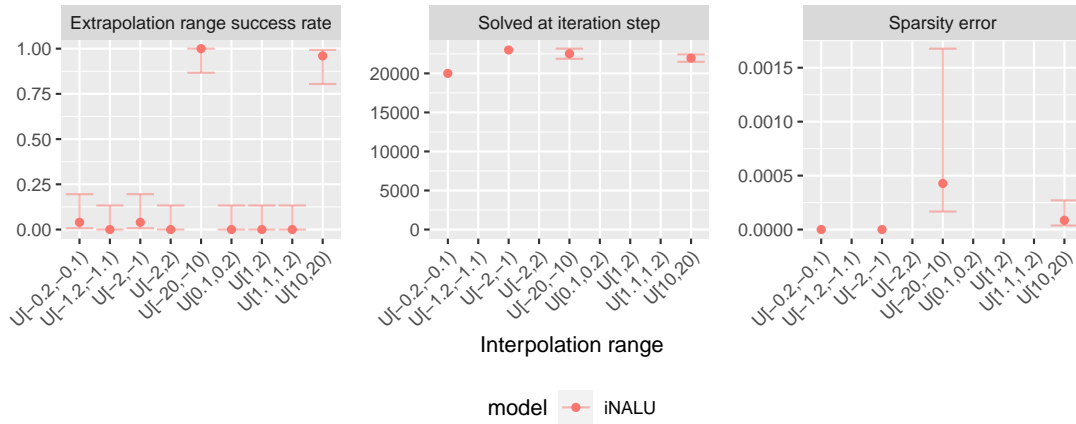


FIGURE 3.8: iNALU failures on the Single Module Task for addition with 10 inputs (8 redundant inputs).

iNALU (Figure 3.8) shows multiple failure ranges at 10 input units (8 redundant inputs), achieving reasonable success only on the larger ranges.

### 3.4 Summary

In this chapter, we have highlighted the need for standardised benchmarks and reliable evaluation metrics for measuring the arithmetic extrapolation performance of NALMs. To this end, we introduce the Single Module Arithmetic Task as a step towards such benchmarks with metrics to measure success rate, convergence speeds and sparsity errors. The Single Module Task assesses the stability of individual NALMs.

Our empirical findings show that NALMs are not robust in learning to extrapolate on the simplest arithmetic setting, as they struggle to consistently learn over different training ranges and seeds. NALMs which achieved full success on all ranges for an operation on the two-input setup break when redundant inputs are introduced, however the extent varies depending on the module. Division is the most challenging operation followed by multiplication, subtraction and addition. NALMs which specialise in at most two operations are found to outperform the NALU in a majority of the cases. Of the NALMs which can model the four operations, i.e., the NALU, iNALU and G-NALU, the iNALU performs best on average over all operations, though the performance gain is less significant for multiplication and division. We will now dive further in and explore potential options to fix this lack of robustness in the upcoming chapters.

## Chapter 4

# Multiplication - Improving Robustness via Stochasticity

So far, by cross-comparing existing NALMs in Chapter 3, we have discovered that current multiplicative modules are unable to be robust over different training ranges. In other words, although modules have adequate representational power, they have difficulty in terms of learnability. We now begin to understand why. We choose to focus on multiplication over the other elementary operations as the function's scalar field is more complex to learn than addition and subtraction, and is well defined for the domain of interest unlike division which is undefined when dividing by zero. We discover that by introducing *reversible stochasticity* to a multiplication NALM, it is possible to improve the robustness without compromising on performance even if the NALM is a component in a DNN.

Using stochasticity to improve learning is a practice that the community has used in many ways. Noise can be injected into the input or weights to improve generalisation by implicitly inducing additional regularisation to the cost (An, 1996). Alternatively, noise can be added to the gradients to encourage exploration giving the model the opportunity to escape the local minima. Such noise is *irreversible*, meaning that the convergence cannot reach zero, but if annealed throughout training then zero convergence can occur (Neelakantan et al., 2015). In contrast to these methods, the input noise method we propose to use is *fully reversible* allowing for better exploration during training, and the ability to get convergence to minimal loss. Stochasticity has also been used to improve model robustness by reducing sensitivity to noisy inputs. For example in adversarial training, data includes inputs perturbed with noise that would trick the model to make incorrect classifications (Goodfellow et al., 2015). In other works, such as denoising autoencoders, noise is added to the input which the network must *learn* to denoise (Vincent et al., 2008). Our approach does not require any learning during the denoising stage and is designed to be automatically reversible no matter the input.

General purpose neural networks which do not use specialist modules can also apply data augmentation to perturb the input to teach the model to be more robust (Shorten and Khoshgoftaar, 2019). However, as our module learns exact multiplication, augmenting the input will result in the output no longer being a function of the multiplication of the original inputs. Therefore, we reverse the effect of input noise in our model at the output to retain the correct input-to-output relation. Code (MIT license) is available at <https://github.com/bmistry4/nalm-robust-nmu>.

## 4.1 Robustness Issues with Multiplication Modules

Previously in Figure 3.4, we highlight robustness issues when training on different inputs when training on a relatively simple task of multiplying two numbers. No module has full success on all ranges, with all modules completely failing on small negative input ranges such as  $\mathcal{U}[-1.2, -1.1)$  and  $\mathcal{U}[-2, -1)$ . Next, we reason as to why such failures occur. To do this, we focus on the best performing NALM, the NMU.

### 4.1.1 Problem: Inputs that Induce Local Optima

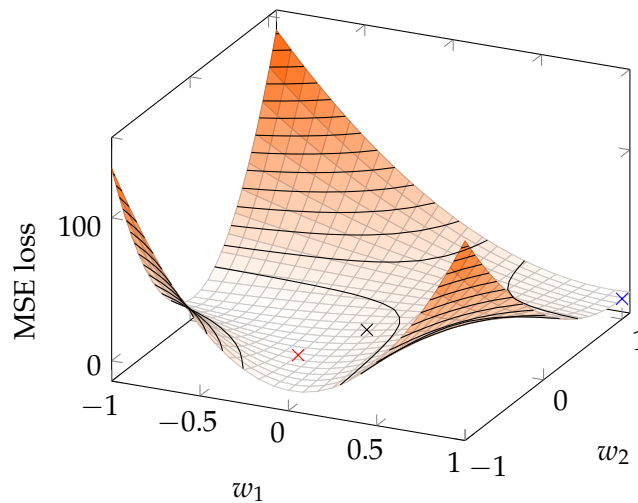


FIGURE 4.1: Static loss landscape with a batch size of 1 for NMU weights in a Single Module Task for learning  $-2 \times -1.8$ . Ideally, the weights should converge to the **global minima (1,1) (blue cross)** which is the extrapolative solution. However, an **alternate minima at  $(-\frac{1}{6}, -0.5)$  (red cross)** exists which solves  $-2 \times -1.8$  but will not extrapolate. Furthermore, since the weights for this minima are  $< 0.5$  the model will stop at  $(0, 0)$  (black cross) due to weight clipping and regularisation.

The NMU fails to get 100% success for interpolation ranges  $\mathcal{U}[-1.2, -1.1)$  and  $\mathcal{U}[-2, -1)$ . Unlike the NALU, the NMU supports negative inputs. Unlike the (Real)NPU and

iNALU, the NMU does not require a mechanism for recovering the sign of the output. Unlike all of the three mentioned NALMs, the NMU does not require supporting division. Therefore, rather than an architectural limitation, there exists some issue with the way weights are learned. Specifically, we find weights converge towards non-extrapolative local optima. Such optima can be considered global for some interpolation cases (achieving a low enough train loss to be considered a solution), but are local optima for extrapolation cases (with high test errors). Figure 4.1 illustrates this issue. For expressing  $-2 \times -1.8 = 3.6$ , rather than learning  $W = [1 \ 1]^T$  which is an extrapolative solution, the weights tend towards the solution  $[-\frac{1}{6} \ -0.5]^T$ . Though this solution does multiply to the correct output value i.e.,

$$\begin{aligned} \text{NMU}([-2 \ -1.8]) &= \prod [-2 \ -1.8] \begin{bmatrix} -\frac{1}{6} \\ -0.5 \end{bmatrix} + 1 - \begin{bmatrix} -\frac{1}{6} \\ -0.5 \end{bmatrix} \\ &= \left( -2 \times -\frac{1}{6} + 1 - \left( -\frac{1}{6} \right) \right) \times \left( -1.8 \times -0.5 + 1 - (-0.5) \right) \\ &= 1.5 \times 2.4 \\ &= 3.6, \end{aligned}$$

it will easily break with other inputs. Furthermore, due to the clipping and regularisation applied to the NMU the final weights become  $[0 \ 0]^T$  rather than  $[-\frac{1}{6} \ -0.5]^T$ . This failure indicates that the NMU's architectural bias of cumulative multiplication, where irrelevant inputs are converted to 1 via the  $+1 - W$ , can cause local minima. As the input range is an independent variable in our experiments, we believe that the input data influences the weight learning towards the local minima which gets wrongly enforced by the model's bias towards discrete weights (from regularisation and weight clipping).

## 4.2 A Stochastic Wrapper: The Stochastic NMU (sNMU)

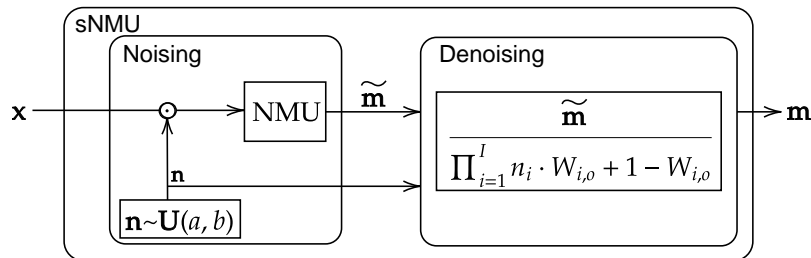


FIGURE 4.2: Stochastic NMU architecture

As the observed issue is correlated with the input values, we ask if there is a way to modify the learning pipeline such that the input has a less negative impact on module learnability. In particular, we focus on solutions that do not modify the existing NMU

architecture as it already provides the relevant mechanics for cumulative multiplication. To this end, we propose the stochastic NMU (sNMU) illustrated in Figure 4.2 to solve the issue.

The sNMU acts as a wrapper on the NMU. There are two stages to the sNMU: (1) *noising* to apply noise to the inputs of the NMU and (2) *denoising* the output of the NMU to cancel the effect of the introduced noise. The noising and denoising are applied with the intention that the resulting output value is the same as using the NMU without noisy inputs. The two stages are detailed below.

**Noising:** Noise  $n_i$  is sampled from  $\mathcal{U}[a, b]$  (where  $a$  and  $b$  are predetermined) and multiplied with each input,  $x_i$ :

$$\text{NMU}_{\text{noisy}} : \tilde{m}_o = \prod_{i=1}^I (n_i x_i W_{i,o} + 1 - W_{i,o}) . \quad (4.1)$$

**Denoising:** Only dividing by the cumulative noise would not fully reverse the effects if there are redundant inputs. To fully cancel the effect of the noise, the output is divided by a denoising factor which induces a bias in the weights forcing them towards being either 0 or 1:

$$\text{sNMU} : m_o = \frac{\tilde{m}_o}{\prod_{i=1}^I (n_i W_{i,o} + 1 - W_{i,o})} . \quad (4.2)$$

The denoising is only used during training; during inference, the module will act exactly like a NMU. Denoising using only the product of the noise values,  $\prod_{i=1}^I n_i$ , is not valid for cases with redundant inputs ( $w_i = 0$ ) where not all inputs are selected for multiplication. To alleviate the redundancy issue, the noise is multiplied with the weight values ( $n_i W_{i,o}$ ). However this causes a division by 0 if weight/noise values are 0 or numeric precision errors if close to 0; therefore, the  $+1 - W_{i,o}$  term is included in the denoising. As the noise distribution is predefined, the lower bound of the noise can be controlled avoiding issues with very small noise values. The denoising occurs at the module output rather than the network output, because when the modules are used as part of a larger end-to-end network the resulting feed-forward expression of the network can become quite complex making it difficult to denoise. Therefore, if denoising is completed at the module-level, the complexity is vastly reduced.

To the best of our knowledge, this work is the first in using reversible noise for NALMs. Since the NMU architecture is not modified, the weight values remain interpretable where 0 still refers to ignoring an input and 1 refers to multiplying the input. Hence, we know with full confidence which inputs will be multiplied. In other words, if weights converge correctly, the sNMU acts as an extrapolative multiplication module which works on any valid input value.



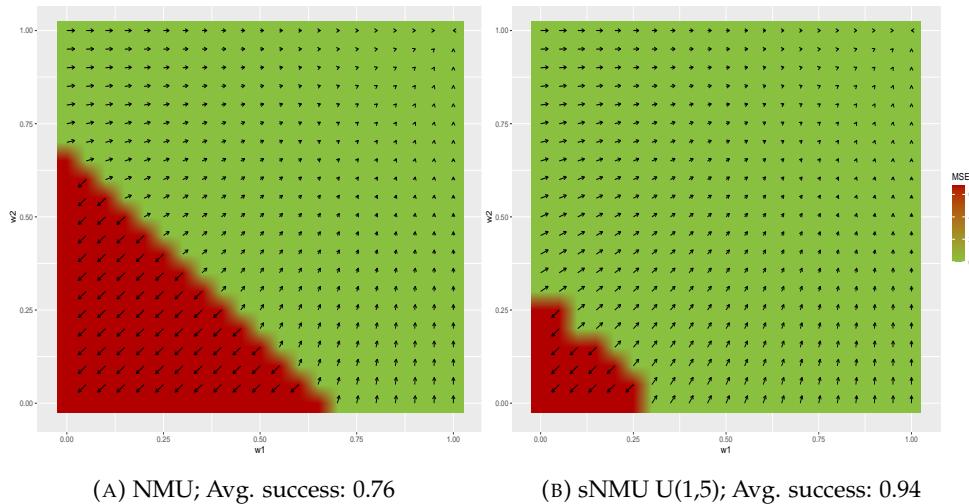


FIGURE 4.3: Heatmap of the MSE for learning  $-2.1 \times -1.8$ . Colours depict the success (green means success and red means failure), assuming that the NALM initial weights are set to  $w_1$  and  $w_2$  respectively. Vectors are shown over the different measure starting points displaying the direction the learning takes. The decimals in the sub-captions represent the average success rate (max=1).

To illustrate how the optimisation improves when using the sNMU, we take the toy example to learn  $-2 \times -1.8$  which displays the local optima issue from Figure 4.1 and empirically measure the success. Multiple independent instances of a NALM are trained, where the starting weights are initialised to some value between  $[0,1]$  which is the allowed range of weights for the NMU/sNMU. Using a grid sweep method, all weight initialisations between 0-1 with a step size of 0.05 are trained. The sNMU samples noise from a  $\mathcal{U}[1,5]$  distribution, resulting in samples which will scale the input magnitude. A model instance is trained without regularisation for 2500 epochs with a SGD optimizer and a learning rate of 0.001. Figure 4.3 illustrates the success (green)/failure (red) for each model instantiation over the grid. A vector field is overlaid on the grid showing the direction the weights will travel from the starting point to the final weights. The successes are shown to converge towards  $[1,1]$ , while failures converge to  $[0,0]$  corresponding to the minima from Figure 4.1. The NMU is clearly more likely to fail over a wider range of initialisations compared to the sNMU, suggesting that the reversible stochasticity aids in escaping the local minima.

### 4.3 Alternate Stochastic Methods

As well as our reversible stochasticity, we consider how other forms of stochasticity can affect the learning of a NALM. The two methods are explicit gradient noise and stochastic gating.

### 4.3.1 Stochastic Gating

Rather than using stochasticity to modify the gradients of the weights, we reformulate the NMU weights such that each weight learns using directly injected noise. To do this, a NMU weight is viewed as a learnable stochastic gate (Yamada et al., 2020). We name this architecture the stochastic gated NMU (stgNMU). A gate represents a continuous relaxation of a Bernoulli distribution by modelling a mean shifted Gaussian random variable clipped between  $[0,1]$ . A learnable mean,  $\mu_i$ , is initialised to 0.5 and the NMU weight is obtained by transforming the gate weight using a hard sigmoid. A NMU weight is defined as  $w_i = \max(0, \min(1, \mu_i + \epsilon_i))$  where  $\epsilon_i$  is noise sampled from  $N(0, \sigma^2)$  and  $\sigma = 0.5$ . During training noise is used, however, during inference no noise is added.  $L_0$  regularisation is applied by taking the probability that the gates are active, which can be calculated using a standard Gaussian CDF i.e.,  $\sum_{i=1}^I \Phi(\frac{\mu_i}{\sigma})$ . To balance the regularisation with the main loss objective, the regularisation is scaled by a pre-defined hyperparameter  $\lambda$ .

### 4.3.2 Gradient Noise

Rather than implicitly altering the gradients by using reversible stochasticity, we see if altering the gradients of a NALM explicitly can improve learning. Following (Nee-lakantan et al., 2015), we add noise sampled from  $N(0, \sigma_t^2)$  to the gradients every training step. The noise is annealed over epochs, therefore  $\sigma_t^2 = \frac{\eta}{(1+e)^{0.55}}$  where  $e$  is the epoch and  $\eta$  is a scaling factor. In experiments, we name this model “NMU + grad noise”.

## 4.4 Single Layer Task

We begin by observing the effect of the stochastic approaches using the single-layer setup from Section 3.3.

The sNMU achieves full success on the Single Module Task on all ranges (Figure 4.4) without compromising the existing advantages of the NMU i.e., low parameter count, fast solving speed and low sparsity error.

Using stochastic gates for the NMU weights (Figure 4.5) can be beneficial if the  $\lambda$  is sufficiently small (0.1 or under). When small  $\lambda$ s are used, the stgNMU can improve on the remaining two ranges where the NMU fails to obtain full success.

Adding gradient noise (Figure 4.6) does not improve the success rate in comparison to the NMU. The convergence speeds get slower with gradient noise and if a larger gradient noise is used (i.e.,  $\eta = 10$ ) then the success can begin to degrade (see  $\mathcal{U}[-0.2, -0.1]$ ). This suggests that naively applying noise is not suited for NALM learning.

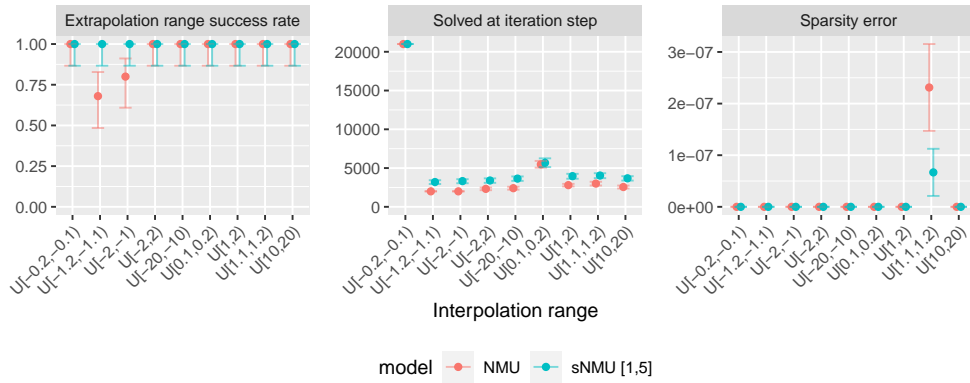


FIGURE 4.4: Single Module Task for multiplication comparing the NMU to a stochastic NMU (sNMU) with noise sampled from  $\mathcal{U}[1,5]$ .

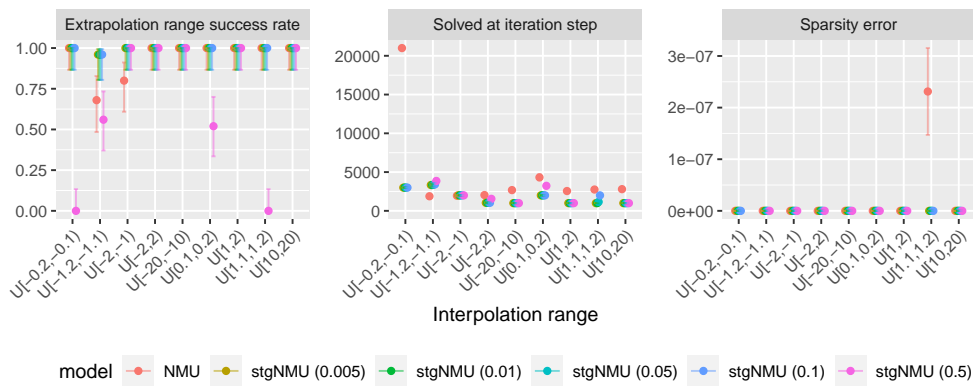


FIGURE 4.5: Single Layer Task results for using stochastic gating to learn the NMU weights over different  $\lambda$  (regularisation scaling).

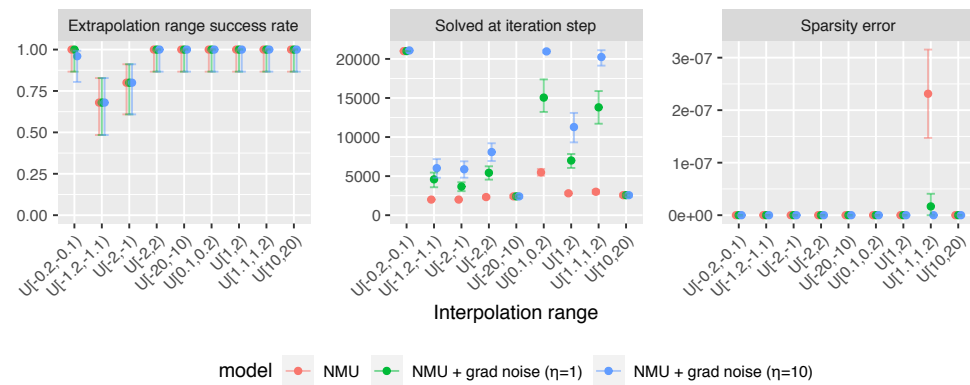


FIGURE 4.6: Single Layer Task results for using additive stochastic noise to learn the NMU weights over different  $\eta$ .

## 4.5 Arithmetic Dataset Task

We now evaluate against the Madsen and Johansen (2019) two-layer task setup described in Section 3.1, presenting results for the stacked ADD-MUL module where the

ADD module is a NAU and the MUL module is a NMU/stochastic NMU variant. A summary of the experiment parameters is shown in Table 4.1. Additional module specific hyper-parameters, hardware and runtimes are found in Appendix E.1.

TABLE 4.1: Parameters for the Arithmetic Dataset Task which are applied to all modules. \*Validation and test datasets generate one batch of samples at the start which gets used for evaluation for all iterations.

Parameter	Arithmetic Dataset Task
<b>Layers</b>	2
<b>Input size</b>	100
<b>Subset ratio</b>	0.25
<b>Overlap ratio</b>	0.5
<b>Total iterations</b>	5 million
<b>Train samples</b>	128 per batch
<b>Validation samples*</b>	10000
<b>Test samples*</b>	10000
<b>Seeds</b>	20
<b>Optimiser</b>	Adam (betas=(0.9, 0.999))

Taking the best hyperparameters from the single-layer task and training on the two-layer Arithmetic Dataset Task results in Figure 4.7 which shows that compared to the original NMU both the stgNMU and gradient noise hurt performance while the sNMU can improve performance. By manipulating the input with the reversible noise during training, the resulting gradients can in some cases be better equipped to escape local optima and providing exploration; we further investigate this in Section 6.2. The stacked NAU-NMU fails on multiple ranges. The sNMU shows improvement with faster solve speeds and lower sparsity errors compared to the NMU. The sNMU fixes all failures in  $\mathcal{U}[-2, 2)$  and improves the success rate of  $\mathcal{U}[-0.2, -0.1)$  from 0.75 to 0.9 and

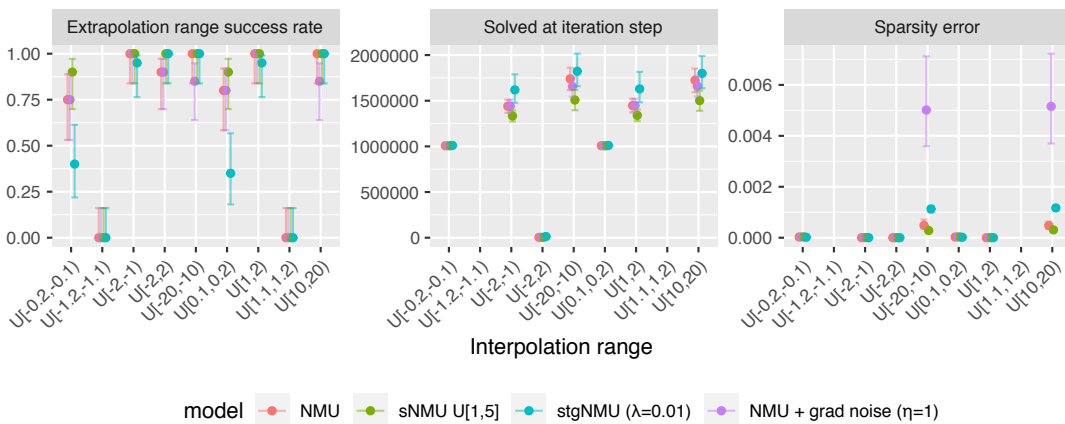


FIGURE 4.7: Arithmetic Dataset Task comparing the NMU to NMUs trained with the following types of stochastic methods: reversible stochasticity (sNMU), stochastic gate weights (stgNMU) and gradient noise (NMU + grad noise).

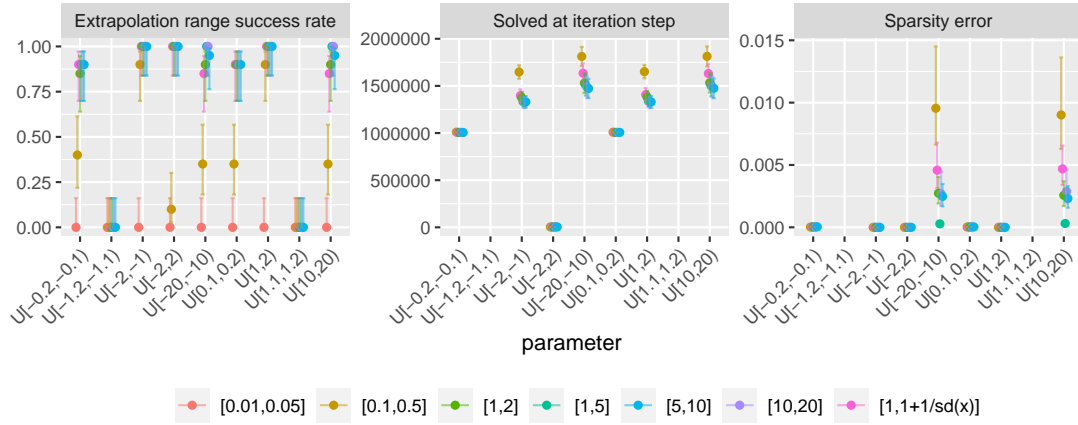


FIGURE 4.8: Arithmetic Dataset Task for multiplication over various Uniform noise ranges for the NAU-sNMU.

$\mathcal{U}[0.1,0.2)$  from 0.8 to 0.9. However, ranges  $\mathcal{U}[1.1,1.2)$  and  $\mathcal{U}[-1.2,-1.1)$  remain at a success rate of 0 for both models.

The reversible stochasticity’s only tunable parameter is the noise range. Figure 4.8 shows the effect of using different noise ranges for the sNMU. Smaller ranges less than one perform worse for data generated with a Uniform distribution and too large a range also shows degradation in the performance in success and sparsity on larger training ranges. Using *batch noise*, which automates the noise range by using batch statistics to sample from  $\mathcal{U}[1, 1 + \frac{1}{\sigma(x)}]$ , where  $\sigma(x)$  is the standard deviation of the sNMU input  $x$ , also achieves reasonable results. The lower bound is set to 1 so the noise cannot scale down the gradient magnitude. The upper bound function (see Figure 4.9) models an exponential decay curve where data batches with larger standard deviations result in smaller noise values while smaller standard deviations have much larger values. The intuition behind this is when the data distribution has similar magnitude samples (i.e., small standard deviations) it is easier for the module to confuse the different input features therefore having more noise makes it easier to differentiate between them. That being said, the range  $\mathcal{U}[1,5]$  is found to be the best for this task in regard to the three evaluation metrics.

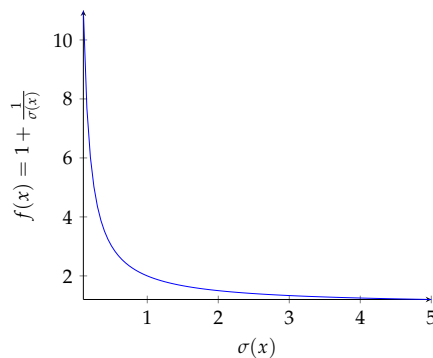


FIGURE 4.9: Upper bounds for batch (statistic) noise.

## 4.6 MNIST Arithmetic

The findings from the previous synthetic number tasks show that from the different noise methods, only the reversible stochasticity is effective and can provide improvements over the NMU. We now take this a step further and focus our attention on image-based tasks where multiplication is required. Monolithic networks have been found to be unable to learn multiplication on image-based inputs (e.g., multiplying the numbers shown in two images) (Hoshen and Peleg, 2016). This section, therefore, explores the effect of including specialist multiplication modules as a downstream layer for two types of image tasks: static MNIST product and sequential MNIST product. The static product task investigates learning to multiply images composed of two MNIST digits, and two variations of this task are explored: isolated digit classification and colour channel concatenated digit classification. The sequential product task investigates multiplying a sequence of MNIST images. Summaries of hardware/runtimes can be found in Appendix E.2.

### 4.6.1 Static MNIST Product


We investigate learning to multiply images composed of two MNIST digits. Two variations of the task are explored: isolated digit classification and colour channel concatenated digit classification. Table 4.2 summarises the experiment parameters.

TABLE 4.2: Static MNIST Product experiment parameters.

Parameter	Isolated Digits	Colour Channel Concatenated
Epochs	1000	1000
Samples per permutation	1000	1100
Train:Val:Test	90:-:10	51:15:34
Batch Size	128	256
Train samples	90,000	61,710
Test samples	10,000	37,400
Folds/Seeds	10	3
Optimiser	Adam (betas=(0.9, 0.999))	Adam (betas=(0.9, 0.999))
Criterion	MSE	MSE
Learning rate	$10^{-3}$	$10^{-3}$
$\lambda_{start} - \lambda_{end}$ epochs	30-40	30-40
$\hat{\lambda}$	100	100

#### 4.6.1.1 Isolated Digit Classification

**Motivation.** First, we determine if digit classification can be learned in upstream layers in a simple setting where no image localisation is required.

**Setup and network.** Following Bloice et al. (2021)'s setup, the dataset contains permutation pairs of MNIST digits side-by-side with the target label being the product of the digits, e.g. input  with output 4 ( $= 4 \times 1$ ). Importantly, although there is no overlap between the permutation pairs in the train and test set, all individual digits (between 0-9) are seen during training. For example, the pair '54' would exist in the test set and not the train set but the digits '5' and '4' would exist in other pairs of the train set such as '15' or '47'.

The network learns a map from the input image to the labels of the two digits (digit classifier), followed by a map from the two labels to their product (multiplication layer). As the commutative property of multiplication can cause learning difficulties for the digit classifier, we separate the two digits into single digits, classify per digit (using softargmax which is defined in Chapter 7's Equation 7.3) and then recombine the two labels. The digit classifier is a convolutional network.<sup>1</sup>

The multiplication layer is done in three different ways: (1) solved multiplication baseline model (MUL), (2) fully connected (FC) layer whose output is the product of the learnable weights, and (3) NALM: NMU/sNMU. For a fair comparison, the fully connected network uses the same initialisation scheme as the NMU. The FC layer uses weighted product accumulators rather than linear layers as the latter does not have the capacity to do multiplication. Additional details regarding the network architecture are given in Appendix E.4.1.

**Metrics and results.** The network is trained using a MSE loss criterion between the predicted multiplication and the target value, and a discretisation regularisation (if a NALM is used as the multiplication layer). The MUL baseline only needs to learn to classify the images to their respective labels and therefore is considered a strong baseline. For a NALM to outperform the baseline would imply that the arithmetic inductive bias can aid learning of downstream layers. The results in Figure 4.10 use a strict criterion for measuring accuracy as the predictions are not processed in any way (e.g. rounded/truncated). Hence a model must learn to apply the operation and classify the digits exactly. Results show both the sNMUs with  $\mathcal{U}[1,5]$  noise and batch noise (96.6% and 85.6%) outperform the NMU (59.3%) for the test output metric, with no overlap in confidence bounds.

---

<sup>1</sup>From the PyTorch MNIST example <https://github.com/pytorch/examples/blob/master/mnist/main.py>.

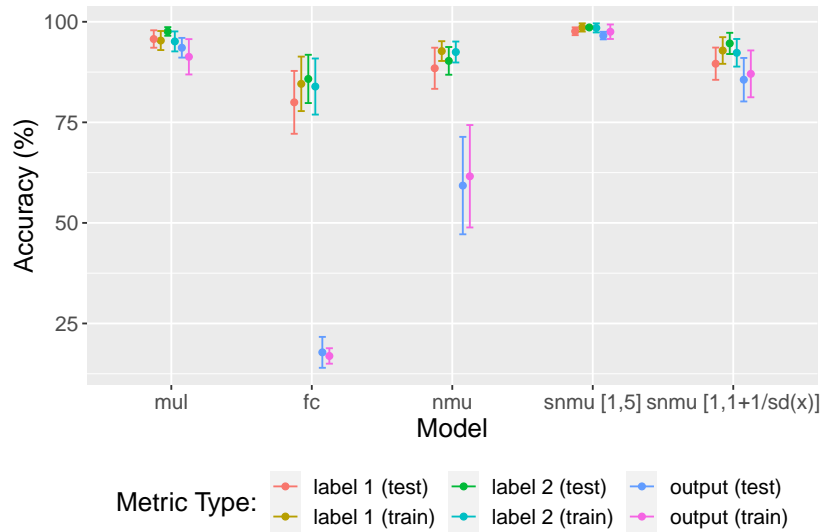
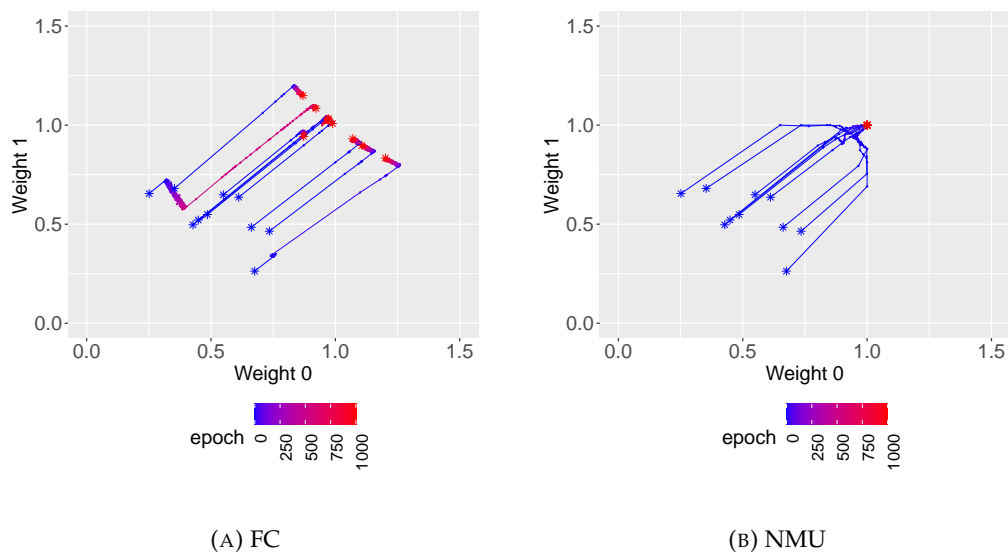


FIGURE 4.10: Isolated Digits task accuracies for classifying each of the two digits (label 1 and label 2) and the final product (output) for training and testing.

The FC model learns to classify each label to a reasonable accuracy but does not learn the multiplication weights robustly resulting in poor output accuracy. In contrast, the specialised modules, the NMU and sNMU all have a similar label and output accuracies. This can be observed by plotting the trajectories for the two learnable weights used when calculating the multiplication operation for each fold. We expect a correct solution to converge to the weights of  $[1, 1]$ . The baseline, which uses a solved multiplier, has no learnable weights for multiplication and hence has no trajectory plot.

Figure 4.11 shows the FC models are unable to reliably converge to the true solution on any run. The NMU gets close to the solution but only 70% of runs converge to weights of 1 exactly, while 100% of the sNMU models converge. The  $\mathcal{U}[1,5]$  sNMU outperforms both the NMU and MUL model, suggesting that the reversible stochasticity not only





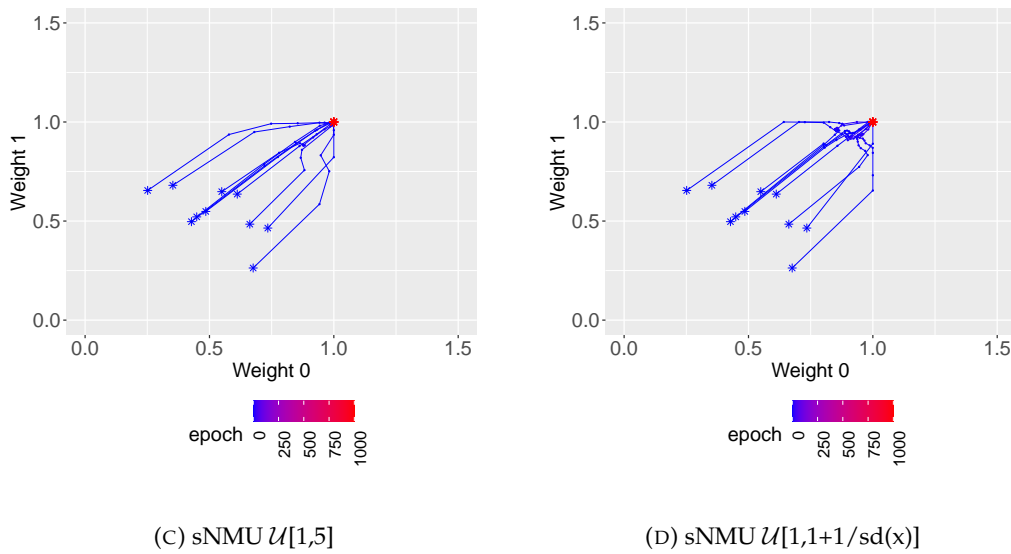


FIGURE 4.11: Path learned by the weights for the multiplication network for the Isolated Digits task. Each path represents a different seed. Blue and red asterisks represent the starting and ending points respectively. The target to reach is (1,1).

improves robustness but can aid with learning upstream layers. For further analysis regarding the classification accuracies of the digit classifier network see Appendix F.

#### 4.6.1.2 Colour Channel Concatenated Digit Classification

**Motivation.** Confirming NALMs can be effective using simple digit classifiers, we now ask if this remains the case if the difficulty for the classification network is increased.

**Setup and network.** Following Jaderberg et al. (2015), random rotation, scaling and translation transforms are applied to the digits and the image classifier must learn to localise digits as images now contain both digits separated by the colour channel.

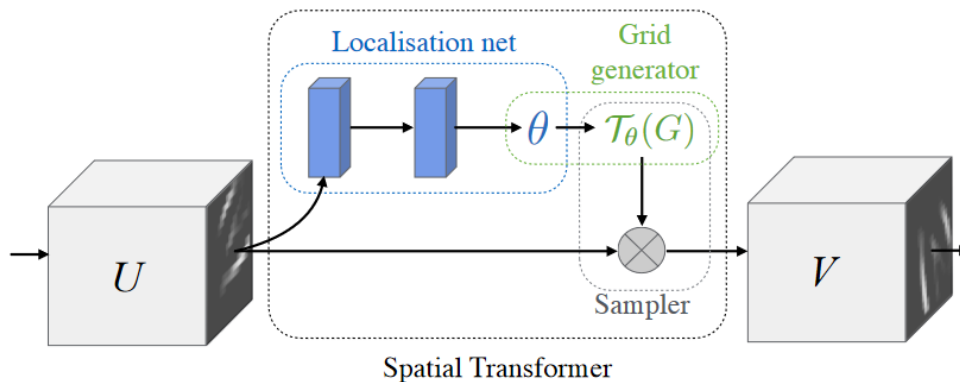


FIGURE 4.12: Spatial Transformer Network (STN), taken from Jaderberg et al. (2015, Figure 2.6).

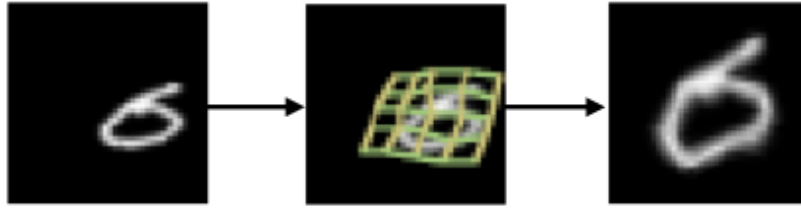


FIGURE 4.13: Example of applying an STN with a Thin Plate Spline (TPS) transformation on a distorted MNIST digit. This image is a partial reproduction of Jaderberg et al. (2015, Table 1).

The digit classifier uses the Spatial Transformer Network (STN) (Jaderberg et al., 2015) shown in Figure 4.12, which is a differentiable layer that allows attention to a relevant area of an image and transforms the area to a less distorted (canonical) pose (see Figure 4.13).

There are three parts to the STN - the localisation network, the grid generator and the sampler. First, the *localisation network* is a regression network that outputs the parameters for the spatial transformation. We use a 16 control point Thin Plate Spline (TPS) transformation for the digit localisation (Bookstein, 1989), but other possibilities would include using an affine transformation or attention transformation. Since there are two MNIST digits to classify, two different localisation networks are learnt; one for each digit. Second, the *grid generator* creates a sampling grid that indicates where the input should be sampled. In other words, by applying the spatial transformation parameters to the target points, we can determine the set of coordinates (an interpolation sampling grid) on the input (as floating-point values) which need to be sampled to generate the output. Third, the *sampler* creates the output by applying the interpolation grid to the input to sample the coordinates from the input via bilinear interpolation. The resulting spatially transformed digits are processed by a convolutional network to obtain the logits for digit classification (using softargmax). The multiplication layer uses the same options as the Isolated Digits task from Section 4.6.1.1 (i.e., MUL, FC, NMU and sNMU). Additional details regarding the network architecture are given in Appendix E.4.2.

**Metrics and results.** The network is trained using a MSE loss criterion between the predicted multiplication and the target value, and discretisation regularisation (if a NALM is used as the multiplication layer). For plotting results, the accuracies of each digit label and the final output value are taken. Due to the increased difficulty of classification, the label and output predictions are rounded before calculating the accuracies. Figure 4.14 shows that the sNMU with batch noise is able to get comparable test output accuracy to the solved baseline (67.9% vs 68.9%) with tighter confidence bounds, suggesting improved robustness for the digit classifier network. There is also an improvement (+13%) in classifying the first digit. Using a noise range of  $\mathcal{U}[1,5]$  has a weaker performance in comparison to using batch noise.

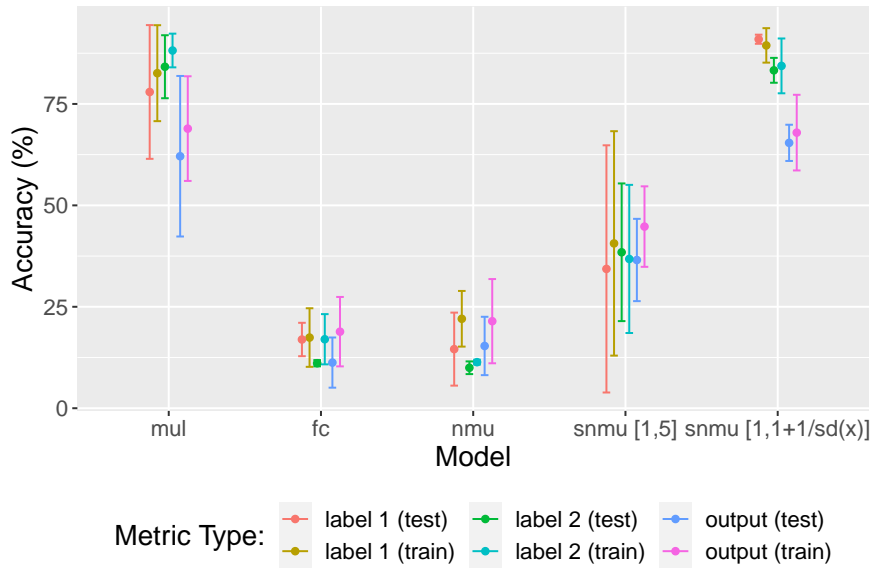
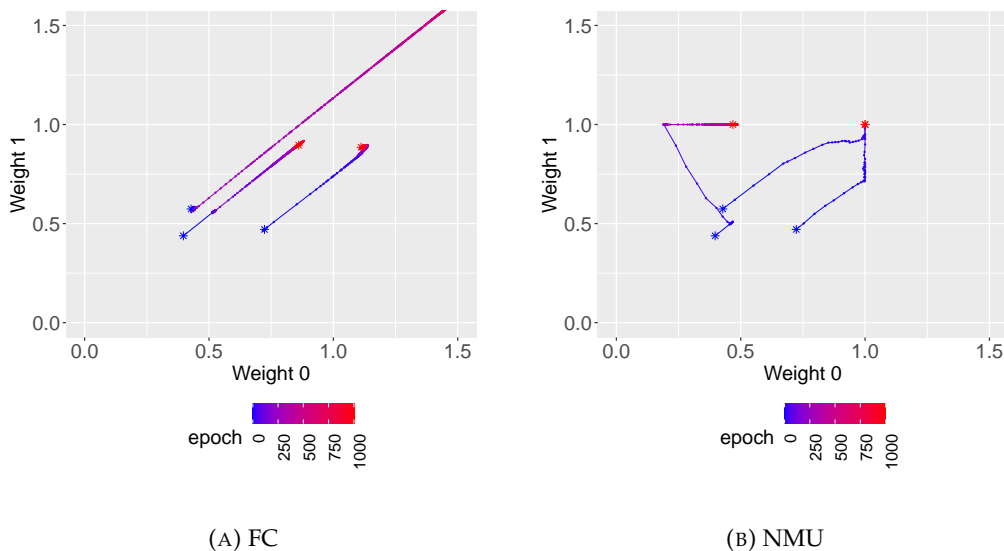


FIGURE 4.14: Accuracies on the Colour Channel Concatenated MNIST task for classifying each of the two digits (label 1 and label 2) and the final product (output) for training and testing. Accuracies are calculated after rounding the labels/output predictions.

The weight trajectories from Figure 4.15 show that the NMU fails to converge towards the correct multiplication weights for a fold, unlike its stochastic versions, achieving similar accuracies to the FC based network. Like the Isolated digits task, the FC models are again unable to converge reliably to the true solution. For further analysis regarding the classification accuracies of the digit classifier network see Appendix F.



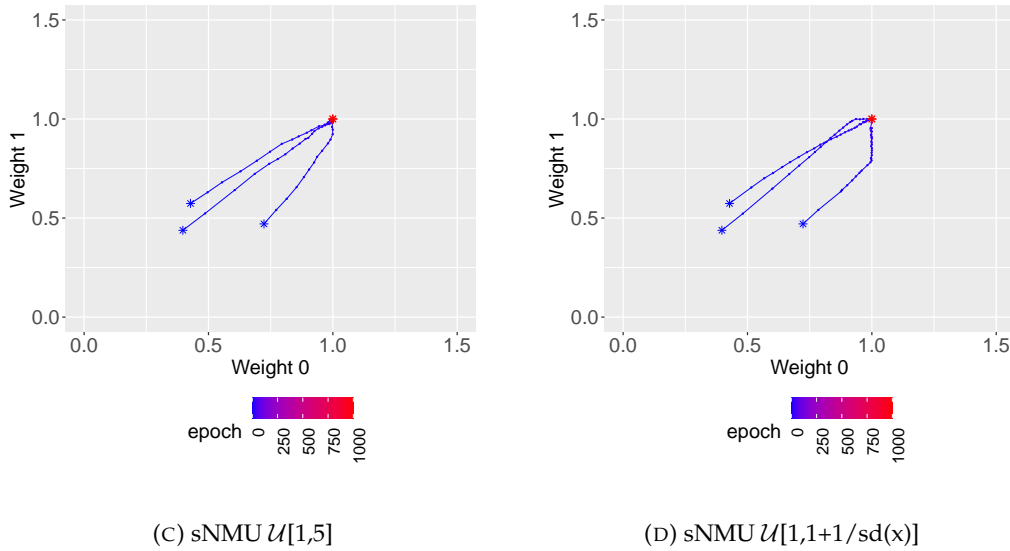


FIGURE 4.15: Path learnt by the weights for the multiplication network in the Colour Channel Concatenated task. Each path represents a different seed. Blue and red asterisks represent the starting and ending points respectively. The target to reach is (1,1).

## 4.6.2 Sequential MNIST Product

A summary of the experiment parameters is given in Table 4.3. Additional details regarding the network architecture are given in Appendix E.4.3.

TABLE 4.3: Sequential MNIST experiment parameters. The Validation digits are the last 5000 images of the MNIST training set. Iterations are equivalent to the number of image batches seen; it is a global count over the epochs.

Parameter	Sequential MNIST
<b>Epochs</b>	1000
<b>Batch Size</b>	64
<b>Train digits</b>	55000
<b>Validation digits</b>	5000
<b>Test digits</b>	10000
<b>Interpolation length</b>	2
<b>Extrapolation lengths</b>	Up to 20
<b>Optimiser</b>	Adam (betas=(0.9, 0.999))
<b>Criterion</b>	MSE
<b>Learning rate</b>	$10^{-3}$
$\lambda_{start} - \lambda_{end}$ <b>iterations</b>	10000-100000
<b>Seeds</b>	10

**Motivation.** To test the effect of the modules in a different type of extrapolative setting, a sequential task is adopted where the number of digits to multiply can be controlled.

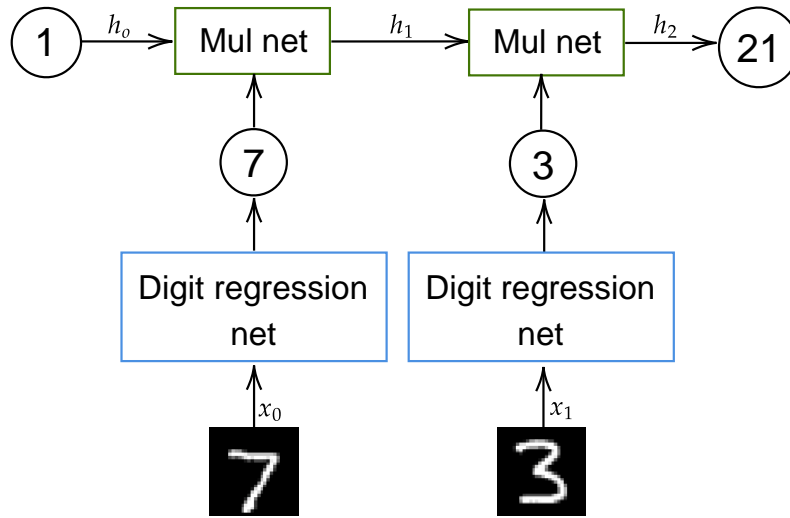


FIGURE 4.16: Example of a Sequential MNIST network unrolled for two timesteps. The Digit regression net is the image-to-value network and the Mul net is the multiplication network for multiplying the hidden state (i.e., the memory which stores the accumulated multiplication value) with the input element. The starting hidden state is 1 which is the identity value of multiplication.

**Setup and network.** Following Madsen and Johansen (2020), given a sequence of MNIST digits (between 1-9), process one image at a time using a classification network to regress an image to its label value, which gets passed into a recursive NALM cell to calculate the cumulative result as shown in Figure 4.16. The NALM will take in two inputs: the predicted label of the image at the current timestep and the predicted accumulated value from the previous timesteps. A convolutional network is used to regress the images to a scalar real number representing the digit. The multiplication layer is either solved (baseline) or requires learning a NALM (NMU or sNMU).

Regularisation on the output of the digit regression network is applied to encourage solutions which adhere to the bias that NMU weights of 1 result in multiplication, i.e.,

$$R_z = \frac{1}{I} \sum_i^I ((1 - W_i) \cdot (1 - \bar{z}_i)^2) .$$

where  $I$  is the number of input features,  $O (=1)$  is the number of output features,  $W_{i,o}$  is a NMU weight and  $\bar{z}_i$  is the average input values into the recurrent NALM cell (over the batch and timesteps). If the NMU weight is 1 then there is no penalty. If the weight is converging towards 0 then the penalty uses the  $(1 - \bar{z}_i)^2$  to penalise the inputs to the cell from becoming too large which encourages sub-optimal solutions. On top of this, for the NALM multiplication units, regularisation which encourages NMU discretisation (see Equations 2.15 and 2.16) is also applied.

**Metrics and results.** The training metric used is the MSE taken between the output for each timestep and its corresponding cumulative target. Similar to the evaluation of arithmetic tasks from Chapter 3, a success threshold is calculated to determine if the MSE of the network should be considered a success. The success threshold is the 1% one-sided upper confidence interval using a student-t distribution over the MSE of a network where the cumulative multiplication layer is solved. Training uses two-digit sequences while testing uses sequences up to 20 digits long.<sup>2</sup> A validation set, made up of two-digit sequences, is used for early stopping by taking the best-performing model on the validation set and using it for testing. Figure 4.17 shows the results with both sNMU networks outperforming the NMU over multiple extrapolation lengths while retaining fast convergence similar to the NMU. The batch sNMU underperforms in comparison to the NMU between sequence lengths 11-14, while the noise range  $\mathcal{U}[1,5]$  only underperforms on length 15.

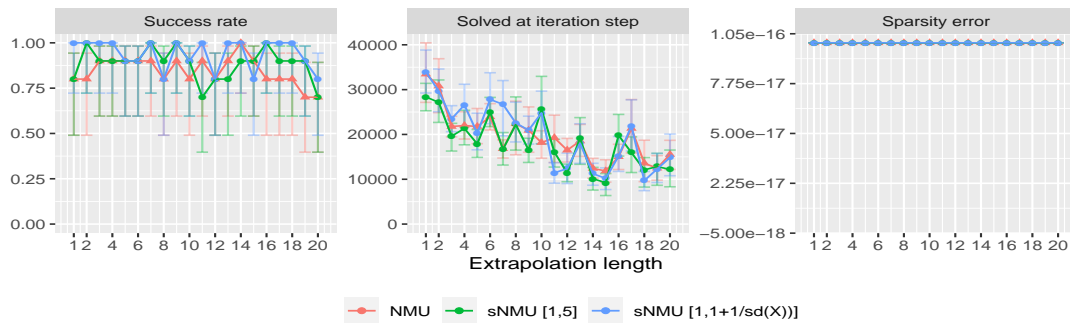


FIGURE 4.17: Performance on Product of Sequential MNIST. Model names represent the type of multiplication cell used. All models use the same CNN architecture to do digit classification. Error bars represent the 95% confidence interval.

## 4.7 Summary

In this chapter, we have identified a cause of the lack of robustness in the NMU multiplication module. Specifically, the issue of local optima causes the NALM to attempt to converge to weights outside of its allowed weight range. To begin to alleviate this issue, we design a fully reversible stochastic wrapper on the NMU which improves the NALM's chances to escape local optima. The wrapper nature of this idea means that the internal NALM dynamics do not need to be altered. We validate the idea empirically over arithmetic and MNIST experiments, finding in many cases the stochasticity improves over the non-stochastic NMU.

There are three next steps that naturally follow this work. The first would be to understand the effect different noise distributions have on learning. The second is to confirm whether stochasticity can also aid other NALMs. As the stochasticity is fully reversible,

<sup>2</sup>Longer sequences can result in floating-point precision errors.

---

the definition of denoising would need to be altered for different NALMs since denoising relies on the weight values of the module. The third is to explore how well stochasticity works when used on more complex visual datasets such as SVHN (Netzer et al., 2011) or handwritten digits e.g., HINT (Li et al., 2021).





## Chapter 5

# Division - Understanding the Underlying Learning Mechanisms

Division is one of the four fundamental arithmetic operations and is necessary for expressing real-world dynamical systems (Sahoo et al., 2018; Li et al., 2019) and physics-based formulas (Udrescu and Tegmark, 2020). However, the properties of division of values around zero lead to undesirable gradients for training neural networks through backpropagation, making it the hardest operation to learn. From reviewing the existing NALM literature in Chapter 2, we identified a constant agreement from all works that division is difficult to learn. In Chapter 3 we empirically show this difficulty does indeed exist, especially compared to the other arithmetic operations. We now break-down the issue of learning division, investigating factors that cause learning issues and mechanisms to alleviate the issues.

To set the scene for this chapter, imagine you must learn to divide 2 numbers from a list of 10 numbers, but are only given the 10 numbers and the expected result. This task requires finding the 2 relevant operands, the order to divide the operands, and learning to divide. In ML, this is equivalent to a supervised regression task where the aim is to learn the underlying function between the inputs and output such that the solution is generalisable to any input. For neural networks, the main challenge of this task comes from having to learn the selection and operations simultaneously, which can lead to conflicting priorities when learning weights. The ability to do a hard selection on relevant inputs/features is a desirable property for neural networks, useful for improved interpretability, reduced preprocessing costs and greater generalisation (Chandrashekar and Sahin, 2014). As differentiable specialist modules, such as those for arithmetic operations, can be integrated with overparametrised networks as an intermediate module, being able to successfully select only the relevant inputs is important (Madsen and Johansen, 2020).

Can we build models which can learn division in the presence of its undesirable, yet valid, properties? We aim to address this question in this chapter by optimising an existing NALM, the Real NPU, and introducing two alternate division modules which attempt to extend the NMU to do division. A desideratum for building a division module is provided in Appendix G.1 and code (MIT license) is available at <https://github.com/bmistry4/nalm-division>.

## 5.1 Related Work

Learning to robustly divide provides a stepping stone for neural networks to achieve symbolic regression. Compared to black-box neural network functions, expressing a mathematical function is significantly easier to interpret. A fully differentiable neural network approach can incorporate biases to improve the interpretability of the network. [Sahoo et al. \(2018\)](#) sets activation functions in a layer to different symbolic operations rather than using a traditional non-linear activation like ReLU. They also encourage only using relevant weights through a sparsity regularisation scheme which varies in strength depending on how much training has occurred. However, gaining the best performance requires using selection strategies over many trained modules which is costly and can be unreliable. In contrast, [Udrescu and Tegmark \(2020\)](#) exploits patterns in the data by designing physics-related biases such as translational symmetry or multiplicative separability into their architecture. Due to the strong prior which assumes the dataset contains underlying physics representations, the model performs poorly when trained on datasets without such representations ([Cava et al., 2021](#)).

As we have seen, NALMs are another type of differentiable neural networks. Multiple studies have shown using NALMs such as the NALU which ignores dealing with the singularity in division is unstable in learning ([Madsen and Johansen, 2020](#); [Schlör et al., 2020](#); [Heim et al., 2020](#)). Even with all the iNALU's modifications on the NALU including weight and gradient clipping, sign retrieval, regularisation, reinitialisation and separating shared parameters, consistently learning division to high precision remains unattainable ([Schlör et al., 2020](#)). The original empirical findings of the Real NPU claim to outperform the iNALU for division ([Heim et al., 2020](#)) which we confirm in our benchmark results of Figure 3.5, finding that the Real NPU has less complete failures over more ranges.

## 5.2 Architectures

This section introduces the architectures for the *Neural Reciprocal Unit (NRU)*, and the *Neural Multiplicative Reciprocal Unit (NMRU)*. Both NALMs are novel contributions

which extend the existing NMU (see Section 2.2.3) to do division. Since these architectures are NALMs, they can be viewed as regression modules trained via supervised learning. We assume the inputs are represented as a vector of features, where only certain input features are relevant to the output. As with the earlier illustrations from Chapter 2, we use Figure 5.1 as the key for the NRU and NMRU architecture illustrations.

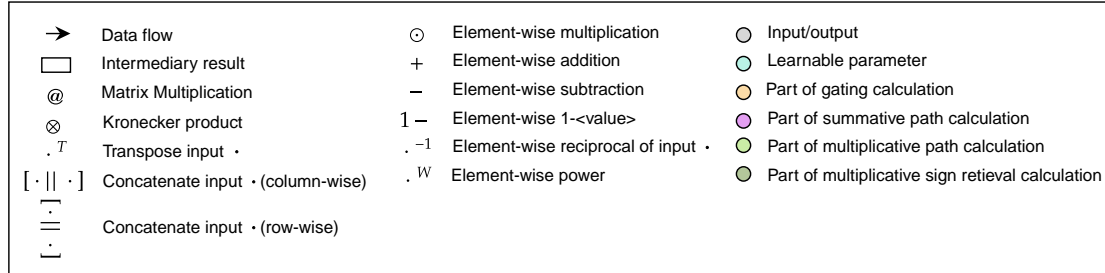


FIGURE 5.1: Key of the symbols and colouring system for architecture illustrations.

### 5.2.1 NRU

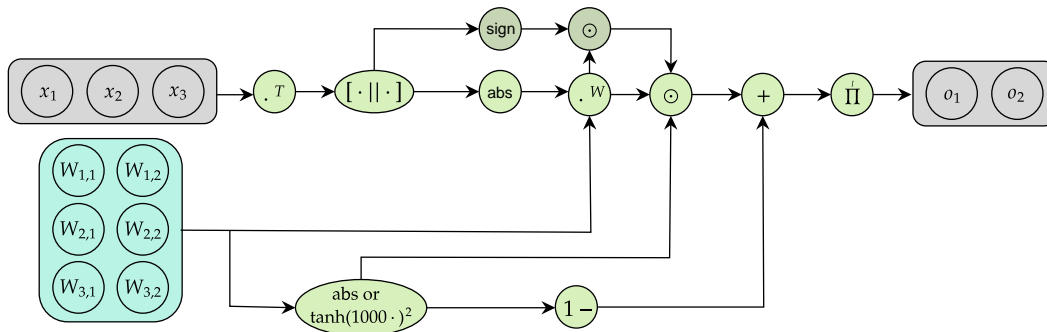


FIGURE 5.2: NRU architecture. Example of a 3-feature input and 2-feature output model.

We propose the NRU, which can model multiplication and division, by extending the NMU, motivated by the fact that *division is the multiplication of reciprocals*. Compared to the NMU, the allowed range for weight values is extended from  $[0,1]$  to  $[-1,1]$ , where -1 represents applying the reciprocal on the corresponding input element. A NRU output element  $z_o$  is defined as

$$\text{NRU} : z_o = \prod_{i=1}^I \left( \text{sign}(x_i) \cdot |x_i|^{W_{i,o}} \cdot |W_{i,o}| + 1 - |W_{i,o}| \right), \quad (5.1)$$

where  $I$  is the number of inputs. Assuming weights are either 1 (multiply) or -1 (reciprocal),  $|x_i|^{W_{i,o}}$  will apply the operation on an input element. The absolute value is

used so that the module only operates in the space of real numbers, as  $x_i^{W_{i,o}}$  for a negative input ( $x_i$ ) when  $-1 < W_{i,o} < 1$  results in a complex number. The use of absolute means the sign of the input must be reapplied. For the no-selection case  $W_{i,o} = 0$ , we want the input element to convert to 1 (the identity value), resulting in applying  $\cdot |W_{i,o}| + 1 - |W_{i,o}|$ . The derivative of the absolute function at 0 is undefined meaning the gradients of Equation 5.1 can contain points of discontinuity. To alleviate this issue, we approximate the absolute function using a scaled tanh (inspired by Faber and Wattenhofer (2020)). More formally,

$$|W_{i,o}| = \begin{cases} \tanh(1000 \cdot W_{i,o})^2 & \text{if training} \\ |W_{i,o}| & \text{otherwise} \end{cases}.$$

The scale factor (1000), selected from tanh scaling experiments (see Appendix G.2), controls how close to the absolute function the approximation is for -1, 0 and 1, where larger scaling factors result in sharper approximation functions. For clipping and regularisation, the same scheme as the Neural Addition Unit (NAU) (see Equations 2.15 and 2.16) is used, forcing weight elements to converge to -1, 0 or 1.

### 5.2.2 NMRU

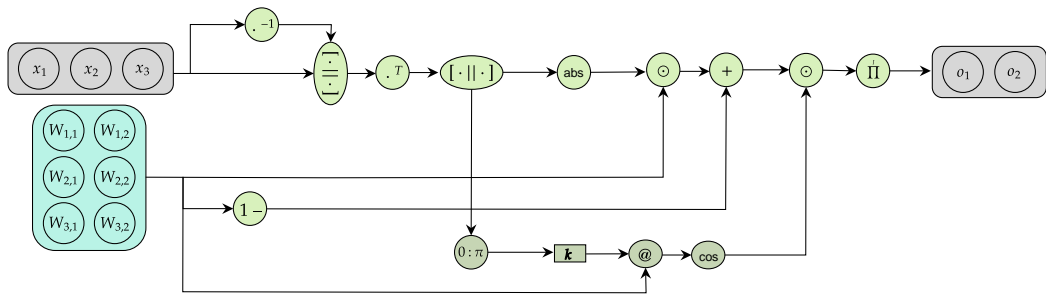


FIGURE 5.3: NMRU architecture. Example of a 3-feature input and 2-feature output model.

An alternate extension of the NMU, also motivated by *division being a multiplication of reciprocals* is the NMRU (Equation 5.2). We concatenate the reciprocal of the input (plus a small  $\epsilon$ ) to the input resulting in a module that only needs to learn selection. Hence, weights can be in the range  $[0,1]$ .

$$\text{NMRU} : z_o = \prod_{i=1}^{2I} (W_{i,o} \cdot |x_i| + 1 - W_{i,o}) \cdot \cos\left(\sum_{i=1}^{2I} (W_{i,o} \cdot k_i)\right), \text{ where } k_i = \begin{cases} 0 & x_i \geq 0 \\ \pi & x_i < 0 \end{cases}. \quad (5.2)$$

The iteration over  $2I$  represents going through all inputs and their reciprocals. We calculate the magnitude and sign separately, joining the result at the end. The magnitude

is calculated by passing the absolute of the concatenated input through an NMU architecture and the sign is calculated by using a cosine mechanism similar to the Real NPU (see Section 2.3.1). However, unlike the Real NPU only the weight matrix is required. The norm of the weight’s gradients is clipped to 1 prior to being updated by the optimiser. This is done to alleviate the issue of exploding gradients caused by including the reciprocal to the inputs. For clipping and regularisation, the same scheme as the NMU (see Equations 2.15 and 2.16) is used, forcing weights to converge to 0 or 1.

### 5.3 Single Module Arithmetic Experiment Setup

TABLE 5.1: Parameters which are applied to all modules. Parameters have been split based on the experiment. \*Validation and test datasets generate one batch of samples at the start which gets used for evaluation for all iterations. † the Real NPU modules use a value of 1.

Parameter	Without redundancy	With redundancy
<b>Layers</b>	1	1
<b>Input size</b>	2	10
<b>Total iterations</b>	50,000	100,000
<b>Train samples</b>	128 per batch	128 per batch
<b>Validation samples*</b>	10000	10000
<b>Test samples*</b>	10000	10000
<b>Seeds</b>	25	25
<b>Optimiser</b>	Adam (betas=(0.9, 0.999))	Adam (betas=(0.9, 0.999))
$\hat{\lambda}^\dagger$	10	10

To evaluate the NALMs we will use our Single Module Arithmetic benchmark (see Section 3.3), where the task evaluates the ability of a single module to divide two numbers from an input vector in two settings: *without redundancy* (2 inputs) and *with redundancy* (10 inputs with 8 redundant inputs).

Table 5.1 summarises the experiment parameters and additional experiment details are found in Appendix E.3. All experiments use a MSE loss with an Adam optimiser (Kingma and Ba, 2015) and 10,000 samples for the validation and test sets. The training uses batch sizes of 128 and the best model for evaluation is taken using early stopping on the validation set. All inputs are required in the *without redundancy* setting, i.e., input size of 2. Training takes 50,000 iterations where each iteration consists of a different batch. The Real NPU uses a learning rate of  $5 \times 10^{-3}$  with sparsity regularisation scaling during iterations 40,000 to 50,000. The NRU and NMRU use sparsity regularisation scaling during iterations 20,000 to 35,000 and a learning rate of 1 and  $10^{-2}$  respectively. In contrast, the *redundancy* setting uses an input size of 10, where 8 input values are not required for the final output. The total training iterations are extended to 100,000. The

learning rates for the Real NPU, NRU and NMRU are  $5 \times 10^{-3}$ ,  $10^{-3}$  and  $10^{-2}$  respectively. Sparsity regularisation scaling occurs during iterations 50,000 to 75,000 for all modules.

## 5.4 Improving the Real NPU’s Robustness

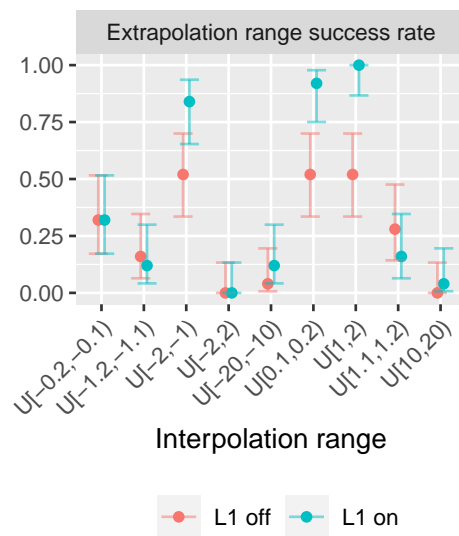
We first improve the robustness of the Real NPU against different training ranges. The Single Module Task with no redundancy (see Section 5.3) is used to investigate the following: (1) Is  $L_1$  regularisation required, and if so, do the regularisation parameters require tuning? (2) Does clipping the learnable parameters aid learning? (3) Does enforcing discretisation on parameters improve convergence? (4) Can the weight matrix initialisation be improved?

To address each question, we apply incremental modifications to the Real NPU. Modifications include an ablation study on the  $L_1$  regularisation (including a sweep over the scaling range hyperparameters), clipping, enforcing discretisation, and a more restrictive initialisation scheme. We assume that we are optimising the Real NPU to perform multiplication or division. Therefore, we trade-off the flexibility of having non-discretised weights, which enables the success of modelling the SIR data in Heim et al. (2020, Section 4.1), in favour of sparse models with discrete weight values. All the modifications can also be generalised for the NPU architecture.

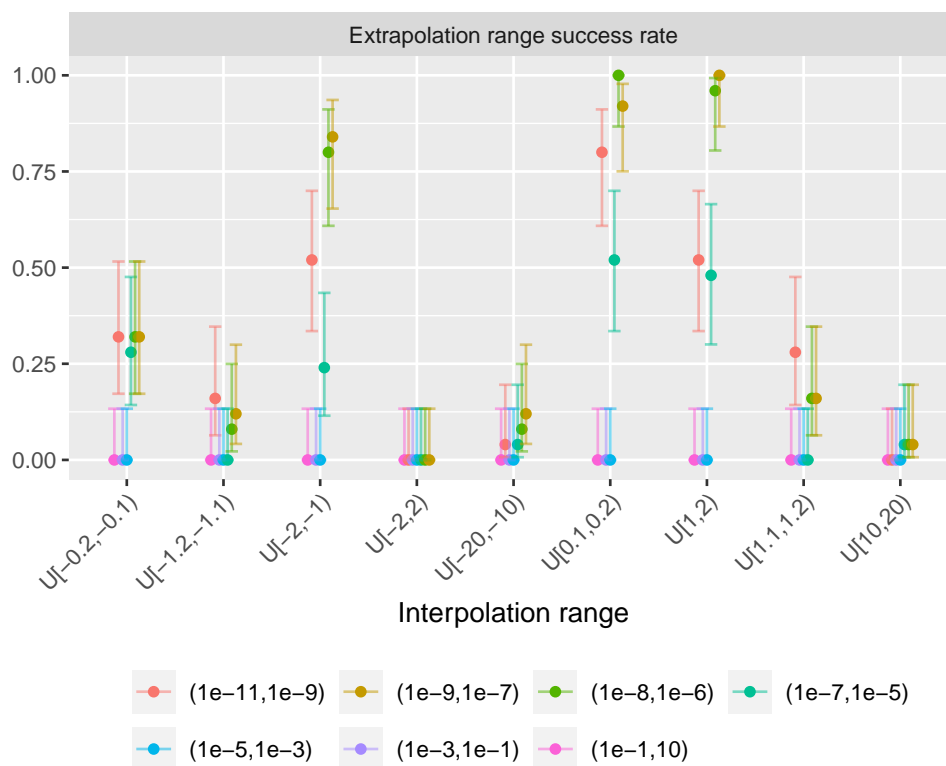
**Is  $L_1$  regularisation required? (Yes.)**  $L_1$  encourages sparsity (i.e., zero weights) in solutions (Tibshirani, 1996). For the Real NPU, zero-valued weights mean not selecting an input and returning the identity value 1. For the task, the optimal weight values require selecting all inputs and therefore non-zero values, suggesting the application of  $L_1$  could be damaging. Removing  $L_1$  regularisation (see Figure 5.4a) proves to be detrimental in five of the nine cases shown and only shows minor improvements in two of the nine ranges (i.e.,  $\mathcal{U}[-1.2, -1.1]$  and  $\mathcal{U}[1.1, 1.2]$ ). Hence, we keep  $L_1$  regularisation.<sup>1</sup> The  $L_1$  regularisation scaling (see Section 2.2.4 for details), requires setting the hyperparameters for the start ( $\beta_{start}$ ) and end ( $\beta_{end}$ ) scaling values. We run a sweep over six different start and end values, denoted ( $\langle start \rangle$ ,  $\langle end \rangle$ ), displaying results in Figure 5.4b. We find the configuration ( $10^{-9}$ ,  $10^{-7}$ ) is the most successful when considering the performance on all the ranges, and larger scaling values perform worse.

**Does clipping the learnable parameters help? (Yes.)** Division and multiplication are represented by weight values of -1 and 1 respectively. The current architecture does not constrain the weights which can result in large weight values. Hence, we investigate applying clipping directly to the weight and gate values after every optimisation step. Figure 5.5a shows clipping is beneficial, with clipping on both weight and gate (or just

<sup>1</sup>Further experiments comparing  $L_2$  regularisation also found  $L_1$  to perform better (see Appendix G.3).



(A)  $L_1$  regularisation



(B) Sweep over  $L_1$  (start,end) beta parameters

FIGURE 5.4: Exploring the effect and sensitivity of  $L_1$  regularisation on the Real NPU.

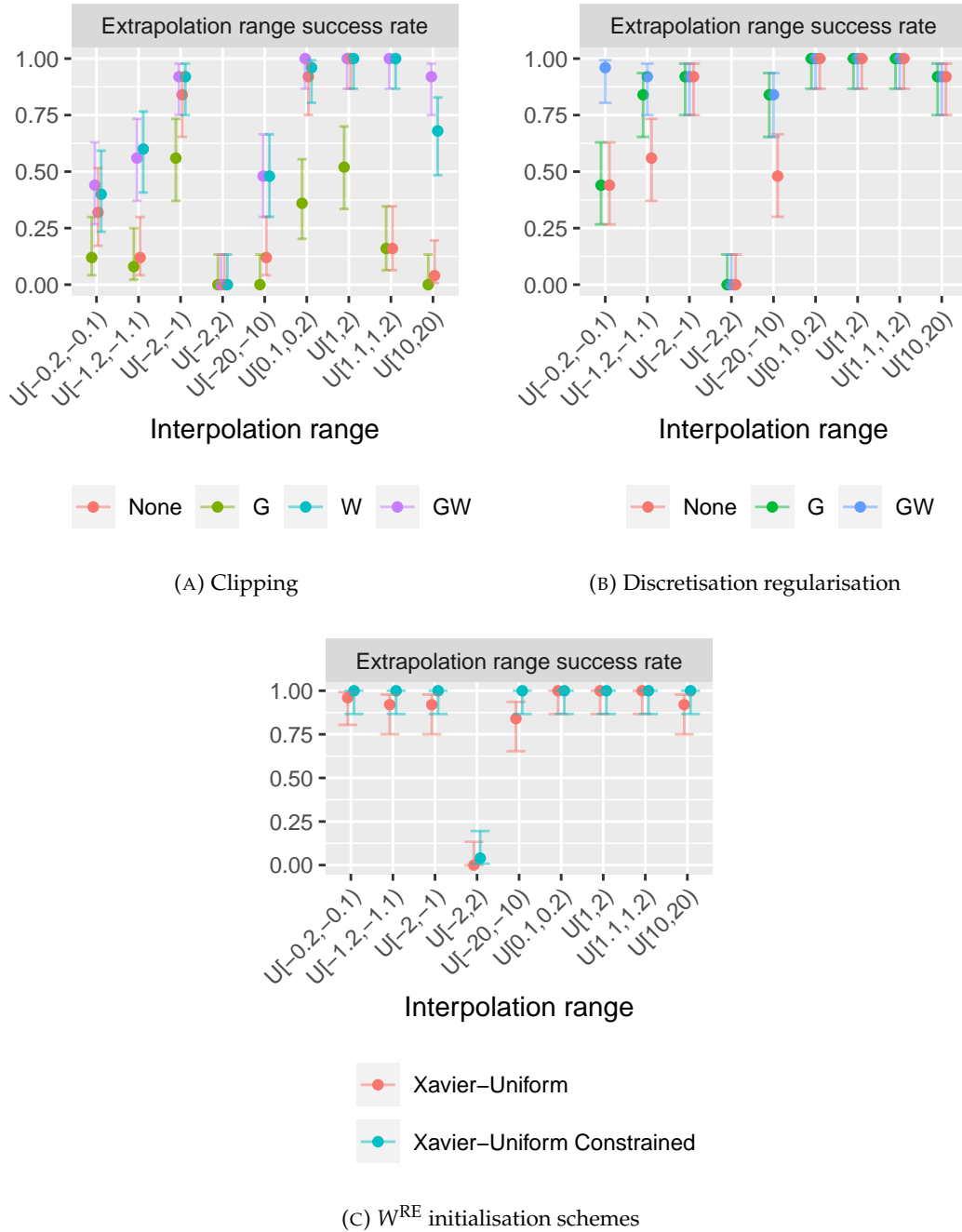


FIGURE 5.5: Effect of clipping, discretisation, and the NAU initialisation scheme on the Real NPU.

on the weights) to improve over the baseline on all ranges (excluding  $\mathcal{U}[1, 2]$  where the baseline has already achieved full success and  $\mathcal{U}[-2, 2]$  where everything fails).

**Does enforcing discretisation help? (Yes.)** Modelling division in a generalisable manner requires all learnable parameters to be discrete i.e., a value from  $\{-1, 0, 1\}$ . Using Madsen and Johansen (2020)'s regularisation scaling scheme (see Section 2.2.3), we penalise weights for not being discrete. Gate parameters should be 0 or 1 and weight



parameters should be -1 or 1. We modify the scaling factor to be  $\hat{\lambda} = 1$  and the regularisation to go from ‘off’ to ‘on’ between iterations 40,000 to 50,000. Figure 5.5b shows discretising the gate improves over the baseline but also discretising the weights is additionally beneficial.

**Does using a more constrained initialisation help? (Yes.)**  $W^{\text{RE}}$  uses a Xavier-Uniform initialisation (Glorot and Bengio, 2010), meaning weights can be initialised out of the range  $[-1,1]$ . Therefore, we use the initialisation for the Neural Addition Unit (NAU) which is a constrained form of the Xavier-Uniform that does not allow the fan values of the Uniform distribution to go beyond 0.5, meaning that no weight value will be out of the range  $[-1,1]$ . Figure 5.5c shows using the constrained initialisation provides improvements, with a learning rate of  $5 \times 10^{-3}$  working best (see Figure G.3 in Appendix G.3).

## 5.5 Uniform Range Datasets

We now compare learning Uniform ranges (Table 3.2) on all modules including the NRU and NMRU for the no redundancy and redundancy setups.

On the no redundancy setup (Figure 5.6) the NRU and NMRU achieve full success while solving the problem consistently fast and with low sparsity error, while the baseline Real NPU without modifications struggles with success on all ranges. Applying the Real NPU modifications described in Section 5.4 improves the robustness such that only range  $\mathcal{U}[-2,2)$  struggles. The NRU was also found to be especially sensitive to the learning rate when learning on negative ranges (see Appendix G.4).

Introducing redundancy (Figure 5.7) causes failure modes to arise on all modules. The baseline Real NPU produces high sparsity errors relative to the other modules suggesting a struggle with discretisation. The modified Real NPU improves over all ranges of the baseline (which were not already at full success) in terms of success, speed and sparsity (except for the sparsity in  $\mathcal{U}[10,20)$ ). To ensure that complex weights do not fix the issue, we test the NPU module with all the modifications used on the real weight matrix but find no significant improvements (see Figure G.5 in Appendix G.5). The redundancy affects the NRU the most, resulting in full failures on all the negative ranges. The NMRU is the only module with success on range  $\mathcal{U}[-2,2)$  due to its sign mechanism (see Figure G.6 in Appendix G.6). It performs well over all ranges but can be outperformed by the modified Real NPU for negative ranges.

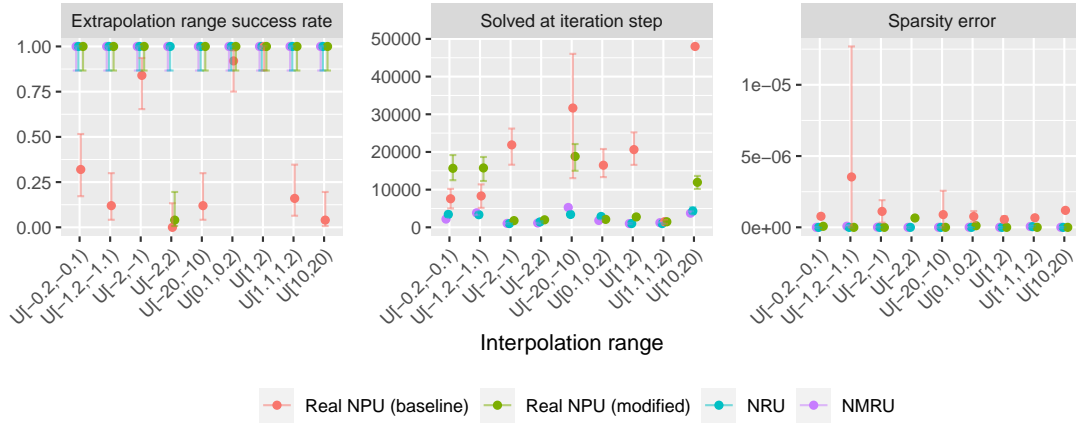


FIGURE 5.6: Division without redundancy (input size 2) on Uniform ranges.

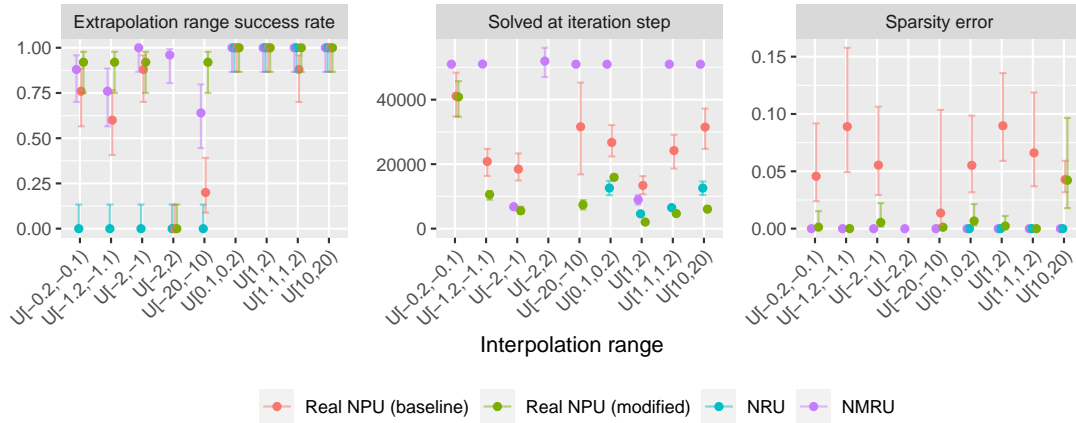


FIGURE 5.7: Division with redundancy (input size 10) on Uniform ranges.

TABLE 5.2: Mixed-Sign Datasets: The interpolation and extrapolation ranges to sample the two input elements for a single data sample. The target expression to learn is: input 1  $\div$  input 2.

DATASET	INTERPOLATION		EXTRAPOLATION	
	INPUT 1	INPUT 2	INPUT 1	INPUT 2
1	$U[-2, -0.1]$	$U[0.1, 2]$	$U[-6, -2]$	$U[2, 6]$
2	$U[-2, -1]$	$U[1, 2]$	$U[-6, -2]$	$U[2, 6]$
3	$U[-2, 2]$	$U[-2, 2]$	$U[-6, -2]$	$U[2, 6]$
4	$U[0.1, 2]$	$U[-2, -0.1]$	$U[2, 6]$	$U[-6, -2]$
5	$U[1, 2]$	$U[-2, -1]$	$U[2, 6]$	$U[-6, -2]$

## 5.6 Mixed-Sign Input Datasets

The Uniform ranges results showed that the Real NPU (modified) and NRU have difficulty in learning when inputs can consist of arbitrary signed values (e.g. all positives, all negatives, or a mixture of positive and negative values) such as  $U[-2, 2]$ .

We question if the failure is due to the input samples in a batch having different signs from each other, or if the problem is due to the fact data samples can be close to 0 (leading to singularity issues). Five mixed-sign datasets which can control the range for each element in the input are generated, with interpolation and extrapolation ranges found at Table 5.2. Datasets 1, 2, 4 and 5 samples a positive value for one input element and a negative value for the other element. Dataset 3 samples the signs randomly. Datasets 2 and 5 avoid sampling close to 0 values to mitigate the singularity issue.

Figure 5.8 shows the Real NPU struggles on all these ranges while the NRU and NMRU do not. This implies that the underlying issue is most likely correlated to each element in an input having different signs. When the denominator of the output is positive (dataset 1 or 2), the solution is found faster than when the denominator is a negative value (dataset 4 or 5). Learning with input redundancy (Figure 5.9), causes the Real NPU and NRU to swap in performance. The Real NPU performs significantly better

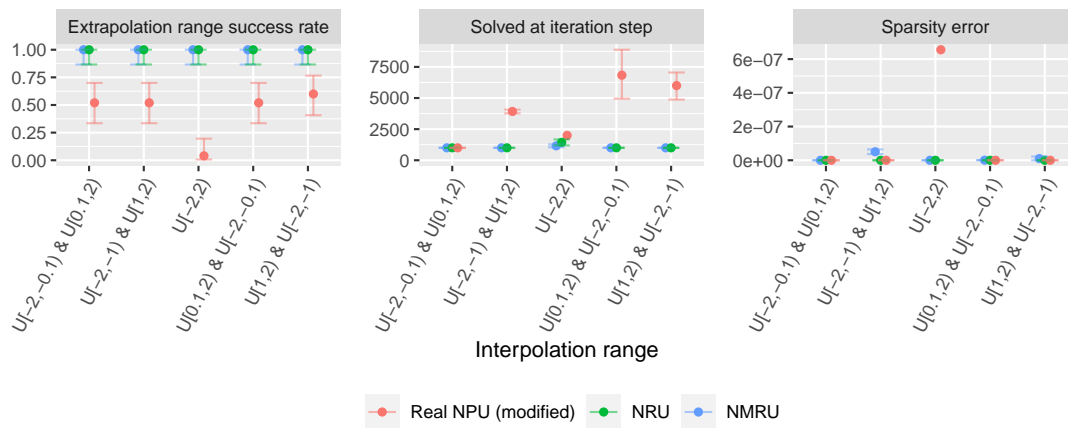


FIGURE 5.8: Division without redundancy on the mixed-sign datasets that control the sign of the input elements. The ranges are in order of the datasets (i.e. dataset 1 to 5).

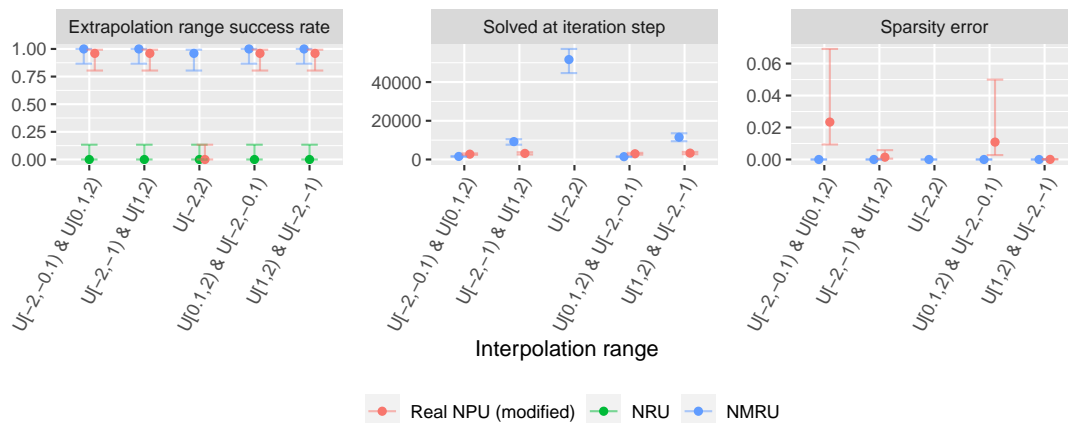


FIGURE 5.9: Division with redundancy on the mixed-sign datasets that control the sign of the input elements. The ranges are in order of the datasets (i.e. dataset 1 to 5).

than the no redundancy task on all ranges except  $\mathcal{U}[-2,2)$ , while the NRU no longer works on any range. The NMRU retains strong performance.

## 5.7 More Challenging Distributions: Larger Magnitudes and Mixed-Signs

To further stress test the modules, we explore the effect of larger Uniform ranges and different distributions (i.e., Benford and Truncated Normal). The ranges are found in Table 5.3. The Uniform ranges test how a mixed-sign dataset is influenced by larger ranges (with magnitudes of 50 and 100). The Benford distribution also tests learning on large magnitude values. It follows a more natural distribution compared to the Uniform, known to underlie real world data such as accounting data (Hill, 1995). The Truncated Normal (TN) distributions also investigate mixed-sign datasets. A Normal distribution allows setting biases via the mean value (and is set to either -1, 0 or 1), while the truncation ensures the extrapolation range not to overlap with the interpolation range. The results for the 2-input and 10-input settings are shown in Figures 5.10 and 5.11.

**Uniform distributions:** Larger ranges are found to be challenging when redundant inputs exist. On the 2-input setup, both NRU and NMRU have full success, while the Real NPU (modified) has failure cases for both Uniform distributions (with success rates of 0.72 on  $\mathcal{U}[-100,100)$  and 0.76 on  $\mathcal{U}[-50,50)$ ). On the 10-input size setup, all modules fail for all runs for both ranges.

**Benford distribution:** All modules succeed on the 2-input setting but on the 10-input setting, the NRU and modified Real NPU have full success implying the Uniform distributions failures are due to using mixed-signed inputs rather than the large ranges. The NMRU shows the majority of failures (failure rate 0.84), suggesting that distributions with large ranges are challenging for learning when using the NMRU.

**Truncated Normal distributions:** On the 2-input setup, both NRU and NMRU have full success but the Real NPU (modified) has failure cases for all three distributions (with success rates 0.48, 0.08, 0.64 respectively). When trained using the 10-input setup, both the NRU and Real NPU (modified) have no success. The NMRU's success rate greatly varies depending on the range (being 0.48, 0.04 and 0.92 for  $TN(-1, 3)[-5, 10)$ ,  $TN(0,1)[-5, 5)$  and  $TN(1, 3)[-10, 5)$  respectively). This suggests the NMRU works better when a majority of the inputs are likely to have the same sign and struggles with values around zero.

TABLE 5.3: Interpolation (train/validation) and extrapolation (test) ranges for different distributions. Data is drawn with the lower and upper bounds. TN = Truncated Normal in the form TN(mean, sd)[lower bound, upper bound). B = Benford.  $\mathcal{U}$ = Uniform.

<b>Interpolation</b>	TN(-1, 3)[-5, 10)	TN(0,1)[-5, 5)	TN(1, 3)[-10, 5)
<b>Extrapolation</b>	TN(-10, 3)[-15, -5)	TN(10,1)[5, 15)	TN(10, 3)[5, 15)
<b>Interpolation</b>	B[10, 100)	$\mathcal{U}$ [-100, 100)	$\mathcal{U}$ [-50, 50)
<b>Extrapolation</b>	B[100, 1000)	$\mathcal{U}$ [-200, -100) $\cup$ [100, 200)	$\mathcal{U}$ [-100, -50) $\cup$ [50, 100)

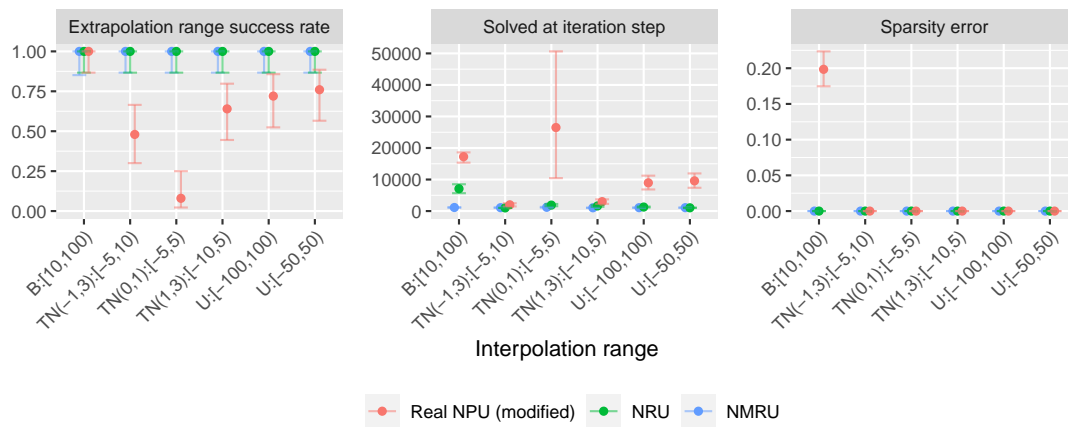


FIGURE 5.10: Division without redundancy on the Benford, Truncated Normal and Uniform distribution.

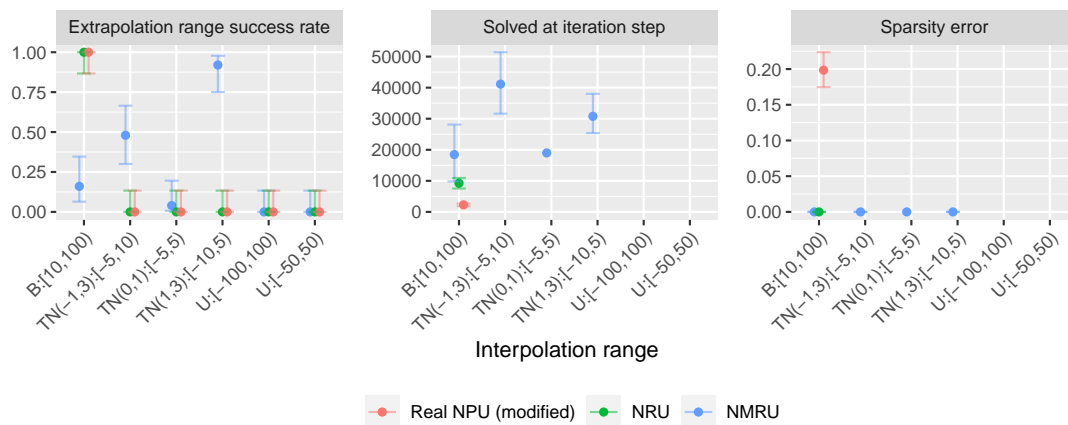


FIGURE 5.11: Division with redundancy on the Benford, Truncated Normal and Uniform distribution.

## 5.8 Division by Small Magnitudes

The discontinuous nature of division at zero results in the inability to provide a computational value for the output/gradient and causes neighbouring values to have large gradients. To understand the extent of this issue when learning, we explore learning to divide by values close to zero using three tasks with increasing difficulty: 1) learning to take the reciprocal of a single input, 2) taking the reciprocal of the first input given two inputs, and 3) dividing the first input by the second given two inputs.

### 5.8.1 Impact of the Singularity Issue on Gold Solutions

Figure 5.12 plots the test error assuming the module weights are set to the ‘gold’ solution for the three tasks. As the range values become closer to zero, the test error thresholds become increasingly large. Therefore, even with the correct weights, relying on the test errors alone as an indicator becomes increasingly deceptive with values close to zero. The Real NPU has larger test errors for all tasks and ranges, caused by adding  $\epsilon$  to the input (see Equation 2.18). Setting  $\epsilon = 0$  reduces the test error at the cost of the ability to deal with zero-valued inputs. Below, we provide the corresponding experimental results finding that only modelling reciprocals can be learnt with extremely small values.

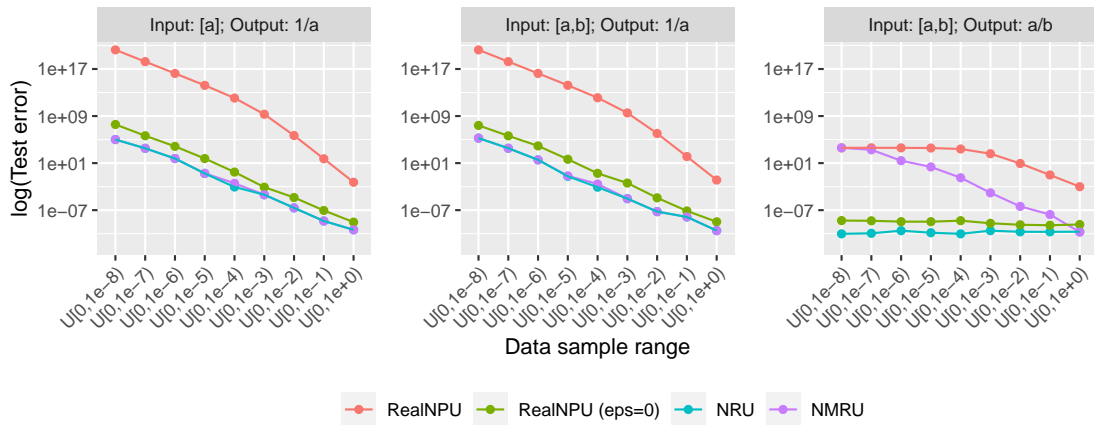


FIGURE 5.12: Effect of the singularity issue on the Real NPU, NRU and NMRU over increasing input ranges. Left: Reciprocal for an input size of 1 (no redundancy). Middle: Reciprocal for an input size of 2 (with redundancy). Right: Division for an input size of 2 (no redundancy).

### 5.8.2 Experimental Results

This section shows the results of trying to learn the reciprocal/division of values close to zero using the Real NPU, NRU and NMRU. We train and test on the ranges where

the lowest bound is 0 and the upper bounds are  $10^{-4}$ ,  $10^{-3}$ ,  $10^{-2}$ ,  $10^{-1}$  and 1. Unless stated otherwise, the hyperparameters of a model are set to what is used for the Single Layer Task without redundancy (see Section 5.3). The first task runs for 5,000 iterations with no regularisation for any module. The second and third tasks both run for 50,000 iterations. Due to precision errors, a solution with the ideal parameters will not evaluate to a MSE of 0. Therefore, we calculate thresholds that the test MSE should be within. A threshold value for a task is calculated from evaluating the MSE of each range's test dataset for each module, using the 'golden' weight values and adding an epsilon term<sup>2</sup> to the resulting error which takes into account precision errors. All experiments are run using 32-bit precision.

In general, successful runs take longer to solve as the input ranges become smaller. The simplest task, of taking the reciprocal when the input size is 1 (Figure 5.13) is achieved with ease for all modules, though for  $\mathcal{U}[0, 10^{-4})$ , we find the NRU begins to struggle.

Introducing a redundant input (Figure 5.14) greatly impacts performance with only the NMRU able to achieve reasonable success for the larger ranges. The successes shown for the Real NPU at range  $\mathcal{U}[0, 10^{-4})$  are false positives caused by the  $\epsilon$  in the architecture used for stability. Test false positives can also be indicated by the high sparsity error of the weights.

Modifying the task to division (Figure 5.15), meaning the redundant input is now relevant, shows improvement for the NMRU and NRU for the larger ranges, but the smallest ranges remain unsolved.

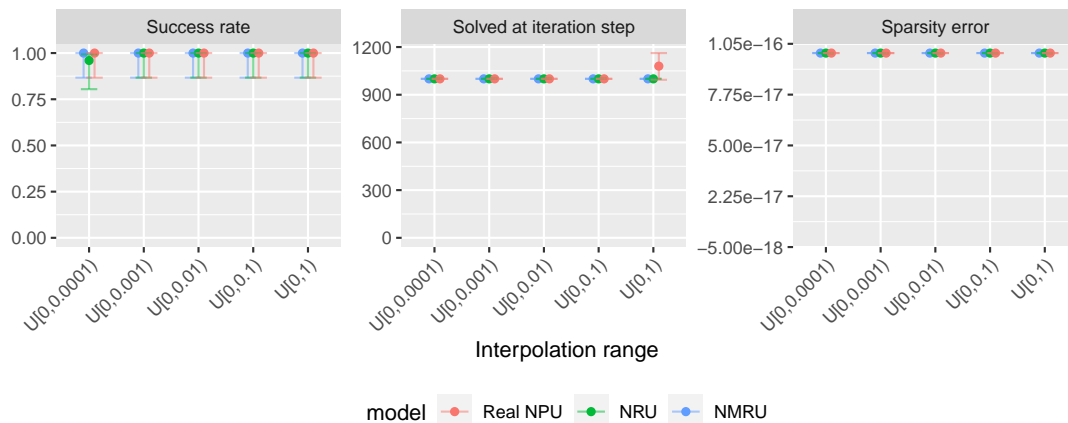


FIGURE 5.13: Input:  $[a]$ , output  $\frac{1}{a}$ . Learns reciprocal when there is no input redundancy.

<sup>2</sup>The term is the Pytorch default eps value for tensors of data type float32, `torch.finfo().eps`

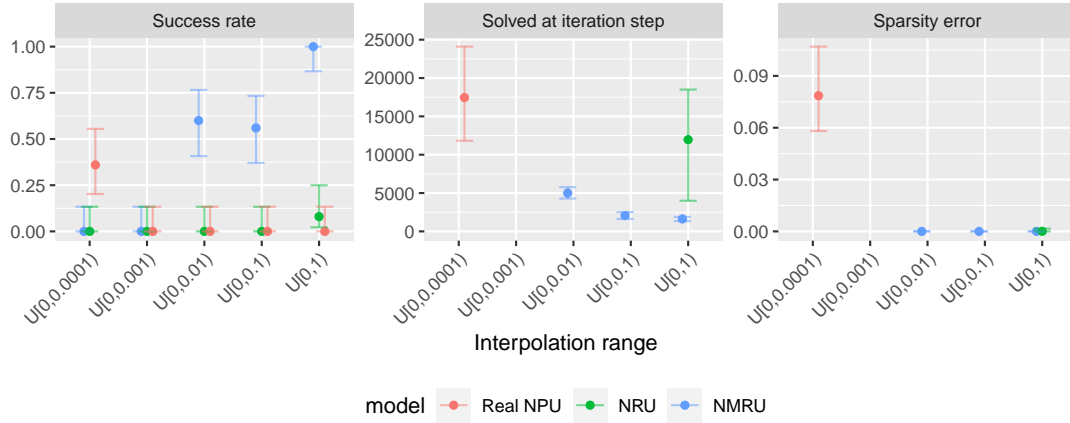


FIGURE 5.14: Input:  $[a,b]$ , output  $\frac{1}{a}$ . Learns the reciprocal of the first input when there is redundancy.

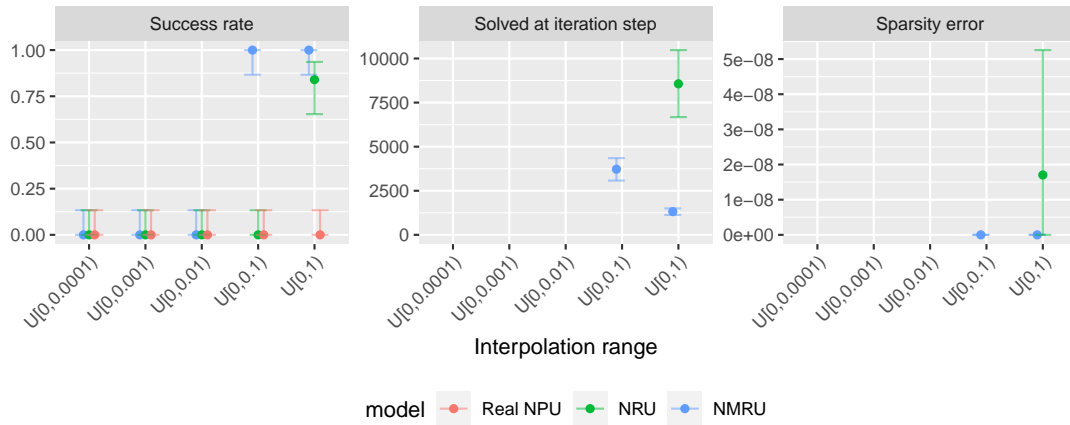


FIGURE 5.15: Input:  $[a,b]$ , output  $\frac{a}{b}$ . Learns division of the first and second value when there is no redundancy.

## 5.9 Traits of Modules when Learning on the Redundancy Setting

**Gradient difficulties with the NRU.** For insight into why the NRU performs poorly with input redundancy, we look at the gradients with respect to the weights. The partial derivative for the weights is,

$$\frac{\partial \hat{y}}{\partial w_i} = \tanh(1000w_i)(\text{sign}(x_i)|x_i|^{w_i}(\tanh(1000w_i) \log(|x_i|) + 2000 \text{sech}(1000w_i)^2) - 2000 \text{sech}(1000w_i)^2) \times \text{NRU}_{\tilde{\mathbf{x}} \in \mathbf{x} \setminus \{x_i\}}(\tilde{\mathbf{x}}). \quad (5.3)$$

The  $\text{NRU}_{\tilde{\mathbf{x}} \in \mathbf{x} \setminus \{x_i\}}(\tilde{\mathbf{x}})$  term applies the NRU to all inputs excluding  $x_i$  influencing the gradient values between subsequent update steps. Factoring out this term, the following observations are made: if  $x_i \approx 0$  and  $w_i \approx 0$  then gradients become increasingly



large; for  $-1 \leq w_i < 0$ , as  $w_i \rightarrow -1$  all gradients for  $x_i$  where  $|x_i| \gg 1$  become increasingly small; the gradients for  $x_i = -1$  and  $x_i = 1$  are 0 regardless of the value of  $w_i$ ; if  $w_i = 0$  then the gradient is 0 for all  $x_i$ , a result of using the tanh approximation; and, even if the sign and magnitude are calculated separately and then combined (see Appendix G.7) to try to control the gradient better, the problem remains. Therefore, we conclude that extending the NMU to divide using a weight of -1 is a poor choice when there are redundant inputs.

**The NMRU's and Real NPU's exploitation of multiplicative rules.** In the redundancy setting, modules with extrapolative solutions learn to exploit rules for multiplication. The NMRU exploits the inverse rule of division i.e.,  $a_i \cdot \frac{1}{a_i} = 1$ . Since the module's input contains the reciprocals numerous extrapolative solutions exist, however this comes at the cost of finding a 'simple' solution containing non-zero weights only for relevant inputs. The Real NPU exploits the rules  $a_i \cdot 0 = 0$  and  $1^{a_i} = 1$  enabling non-zero weights if the corresponding gate value is 0. However, this can be avoided by allowing 0 to not be penalised during the regularisation stage (see Figure 5.16); this alleviates the exploitation issue with no cost to performance.

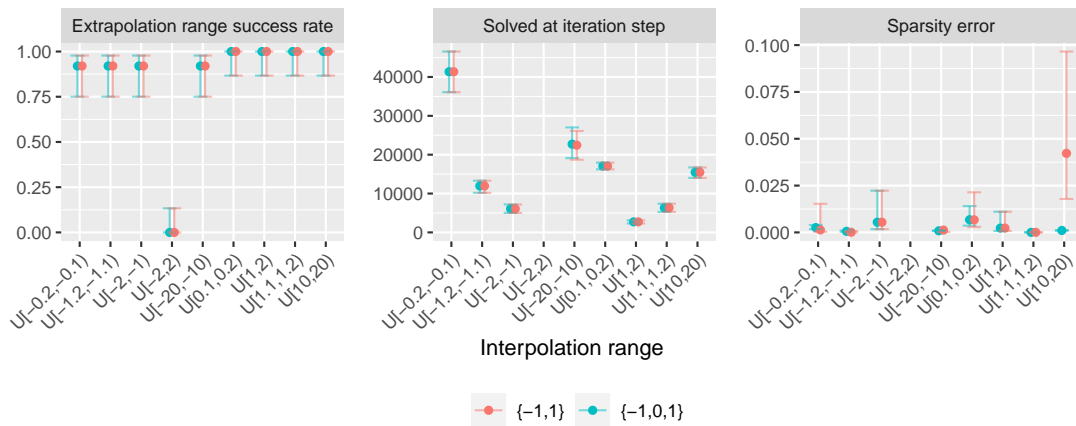


FIGURE 5.16: Comparing weight discretisation on the Real NPU weights which penalises not having a weight of  $\{-1,1\}$  vs  $\{-1,0,1\}$ .

## 5.10 MNIST Arithmetic - Isolated Digit Classification

To determine and observe how division NALMs fare when attached to additional networks, we learn to divide the labels of an image composed of two MNIST digits. A summary of the experiment setup is found in Table 5.4.

TABLE 5.4: MNIST (Isolated Digit Classification) experiment parameters.

Parameter	Two digit MNIST
<b>Epochs</b>	1000
<b>Samples per permutation</b>	1000
<b>Train:Test</b>	90:10
<b>Batch Size</b>	128
<b>Train samples</b>	72,000 (1 fold)/73,000 (9 folds)
<b>Test samples</b>	9,000 (1 fold)/8,000 (9 folds)
<b>Folds/Seeds</b>	10
<b>Optimiser</b>	Adam (betas=(0.9, 0.999))
<b>Criterion</b>	MSE
<b>Learning rate</b>	$10^{-3}$
$\lambda_{start} - \lambda_{end}$ <b>epochs</b>	30-40
$\hat{\lambda}$	2
<b>grad norm clip</b>	MLP = None; Real NPU = None; NRU = None; NMRU = 1

### 5.10.1 Setup and Network Architecture

Unless stated otherwise, assume the setup is the same as the Isolated Digit task from Section 4.6.1.1. All instances of zero are removed from the datasets to avoid a division by zero cases from occurring. The network learns a map from the input image to the labels of the two digits (digit classifier), followed by a map from the two labels to their divided value (division layer). There are three possibilities for the division layer: (1) a solved division baseline model (DIV), (2) an MLP made of 2 hidden layers with 256 hidden units and ReLU activations and  $L_2$  regularisation, and (3) a NALM being either the Real NPU (modified), NRU, or NMRU. As the DIV baseline only requires learning to classify the images to their respective labels, it is considered a strong baseline. A NALM should perform similarly to the DIV baseline; if a NALM outperforms the baseline it implies the NALM can also aid in learning downstream layers as well as learning division.

### 5.10.2 Metrics and Results

The output accuracy is given based on the predicted and target values rounded to 5 decimal places (d.p.). Results are taken over a 10-fold cross validation setting and the NALM's initialisation is the same for each fold. Table 5.5 and Figure 5.17 displays the results. The MLP is not used in the violin plot so the distributions of the other modules can be better seen. The DIV baseline performs well as expected since only the classification network requires to be learnt. The Real NPU (modified) has consistent accuracy on par with the DIV results. The NRU can outperform even the DIV baseline. The

NMRU performs the worst out of all NALMs struggling with robustness; we question if this is a result of the gradient norm clip in Appendix G.8 but find it is not. The MLP is the worst division layer showing nearly no success across all folds.

TABLE 5.5: Test accuracies of the output label for the MNIST task. The predictions and targets are rounded to 5 d.p. before the accuracy is calculated. The mean accuracy over 10-folds is given with the standard error.

	DIV	MLP	Real NPU (mod.)	NRU	NMRU
Test Acc. (5 d.p.)	$97.497 \pm 0.183$	$0.004 \pm 0.004$	$97.147 \pm 0.242$	$97.517 \pm 0.291$	$46.891 \pm 13.969$

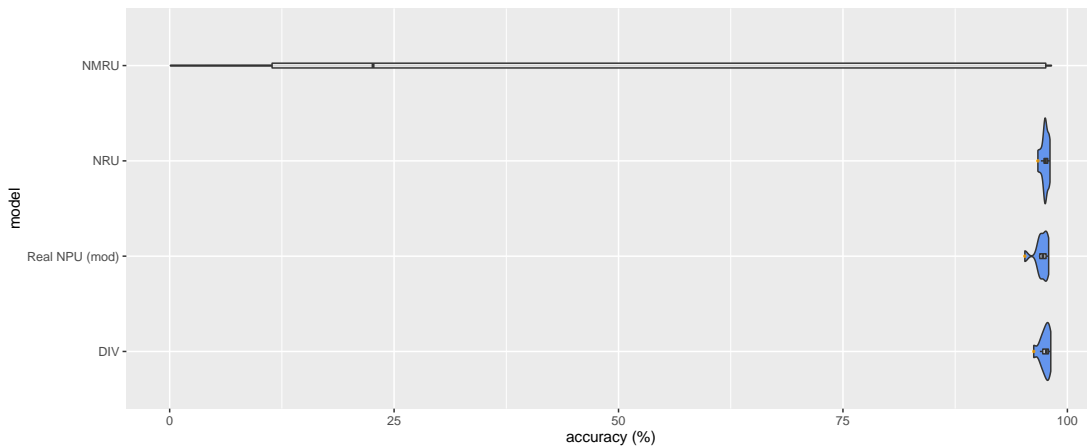


FIGURE 5.17: MNIST Arithmetic Isolated Digit task test accuracies on the rounded 5 d.p. output values.

## 5.11 Discussion

**Single layer division robustness.** We summarise the key challenges for learning independent modules in Table 5.6 and give the ranges used to generate the values in Appendix E.3.3. In the no redundancy setting (2-inputs), the Real NPU is challenged when the training data consists of mixed-signed inputs even with our applied improvements. Increasing the difficulty to have an input redundancy (with 8 redundant and 2 relevant input values) improves performance when the Uniform distribution is used but magnifies the issue when ranges are samples from a Truncated Normal distribution. The NRU and NMRU have strong performance across the no redundancy tasks but show failures when redundancy is included. In particular, the NRU loses its ability to learn successfully on most of the input settings. Negative ranges also become an issue for the NRU, in which we conclude it is not wise to use with MSE. The NMRU drops most in performance on large magnitude datasets regardless of the distribution. In the redundancy setup, the NMRU’s robustness comes at the cost of the simplicity of the solution due to its exploitation of the identity rule; an issue the Real NPU does not have. The Truncated Normal distribution causes the greatest learning difficulties

TABLE 5.6: Summary of the types division tasks the models can/cannot solve. Using redundancy means there are irrelevant inputs (10-input setup). The values are the mean success rate (out of 1) for the specific input task, bold values are the best model for the respective row.

Redun- dancy?	Input type	Distribution	Real NPU (modi- fied)	NRU	NMRU	Figure
No	Mixed-signs	Uniform	0.44	<b>1</b>	<b>1</b>	5.8
	Mixed-signs	Truncated Normal	0.61	<b>1</b>	<b>1</b>	5.10
	Negative	Uniform	<b>1</b>	<b>1</b>	<b>1</b>	5.6
	Positive	Uniform	<b>1</b>	<b>1</b>	<b>1</b>	5.6
	Large magnitude	Uniform	0.74	<b>1</b>	<b>1</b>	5.10
	Large magnitude	Benford	<b>1</b>	<b>1</b>	<b>1</b>	5.10
	Close to 0	Truncated Normal	0.08	<b>1</b>	<b>1</b>	5.10; see TN(0,1)[-5, 5)
	Close to 0	Uniform	0	0.17	<b>0.4</b>	5.15
Yes	Mixed-signs	Uniform	0.77	0	<b>0.99</b>	5.9
	Mixed-signs	Truncated Normal	0	0	<b>0.48</b>	5.11
	Negative	Uniform	<b>0.92</b>	0	0.82	5.7
	Positive	Uniform	<b>1</b>	<b>1</b>	<b>1</b>	5.7
	Large magnitude	Uniform	0	0	0	5.11
	Large magnitude	Benford	<b>1</b>	<b>1</b>	0.16	5.11
	Close to 0	Truncated Normal	0	0	<b>0.04</b>	5.11; see TN(0,1)[-5, 5)

for all the modules. Learning to divide values around zero remains challenging for all modules, even on the no redundancy setup, implying an alternate method for dealing with zero denominators should be open for exploration.

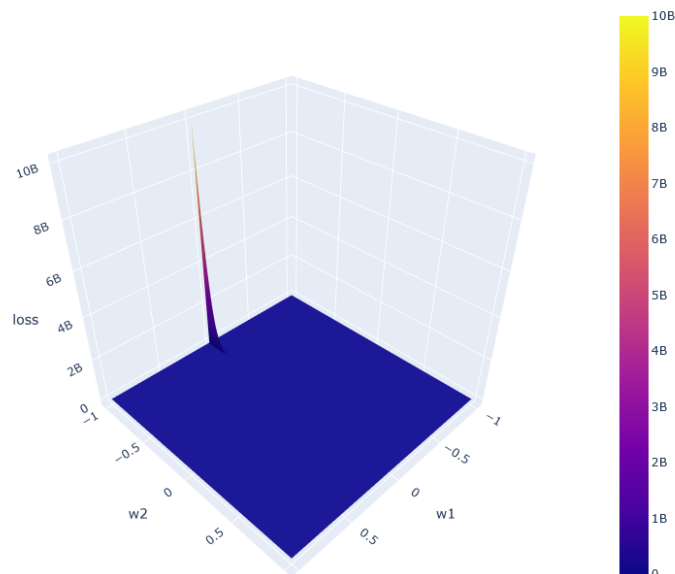
**NALMs can be used as part of larger networks.** The MNIST experiment shows NALMs can act as downstream layers in a non-trivial regression experiment which requires an intermediary classification network without a direct classification loss. This is promising as it implies the use of NALMs in more complex tasks, however two points of caution should be considered. Firstly, the results show that there is not a direct correlation between the performance of a NALM in the single layer tasks to their performance if embedded in larger networks. For example, the NRU and NMRU which outperform the Real NPU on the single layer tasks perform worse in the MNIST task. Secondly, if such units are to be utilised in larger embedded networks, we encourage performing tests in the target domain before employing NALMs in the wild. Therefore, a future direction for this work and NALMs, in general, includes developing more challenging experimental tasks with rigorous evaluations.

**Number of learnable parameters.** The NRU requires  $I \times O$  parameters, the Real NPU requires  $I(O + 1)$  parameters and the NMRU requires  $2I \times O$  parameters. Although the NRU has the lowest parameter count, it performs the worst when redundancy is involved. The doubling of the input dimensionality in the NMRU results in more parameters, especially if the output dimension is high. Additionally, as half the inputs of the modules require being inverted (which includes the irrelevant elements), scaling difficulties can arise.

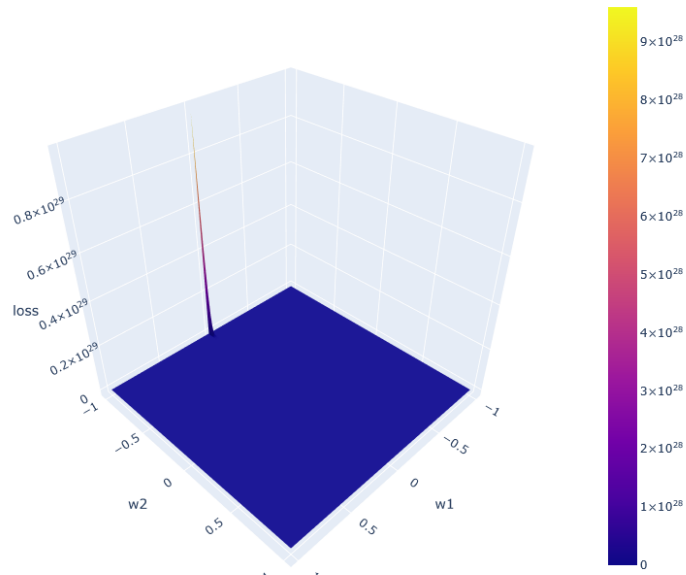
**Two-layer learning.** Once robust modules are attainable in a single layer setting, the next step would be to question performance when learning stacked modules, e.g. learning a stacked additive and multiplicative module. Madsen and Johansen (2020, Figure 2) illustrates the troubles for multiplicative models with the capacity for division. They show how a stacked summative-multiplicative module can lead to an exploding loss when the output of the summative module is close to 0 and the multiplicative model tries to divide. We recreate their setup in Figure 5.18 to produce loss surfaces for the NAU-Real NPU, NAU-NRU and NAU-NMRU, where the NAU is a summative module (see Section 2.2.3). A similar issue exists with the Real NPU and NRU which use a weight range of  $[-1,1]$ , whereas the NMRU whose weight's range is limited to  $[0,1]$  does not have exploding losses.

## 5.12 Summary

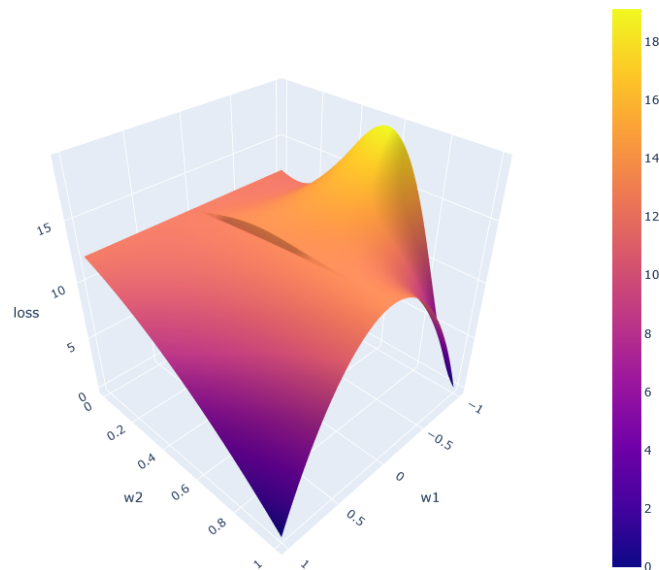
In conclusion, division remains a challenge to learn using interpretable neural networks, even for the simplest tasks. Nevertheless, by identifying the specific areas causing difficulty (e.g., training ranges), and useful architecture properties (e.g., using a sign retrieval mechanism), we hope the community has better intuition for dealing with division and developing more robust specialist modules.



(A) NAU-Real NPU (where  $\epsilon = 10^{-5}$ )



(B) NAU-NRU



(C) NAU-NMRU

FIGURE 5.18: Root Mean Squared loss landscape for the NAU stacked with either a RealNPU, NRU, or NMRU. “The weight matrices are constrained to  $\mathbf{W}_1 = \begin{bmatrix} w_1 & w_1 & 0 & 0 \\ w_1 & w_1 & w_1 & w_1 \end{bmatrix}$ ,  $\mathbf{W}_2 = [w_2 \ w_2]$ . The problem is  $(x_1 + x_2) \cdot (x_1 + x_2 + x_3 + x_4)$  for  $x = (1, 1.2, 1.8, 2)$ ” (Madsen and Johansen, 2020). The ideal solution is  $w_1 = w_2 = 1$ , though other valid solutions do exist e.g.,  $w_1 = -1, w_2 = 1$ . The NMRU’s weight matrix would be  $\mathbf{W}_2 = [w_2 \ w_2 \ 0 \ 0]$ , and the Real NPU’s  $\mathbf{g} = [1 \ 1]$ .

## Chapter 6

# Factors to Consider when Learning NALMs

Recent chapters have focused on architectural changes to NALMs in order to improve robustness. Rather than architecture related traits, this chapter focuses on other parts of the learning pipeline, namely the pre/post-processing of data and the loss criterion.

### 6.1 Feature Scaling

The earlier benchmarking experiments from Chapter 3 uncover a noticeable trait that NALMs are sensitive to the training range. This is unsurprising considering the gradient descent based approach to parameter learning, which causes the input values to have a direct effect on gradients. Due to this, neural networks ideally want features to be processed in some way to allow for the expectation of the input features to be approximately zero (LeCun et al., 2012). Doing so allows for parameters to update at the same rate and have better tolerance with larger learning rates, enabling faster convergence (Ioffe and Szegedy, 2015). A common technique for many ML workflows is the use of feature scaling in the preprocessing step of data, which scales the features into a common 'dimensionless' unit (Dick et al., 2020). The two common options for feature scaling are standardisation and normalisation. Standardisation (or z-score normalisation) scales individual features to fit a standard normal distribution with a mean of 0 and a standard deviation of 1. Normalisation (or min-max-scaling) scales individual features within a range of typically  $[0,1]$  or  $[-1,1]$ . Standardisation is most useful when the data fits a Gaussian distribution, whereas normalisation is used when the underlying distribution of the data is unknown.

Recently, there has been a resurgence in using such techniques in the genetic programming field for symbolic regression. For example, the Feyn Python library,<sup>1</sup> developed by the company Abzu<sup>2</sup> use the following min-max normalisation technique

$$\begin{aligned}\hat{X} &= \frac{2(X - \overline{X_{tr}})}{\max(X) - \min(X)} \odot \mathbf{w}^{(in)} + \mathbf{b}^{(in)}, \\ \mathbf{z} &= f(\hat{X}), \\ \hat{\mathbf{y}} &= (\mathbf{z} \odot w^{(out)} + b^{(out)}) \times \frac{\max(\mathbf{y}_{tr}) - \min(\mathbf{y}_{tr})}{2},\end{aligned}\tag{6.1}$$

where terms denoted with  $w$ 's and  $b$ 's are learnable weights for scaling and offsetting, and max and min are the max of min for each feature.

Owen et al. (2018); Dick et al. (2020); and Dick (2022) all use linear scaling with feature standardisation finding consistent improvements. The scale and offsets are calculated based on training data statistics of both the input  $X$  and target  $\mathbf{y}_{tr}$  data resulting in

$$\hat{\mathbf{y}} = \overline{\mathbf{y}_{tr}} + \text{SD}(\mathbf{y}_{tr}) \times f\left(\frac{X - \overline{X_{tr}}}{\text{SD}(X_{tr})}\right),\tag{6.2}$$

where  $\overline{\mathbf{y}_{tr}}$  and  $\overline{X_{tr}}$  are feature means over the training targets and inputs respectively, and  $\text{SD}(\mathbf{y}_{tr})$  and  $\text{SD}(X_{tr})$  are the sample standard deviations of the training data. Standardising helps reduce the effect of outliers and treats all features with equal importance. As  $f$  is applied on a standardised feature space, a reverse transformation using the target data statistics is applied to convert the output back to the original space. The combination of linear scaling and standardisation is viewed as allowing one to learn the function shape and the placement of the function in the search space respectively.

Though, the results from such symbolic regression works suggest that normalisation is possible for equation discovery, we find that such techniques are not highly compatible with NALMs. Specifically, the normalisation causes the NALM weights to deviate from their expected weights (see Table 6.1).

<sup>1</sup><https://docs.abzu.ai/>

<sup>2</sup><https://www.abzu.ai/>



TABLE 6.1: Resulting weights learned from training a normalised NMU model using the Feyn min-max normalisation. The fixed NMU fixes the NMU weights to  $[1 \ 1]^T$  and learnt NMU will learn the NMU weights. The output for these weights will model  $x_1 \times x_2$ .

Term	Fixed NMU	Learnt NMU
$\frac{2}{\max(X) - \min(X)}$	$\begin{bmatrix} 0.649115 \\ 0.261765 \end{bmatrix}$	$\begin{bmatrix} 0.649115 \\ 0.261765 \end{bmatrix}$
$\overline{X_{tr}}$	$\begin{bmatrix} 2.15035 \\ 5.67394 \end{bmatrix}$	$\begin{bmatrix} 2.15035 \\ 5.67394 \end{bmatrix}$
$\mathbf{w}^{(in)}$	$\begin{bmatrix} 0.784557 \\ 0.669749 \end{bmatrix}$	$\begin{bmatrix} 1.46126 \\ 2.13898 \end{bmatrix}$
$\mathbf{b}^{(in)}$	$\begin{bmatrix} 1.09510 \\ 0.994739 \end{bmatrix}$	$\begin{bmatrix} 0.631827 \\ 0.765059 \end{bmatrix}$
NMU weights	$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0.415312 \\ 0.293096 \end{bmatrix}$
$\frac{\max(\mathbf{y}_{tr}) - \min(\mathbf{y}_{tr})}{2}$	11.9184	11.9184
$w^{(out)}$	0.939748	1.29786
$b^{(out)}$	$1.46041e^{-8}$	$-1.29726e^{-8}$

Training a NMU on a two-input multiplication task using a single batch of data with the normalisation from Equation 6.1 can learn to multiply, as

$$\hat{X} = \begin{bmatrix} 0.948526x_1 - 1.40783 \\ 0.559912x_2 - 2.41185 \end{bmatrix},$$

$$\begin{aligned} \mathbf{z} &= (0.393934x_1 - 1.19209e^{-7})(0.164108x_2 - 2.98023e^{-8}) \\ &= 0.0646477x_1x_2 - 1.17401e^{-8}x_1 - 1.95632e^{-8}x_2 - 3.55271e^{-15}, \end{aligned}$$

$$\begin{aligned} \hat{y} &= 15.4685(0.393934x_1 - 1.19209e^{-7})(0.164108x_2 - 2.98023e^{-8}) - 1.54613e^{-7} \\ &= x_1x_2 - 1.81602e^{-7}x_1 - 3.02613e^{-7}x_2 - 1.54613e^{-7} \\ &\approx x_1x_2, \end{aligned}$$

however, the resulting NMU weights would not converge to the expected  $[1 \ 1]^T$  but to  $[0.415312 \ 0.293096]^T$ . We assume no discretisation is applied, otherwise, the resulting NMU weights would become  $[0 \ 0]^T$ . In contrast, if the NMU weights were

fixed to their expected solution then the learnt weights and biases would be quite different:

$$\hat{X} = \begin{bmatrix} 0.509268x_1 - 1.19209e^{-7} \\ 0.175317x_2 - 1.19209e^{-7} \end{bmatrix},$$

$$\begin{aligned} \mathbf{z} &= (0.509268x_1 + 1.19209e^{-7})(0.175317x_2 + 1.19209e^{-8}) \\ &= 0.0892834x_1x_2 + 6.07095e^{-8}x_1 + 2.08994e^{-8}x_2 + 1.42109e^{-14}, \end{aligned}$$

$$\begin{aligned} \hat{y} &= 11.2003(0.509268x_1 + 1.19209e^{-7})(0.175317x_2 + 1.19209e^{-7}) + 1.74057e^{-7} \\ &= x_1x_2 + 1.74057e^{-7}x_1 + 2.3408e^{-7}x_2 - 1.74057e^{-7} \\ &\approx x_1x_2. \end{aligned}$$

Using the Single Layer Task (see Chapter 3) for multiplication, we conduct empirical studies on applying normalisation to the NMU, finding that both z-score normalisation and Feyn normalisation resulted in no successes on all nine training ranges. Plotting the errors in Figure 6.1 shows only the (no feature scaled) NMU can achieve errors within the extrapolation thresholds and consistently low sparsity errors. Clearly, there is little difference when regularisation is and is not used. Even with regularisation on, the sparsity errors are never as low as the (no feature scaled) NMU suggesting that an optimum solution is not achieved. The min-max based normalisation shows slightly lower extrapolation errors over the z-score feature scaling for positive ranges but not enough to have any extrapolative solutions. Furthermore, unlike the (no feature scaled)

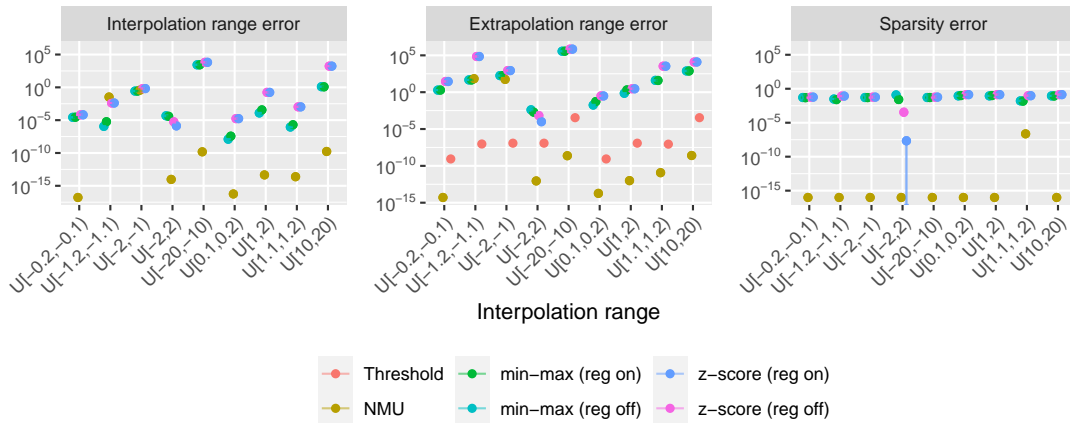


FIGURE 6.1: Learning multiplication on the two input Single Layer Task using normalisation. Mix-max normalisation refers to Feyn normalisation (Equation 6.1) and z-score refers to standardisation with linear scaling (Equation 6.2). The ‘Threshold’ refers to the minimum acceptable threshold for the extrapolation errors to be considered low enough for the NALM to succeed in the extrapolation range. The ‘reg off’ and ‘reg on’ refers to if discretisation regularisation is active on the NMU.

NMU which maintains a similar error magnitude for both interpolation and extrapolation ranges, the normalisation methods have much larger errors when comparing interpolation and extrapolation ranges.

## 6.2 Uninformative MSE Loss

In Chapter 4 we attributed the lack of robustness of NALMs (specifically the NMU) to being stuck in local minima and suggested the use of stochasticity to alleviate the issue. However, from the two-layer arithmetic task from Section 4.5 we found that certain training ranges remained unsuccessful when using reversible stochasticity. We now consider an additional factor that can lead to this behaviour, namely the loss. We shall show how these failures can be due to the input range causing difficulty for the NAU's (addition module's) weights to select the relevant inputs by causing uninformative loss values. This is explained by the following scenario. Imagine the same task but with input size four and overlap and a subset ratio of 0.5. Like before the aim is to select and add two different subsets of the input and multiply them together. For this specific case, we want the first subset to sum the 2nd and 3rd elements and the second subset to sum the 3rd and 4th elements. Consider two inputs, one sampled from  $\mathcal{U}[1,5]$  ( $i_1 = [1, 2, 3, 4]$ ) and another from  $\mathcal{U}[1.1,1.2]$  ( $i_2 = [1.11, 1.12, 1.13, 1.14]$ ). Using  $i_1$  as input and assuming weights select the correct inputs and converge as expected, we get the following:

$$\begin{aligned} f_{\text{NMU}}(f_{\text{NAU}}(i_1, \mathbf{W}^{(\text{NAU})}), \mathbf{W}^{(\text{NMU})}) &= f_{\text{NMU}} \left( f_{\text{NAU}} \left( [1 \ 2 \ 3 \ 4], \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \right), \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \\ &= f_{\text{NMU}} \left( [5 \ 7], \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \\ &= 35 \end{aligned}$$

Now consider cases: 1) NAU selection is correct but one weight did not converge, and 2) NAU selection is incorrect for one element and that weight did not converge. For each case a valid example of the NAU weight matrix ( $\mathbf{W}^{(\text{NAU})}$ ) is:

$$\text{case 1: } \begin{bmatrix} 0 & 0 \\ 0.9 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} ; \quad \text{case 2: } \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 0.9 & 1 \end{bmatrix} .$$

TABLE 6.2: The output values and absolute errors for the simplified 2-layer task with inputs  $i_1$  [1, 2, 3, 4] and  $i_2$  [1.11, 1.12, 1.13, 1.14]. Correct selection means the NAU module selected the correct inputs. Correct weights mean the weights for the NAU converged to the correct values.

CASE	$i_1$ OUT	$i_1$ AE	$i_2$ OUT	$i_2$ AE
SELECTION ✓ WEIGHTS ✓	35	0	5.1075	0
SELECTION ✓ WEIGHTS ✗	33.6	1.4	4.85326	0.25424
SELECTION ✗ WEIGHTS ✗	46.2	11.2	4.89412	0.21338

Calculating the output and the absolute error (from the ideal solution) of these cases for both inputs (Table 6.2) shows that  $i_2$  ([1.11, 1.12, 1.13, 1.14]) has a much smaller difference in error than  $i_1$  ([1, 2, 3, 4]). The model will struggle to differentiate between the correct and incorrect selection of weights for the input drawn from the distribution with a smaller range. This specific case also shows that for  $i_2$ , the selection of an incorrect input and non-converged weight gives a lower error than the case with the correct selection and non-converged weight, suggesting that the MSE calculation cannot differentiate between a better and worse solution.

The gradients of the loss also contribute to the learning of sub-optimal NAU weights. Considering the stacked NAU-NMU, the gradients of both the NAU and NMU weight matrix are scaled by a residual factor ( $\mathbf{y} - \hat{\mathbf{y}}$ ) since

$$\begin{aligned} \text{MSE : } \mathcal{L} &= (\mathbf{y} - \hat{\mathbf{y}})^2 \\ \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} &= -2\hat{\mathbf{y}}'(\mathbf{y} - \hat{\mathbf{y}}) . \end{aligned}$$

Input ranges with little variance can be trained to small training errors which give the illusion of a solved model. Therefore, by multiplying the small residual, the gradient gets scaled to very small values. If reversible stochasticity is used (e.g., the NAU-sNMU) then the gradients of the weights can take a different trajectory to the NAU-NMU's gradients. The trajectory from the sNMU can therefore escape some local minima which causes the NMU to get stuck. However, as there is no guarantee of convergence to the global optima, the sNMU may still converge to other local minima. Furthermore, as the residual term still remains, small gradients can still persist.<sup>3</sup>

<sup>3</sup>For full derivations see Appendix H (specifically H.3 and H.4).

TABLE 6.3: The properties of different loss functions.

	MSE	PCC	MAPE
Batch mean	✓	✓	✓
Difference of prediction from target	✓		✓
Standardisation		✓	✓
Mean centering		✓	

### 6.3 Alternate Losses: PCC and MAPE

Intuitively, these failures can be interpreted as the loss function being unable to provide an informative signal to learn from. Different losses will inherently have different properties which influence learning (Cherkassky and Ma, 2004; Chicco et al., 2021). Therefore we consider training the NALMs with a different loss. We explore the effects of two additional losses, the Pearson’s Correlation Coefficient (PCC) (Equation 6.4), and the Mean Absolute Precision Error (MAPE) (Equation 6.5). The properties of each loss are summarised in Table 6.3. Note that for a fair comparison between the losses, the validation and testing will continue to use the MSE as a measure of performance.

We define the PCC loss as

$$\begin{aligned}
 v_{x,i} &= (\hat{y}_i - \bar{\hat{y}}), & s_x &= \sqrt{\text{clamp}\left(\frac{1}{N} \sum_i v_{x,i}^2, \epsilon\right)} \\
 v_{y,i} &= (y_i - \bar{y}), & s_y &= \sqrt{\text{clamp}\left(\frac{1}{N} \sum_i v_{y,i}^2, \epsilon\right)} \\
 r &= \frac{1}{N} \sum_i \left( \frac{v_{x,i}}{s_x + \epsilon} \cdot \frac{v_{y,i}}{s_y + \epsilon} \right)
 \end{aligned} \tag{6.3}$$

$$\text{pcc loss} := 1 - r \tag{6.4}$$

where  $N$  is the batch size, and the means ( $\bar{\hat{y}}$  and  $\bar{y}$ ) are taken over the batch.  $\epsilon$  is used to provide better numerical stability. The clamping function ensures that the minimum value of the first argument of the clamp function is  $\epsilon$ . The correlation value  $r$  will be in the range  $[-1,1]$ .  $-1/1$  is a perfect negative/positive correlation respectively and  $0$  is no correlation between the predicted  $\hat{y}$  and target values  $y$ . Intuitively the numerator of  $r$  enables translation invariance from the mean centering, and the denominator enables scale invariance from the standardisation. Minimising the PCC loss enforces a positive correlation between the predicted and target values.

The MAPE loss is defined as

$$\text{mape loss} := \frac{1}{N} \sum_i \left( \frac{|y_i - \hat{y}_i|}{y_i} \right). \tag{6.5}$$

This loss can be thought of as a weighted mean absolute error, where the weight for each loss prediction is  $\frac{1}{y_i}$ . Unlike the MSE, the MAPE loss is scaled by the target value, meaning that the input magnitude has less influence over the gradients. However unlike the PCC loss, there is no mean centering, hence the position of the target value will have some impact.

For the remaining parts of this section, we will test the losses on the Arithmetic Dataset Task which is the original task where we first observed the issue of the MSE loss. Then, we extend the empirical studies to the Sequential MNIST Product task to see how well losses perform on a task requiring an intermediary classification and end with also studying the losses on the Single Layer Task with input redundancy for learning division modules.

### 6.3.1 Arithmetic Dataset Task

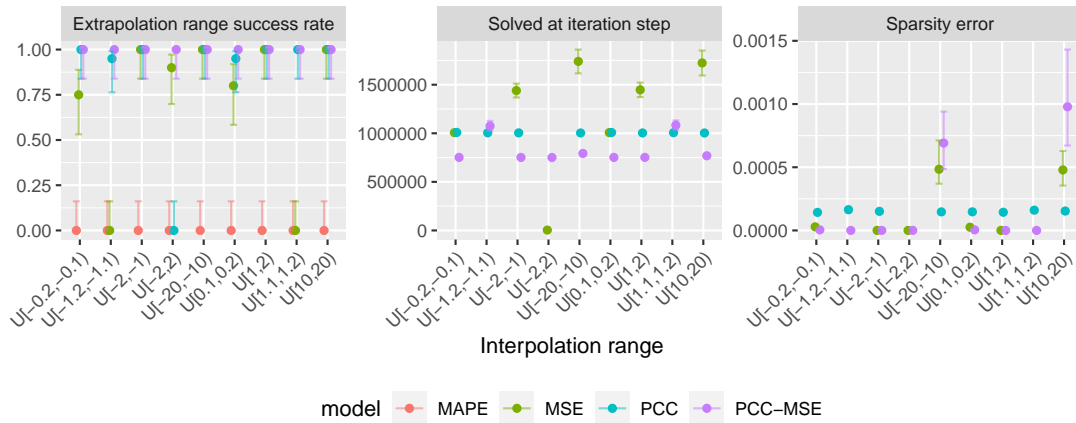


FIGURE 6.2: Two layered Arithmetic Dataset Task on the NMU for different losses.

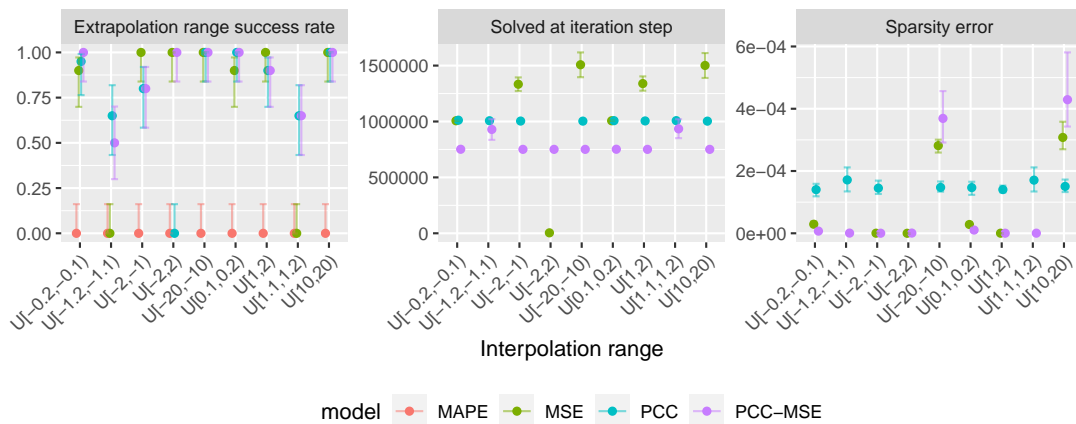


FIGURE 6.3: Two layered Arithmetic Dataset Task on the sNMU for different losses.

Figures 6.2 and 6.3 display the results for the NMU and sNMU respectively. The MAPE had no successes for either module. In contrast, the PCC based losses were effective for

the NMU but not as much for the sNMU. Initial experiments indicated that the PCC loss alone results in solutions where weights tend towards the correct discrete values but the final weights are not discrete enough even with regularisation. In contrast, a MSE loss can discretise well but does not select the correct values to discretise to. Therefore, we also test the PCC-MSE loss which initially uses a PCC loss and then switches to a MSE loss at 750,000 iterations. Using the PCC-MSE loss resulted in successes in all the tested ranges, whereas all other losses have at least one range with no success.

### 6.3.2 Product of Sequential MNIST

Figures 6.4 and 6.5 show the results for calculating the cumulative product using different losses on the NMU and sNMU respectively. The low sparsity errors indicate that the NALMs are able to converge to discrete values well for all losses for successful runs. The MAPE shows little compatibility when stochasticity is used in the NALM, whereas the MSE works well with stochasticity. Using the PCC-MSE loss on either the NMU or sNMU degrades performance on most extrapolation lengths, only outperforming a MSE trained sNMU for sequence lengths 12 and 15.

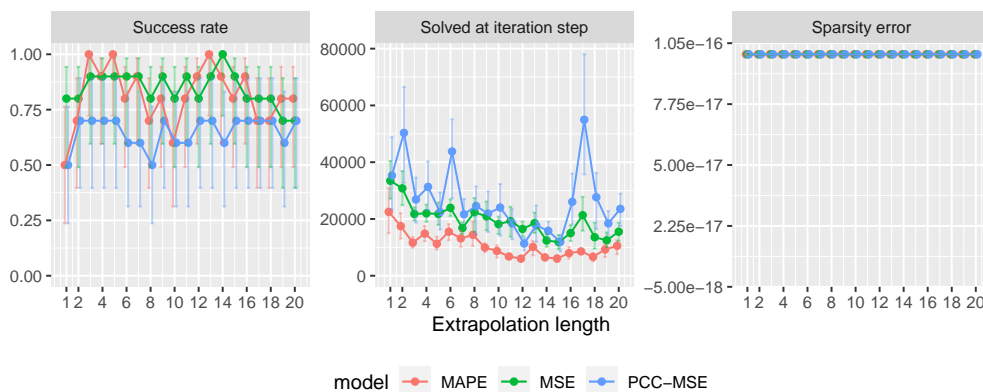


FIGURE 6.4: Performance on Product of Sequential MNIST on different losses using a NMU cell. All models use CNN architecture to convert the MNIST images into labels.

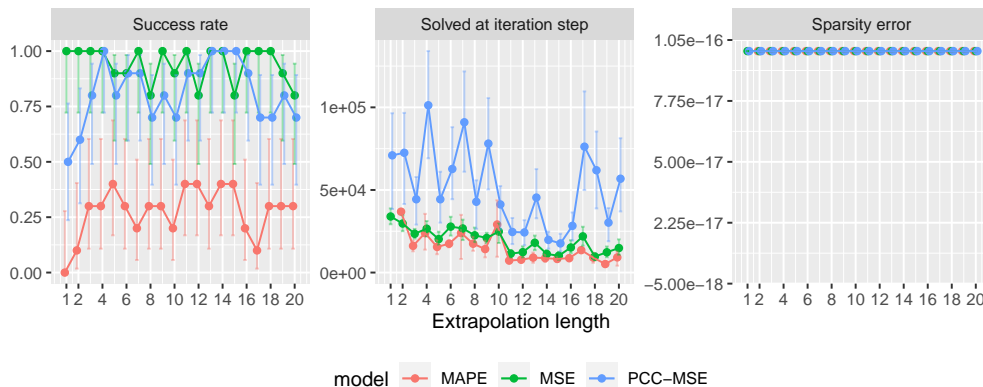


FIGURE 6.5: Performance on Product of Sequential MNIST on different losses using a sNMU cell. All models use CNN architecture to convert the MNIST images into labels.

### 6.3.3 Division: Different Losses on the Single Module Task (with Redundancy)

This section observes the effect of losses on the three division NALMs explored from Chapter 5 - the Real NPU (with our modifications), the NRU, and the NMRU.

For the Real NPU (Figure 6.6), both PCC losses and MAPE are able to get success on the  $\mathcal{U}[-2,2)$  range, which the MSE completely fails on, implying that having a loss with standardisation is useful. However, whilst gaining success in the mixed-sign range, the other negative ranges have reduced success for both PCC and MAPE. Both speed and sparsity retain similar performance to MSE in a majority of cases, with PCC solving especially fast for all tested ranges. There the PCC-MSE loss shows no significant difference to PCC loss, suggesting that discretisation was not difficult to achieve.

For the NRU (Figure 6.7), the different losses have little effect. All three losses perform well on the positive ranges but find negative ranges challenging. Compared to the Real NPU, the PCC loss on the NRU takes longer to converge to a success for negative ranges.

For the NMRU (Figure 6.8), all losses perform reasonably well, with the PCC-MSE struggling the most. Unlike the other units,  $\mathcal{U}[-20,-10)$  causes the most trouble, whereas  $\mathcal{U}[-2,2)$  gains near full success on three of the four losses.

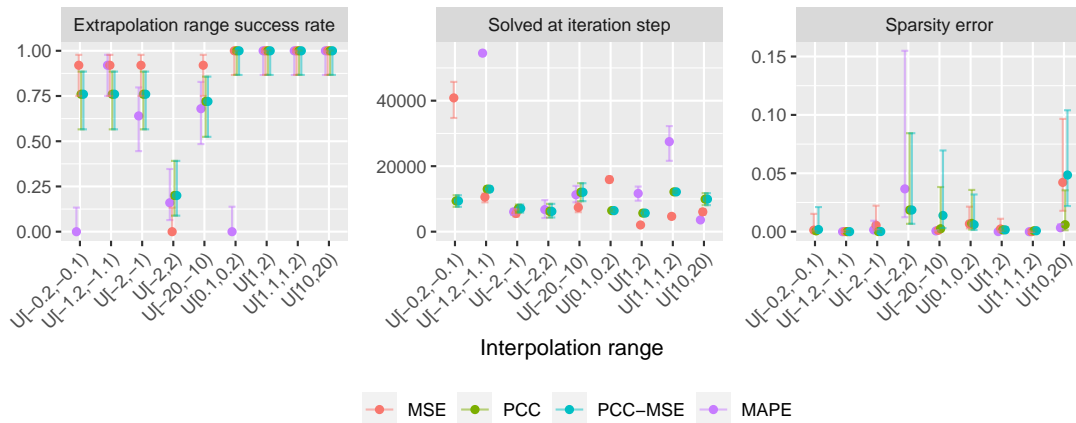


FIGURE 6.6: Single Module Task with redundancy on the Real NPU, comparing different loss functions.



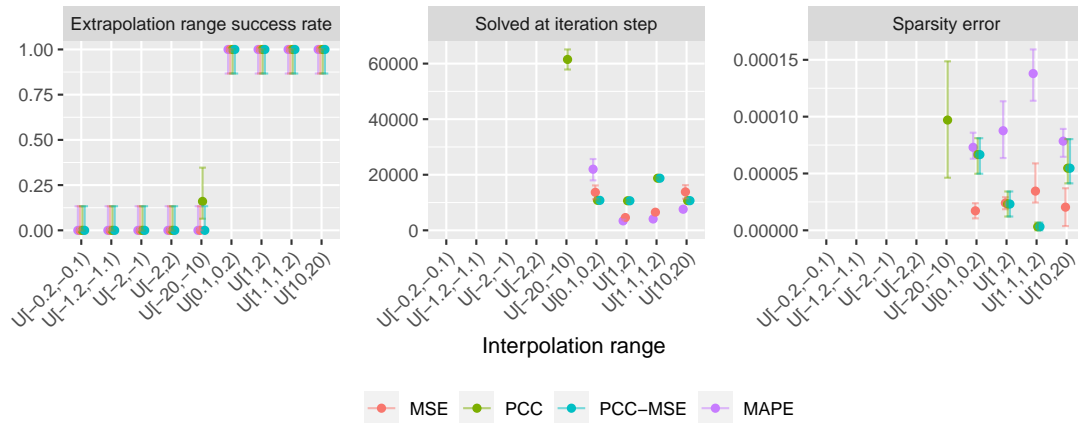


FIGURE 6.7: Single Module Task with redundancy on the NRU, comparing different loss functions.

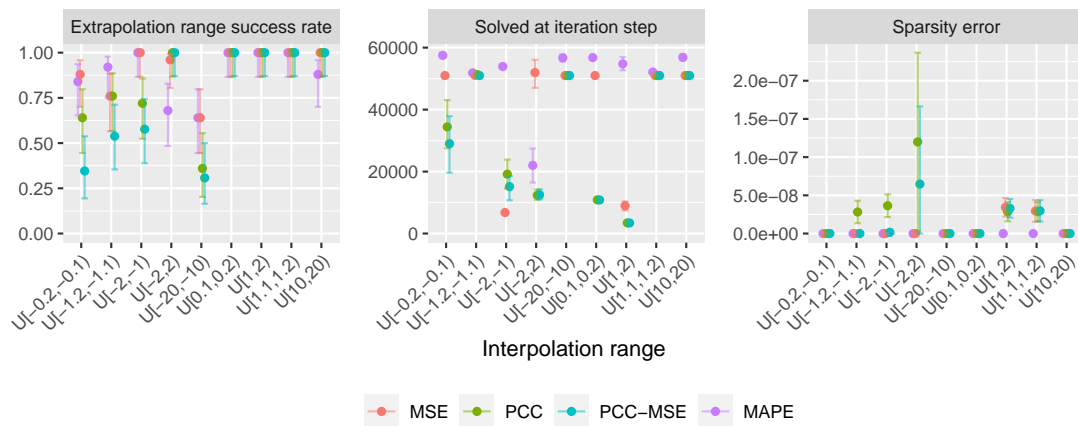


FIGURE 6.8: Single Module Task with redundancy on the NMRU, comparing different loss functions.

### 6.3.4 Summary

In this chapter, we have looked into the effect of feature scaling and using alternate losses when training. Both methods require adapting the training pipeline, but not the NALM architecture. Feature scaling, which has been applied in symbolic regression to improve convergence, was found to not work well with the arithmetic bias of NALM weights.

In contrast, the choice of loss is significant for the task. There is no ‘one for all’ loss which will work well for all modules/tasks and therefore should be set accordingly. For example, the PCC loss can improve performance on pure arithmetic tasks but is not much aid in the MNIST tasks. The MSE generally works well over different tasks but can struggle with robustness on different training ranges. The MAPE is the most volatile of the losses, with no success on the Arithmetic Dataset Task, but comparable to the MSE on the Sequential MNIST task when using the NMU.



## Chapter 7

# Compositionality - Learning Multi-Step Operations

Previous chapters have focused on considering NALMs as single units. However learning single operations in isolation is limited in terms of expressiveness and application. Ideally, we want access to a range of operations which can be combined in different ways. Therefore, in this chapter, we explore ways in which to combine NALMs to achieve arithmetic composition.

For composition based tasks, networks must learn to select the relevant modules and inputs in the correct order, all while learning the module parameters for specialisation. In recent work, [Mittal et al. \(2022a\)](#) shows even with biases towards modular architectures, learning specialisation for individual modules and selecting the relevant modules remain a challenge. A problem analogous to the ‘what came first - the chicken or the egg?’ question occurs with choosing the order to learn the structure of the network (i.e. input and module selection) and the module weights/coefficients. Some have suggested using two separate steps in learning the two parts, learning the structure first and then optimising the parameters ([Kommenda et al., 2020](#); [Chen, 2020](#); [Li et al., 2022](#)), whereas others have suggested learning both at the same time ([Martius and Lampert, 2017](#); [Kamienny et al., 2022](#)). Some even attempt to tackle the issue by creating two loss functions; one for optimising the gates and the other for the weights ([Makkuva et al., 2020](#)). Though recent studies indicate that separation of structure and weight parameter learning is best for architectures which take a GP approach ([Cava et al., 2021](#)), the most reliable approach when using an end-to-end gradient based approach remains an open question.

The choice of specialists can heavily influence the architecture and training regime. For example, [Martius and Lampert \(2017\)](#) who introduce the original Equation Learner architecture (an end-to-end differentiable feed-forward network that uses arithmetic

operations as activation functions), had not included functions with singularities/restricted domains such as division or logarithms due to their challenging learning properties. Though later works incorporated such operations, there were heavy assumptions made on the architectures such as assuming that division would occur only in the last layer (Sahoo et al., 2018).

To increase chances of selecting the correct expression, methods can keep a repository of candidates which represent promising expressions (Martius and Lampert, 2017; Chen, 2020; Werner et al., 2021; Kamienny et al., 2022). Similarly, for Transformer based methods, a decoder would create multiple candidates using beam search (Wiseman and Rush, 2016). However, there is no guarantee that the correct expression will exist/be selected from these candidates. Furthermore, maintaining multiple candidates can be expensive especially if a candidate is an entire network instance and therefore relying on a repository would ideally be avoided.

The scope of this work considers compositionality from the viewpoint of building architectures consisting of multiple NALMs. However, this is not the only type of compositionality. Another view of compositionality can also be considered where NALMs are composed with neural networks of different architectures. Two such approaches are (1) a stacked approach which connects the NALM to the input/output of other networks, or (2) building NALMs into the networks. We have touched upon the first stacked approach. For example, in Section 2.4 we provide numerous applications where NALMs are used alongside other neural architectures, mainly as part of more complex end-to-end tasks. In Chapters 4 and 5 we also investigate such forms of composition through the MNIST arithmetic tasks where a NALM is attached to the output of CNN/LSTMs digit recognition networks. The second approach remains open for future work. Such an approach includes building NALMs into Transformers. For example, replacing the residual connection in the Transformer with a NALM. This would enable the Transformer to have a bias towards performing precise arithmetic operations while keeping the overall architecture differentiable and retaining the flexibility of the Transformer’s context-based routing.

## 7.1 Task

This task requires learning operations of the form  $op_2(op_1(a, b), c)$  given an input vector  $[a, b, c, d]$ , where  $op_1, op_2 \in \{+, -, \times, \div\}$  are cumulative operations. Therefore, division represents the reciprocal operation e.g.,  $div(x_1, x_2) = \frac{1}{x_1 x_2}$ . Learning these multistep expressions can be thought of as a symbolic regression problem, which has been proven to be an NP-hard (Virgolin and Pissis, 2022). There are 16 different combinations to learn (see Table 7.1). The ability to do multi-step reasoning with a selection of the relevant operations and input elements is required in order to complete this task

TABLE 7.1: All 16 possible combinations of the two operation expressions written out in the expanded form.

$op_1$	$op_2$	expanded expression
add	add	$a + b + c$
	sub	$-a - b - c$
	mul	$(a + b)c$
	div	$\frac{1}{(a+b)c}$
sub	add	$-a - b + c$
	sub	$a + b - c$
	mul	$(-a - b)c$
	div	$\frac{1}{(-a-b)c}$
mul	add	$ab + c$
	sub	$-ab - c$
	mul	$abc$
	div	$\frac{1}{abc}$
div	add	$\frac{1}{ab} + c$
	sub	$-\frac{1}{ab} - c$
	mul	$\frac{c}{ab}$
	div	$\frac{ab}{c}$

successfully. Furthermore, some expressions can have multiple solutions meaning the steps taken can be ambiguous. For example,  $(a + b) - c$  has associative properties and therefore can also be learned as  $a + (b - c)$ . The interpolation range for training and validation is  $\mathcal{U}[1,2)$  and the extrapolation range is  $\mathcal{U}[2,6)$ . If a method uses stochastic techniques during training (e.g., Gumbel-Softmax), then during inference only a deterministic approach is used (e.g., Softmax). Hence, training can use different types of categorical sampling methods but inference will always use maximum likelihood (for module selection). A summary of the experiment parameters is given in Table 7.2.

TABLE 7.2: Experiment parameters used for the two-step compositionality task.

Parameter	Value
Epochs	200000
Learning rate	0.001
Optimizer	SGD
Batch size	128
Regularisation start epoch	100000
Regularisation end epoch	150000
NALM weights regularisation scale	10
Real NPU gate regularisation scale	5

## 7.2 Methods

This section will explain the architectures used for the experiment. As the layout of modules can influence learning (Bahdanau et al., 2019), we explore using stacking, gating, and timestep dependant routing of NALMs. Illustrations of all NALM based compositional architectures are given in Figure 7.1.

### 7.2.1 MLP

A six-layered neural network with 16 hidden units per layer using ReLU activations.

### 7.2.2 Quadratic Network

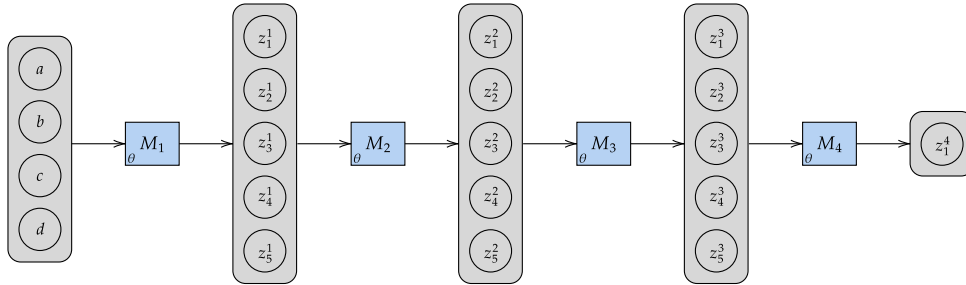
A two-layered quadratic neuron architecture (Fan et al., 2018). A quadratic neuron replaces the inner product of linear neurons with the quadratic function resulting in increased expressiveness. A quadratic layer is expressed as

$$f(\mathbf{x}) = (\sum_{i=1}^n W_{i,j}x_i + b_j) \cdot (\sum_{i=1}^n W_{i,k}x_i + b_k) + \sum_{i=1}^n W_{i,l}x_i^2 + c. \quad (7.1)$$

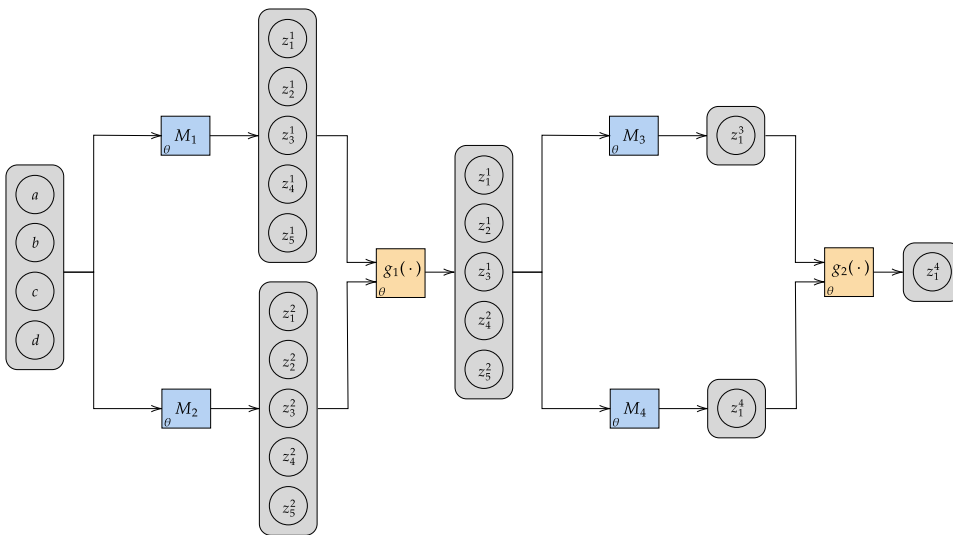
Assuming an identity activation function, the two-operation task for any combination of addition, subtraction and multiplication operations can be accomplished with two layers with two hidden units. To our knowledge, division cannot be expressed in a generalisable manner using quadratic layers with such a configuration.

### 7.2.3 Stacked NALMs

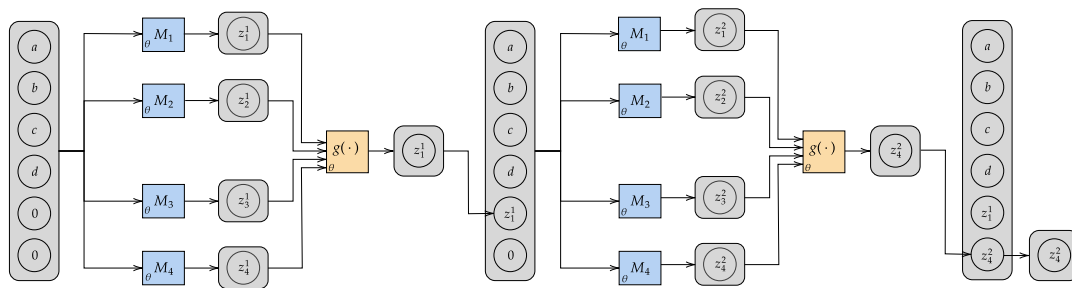
A stack of individual NALMs in a sequence. We use four layers consisting of either [iNAC, iMNAC, iNAC, iMNAC], [NAU, Real NPU, NAU, Real NPU], or [NAU, NMU, NAU, NMU]. The Real NPU modules are the modified versions which we introduce in Chapter 5 in Section 5.4. Even though the task requires a two-step operation, four layers are used to avoid any assumptions on the type of operation to occur first. For example, to do the mul-add tasks using the [NAU, NMU, NAU, NMU] setup requires having to use the second (NMU) and third modules (NAU). If only a two-layered structure was used then the network must have the order [NMU, NAU] meaning other compositions such as add-mul can no longer be learned. Each intermediate layer contains five outputs.



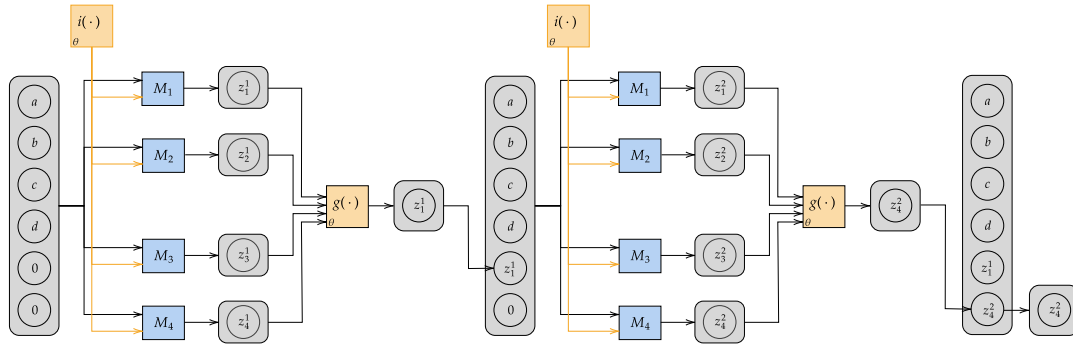
(A) Stacked NALMs



(B) Stacked Gated NALMs



(C) Recurrent Input Selector with Learnable NALMs



(D) Recurrent Input Selector with Frozen NALMs

FIGURE 7.1: Example NALM based architectures for solving the two-step multi-operation tasks assuming the two relevant modules (in order) are  $M_1$  and  $M_4$ . All architectures assume four element inputs (with two additional inputs for models allowing for lines of working) and 1 output.  $\theta$  means the block of calculation contains learnable parameters.  $M_i$  are arithmetic modules,  $z_i^j$  are module outputs where  $i$  is the output neuron index and  $j$  is the module index,  $g_i(\cdot)$  are module selection modules, and  $i(\cdot)$  are input selection modules. The Recurrent Input architectures have been unrolled for two timesteps.

## 7.2.4 Stacked Gated NALMs

A two-layered stack of gated NALMs. A gated NALM can select between NALMs per layer using the sigmoid gating technique from Schlör et al. (2020), i.e.,  $\sigma(g) \cdot m_i(\mathbf{x}) + (1 - \sigma(g)) \cdot m_j(\mathbf{x})$ . A two-layered stacking of two gated NALMs includes: [iNAC;iMNAC, iNAC;iMNAC], [NAU;Real NPU, NAU;Real NPU], or [NAU;NMU, NAU;NMU] where ‘;’ represents a gated NALM. Similar to the Stacked NALMs, the output of the first layer will have five outputs.

## 7.2.5 Recurrent Input Selector with Learnable NALMs

A key change to the input processing is made to this architecture compared to the previous NALM based architectures. The task is reconsidered as a recurrent problem where the input is extended with a scratchpad containing intermediate calculations called ‘lines of working’. This means that at each timestep, the entire input will be accessible along with an accessible memory of some intermediate calculations (i.e., lines of working). The number of lines of working in memory is equal to the number of timesteps which is set to two as the tasks are two-step operations. Every timestep the memory will be updated with a new line of working which will contain a module’s output. A layer is represented by independent NALMs which are given the same (intermediary) input and produces an output. The output for each module undergoes module selection where a single module’s output is set to the slot of the input corresponding to that timestep’s memory meaning it can be accessible in the forthcoming timesteps. For the



final time step, the output of the selected module is given as the final output rather than doing another selection on all the inputs and lines of working for two reasons: (1) it enforces a weak bias to force the final timestep to contain the output and (2) to avoid having to learn additional sparse selections which can impede gradient-based learning.

This architecture is inspired by the Global Workspace Theory (GWT) (Baars, 1993, 1997), where we consider the input (concatenated with the scratchpad) as the workspace. To write to the scratchpad's line of working segment the modules must engage in a competition where only one module is selected and allowed to write. Once written, that line of working (along with the rest of the input and scratchpad) is available to all modules in the next timestep (i.e., the information is 'broadcasted').

Each timestep still requires the NALMs to learn to select the relevant inputs and apply the operation. For module selection, the straight-through estimator/trick (Bengio et al., 2013) is used to select the output for a single module. The straight-through trick is a biased estimator which allows learning approximates of naturally discrete functions when using gradient-based backpropagation; a good alternative to the high-variance unbiased estimates which can be gained when using REINFORCE (Havrylov and Titov, 2017). The estimator creates gradients of the desired discrete function by using a differentiable (continuous) proxy function in the backward pass while the forward pass will use the discrete function. For example, if a hard gate was desired then the forward pass can use a threshold such that the gate value is either 0 or 1 and the backward pass would use the sigmoid function. Practically, the straight-through trick can be expressed in a single line of code `(hard - soft).detach() + soft`, where `detach()` represents detaching the gradient of the computation from the computation graph and the `soft/hard` represents the type of outputs the function gives. For our purposes, the straight-through trick allows the forward pass to use hard weights allowing for an exact selection of modules, whereas the backward pass uses gradients calculated using the soft weights, therefore, allowing for non-sparse gradient updates.

The module selection function also contains a learnable scalar logit (`g_ml`), which once transformed by a sigmoid operation, represents the probability of using a deterministic module selection. The probability is checked against a threshold of 0.5 to determine whether the hard selection will use the deterministic ( $p(\text{gate}) > 0.5$ ) or stochastic selection ( $p(\text{gate}) \leq 0.5$ ). A deterministic selection represents using a maximum likelihood approach to selecting the most likely module by using a straight-through Softmax, whereas the stochastic selection represents using an straight-through Gumbel Softmax (Maddison et al., 2017; Jang et al., 2017).

The straight-through Softmax will use the straight-through trick to do a hard selection of a module using `argmax` but uses the gradients generated from applying the Softmax function to the logits. The Gumbel Softmax estimator is a reparameterisation which allows one to sample a categorical (discrete) distribution ( $t \sim \text{Cat}(p_1, \dots, p_K)$ ) where

$\sum_i^K p_i = 1$ ) given logits  $\mathbf{x}$ . Intuitively, the simplest way to sample a categorical distribution would require taking the probabilities of the classes and applying the max function to get the most likely class as a one-hot vector. However, as the max function is not differentiable, we instead use the Gumbel Softmax which is differentiable.

To understand the Gumbel Softmax means to first understand the Gumbel-max trick. The Gumbel-max trick allows one to sample from the Gumbel distribution, where the reparameterisation trick is used to convert the sampling of the class to a deterministic function of parameters, with added noise sampled from variates of an independent and identically distributed Gumbel(0,1) distribution  $z_i \sim -\log(-\log(\mathcal{U}(0,1)))$ . That is,

$$\text{one\_hot}(\text{argmax}_{i \in \{1, \dots, K\}} x_i + z_i) . \quad (7.2)$$

The `one_hot` creates a one-hot vector ( $\in \mathbb{R}^K$ ) of the returned index<sup>1</sup> which can be used to select the relevant element of  $\mathbf{x}$  via a dot product. Rather than taking the max, the `argmax` is used but this is still non-differentiable. To make it differentiable, a `softargmax` is used as an approximation of the selected index:

$$\text{softargmax}(\mathbf{y}) = \sum_{i=1}^K \left( \frac{e^{y_i/\tau}}{\sum_j e^{y_j/\tau}} \right) i \quad (7.3)$$

resulting in the Gumbel Softmax. We can control how close the approximation is to the discrete distribution by controlling a temperature parameter  $\tau$ . A small  $\tau (\rightarrow 0)$  results in an approximation more similar to the discrete distribution and a larger  $\tau (\rightarrow \infty)$  is more relaxed and is similar to a Uniform distribution. This along with the straight-through trick (resulting in the straight-through Gumbel Softmax) enables doing a hard selection with usable gradients.

Intuitively, the network should initially want to explore and favour using the Gumbel Softmax, whereas when an ideal module selection is found the selection should be exploited (in favour of maximum likelihood). To initially encourage exploration the gate logit is initialised to -1 such that the  $p(\text{maximum likelihood})=0.27$  with a  $\tau$  dampening factor of 10. This procedure is used during training as shown in Algorithm 1; in inference, we only use the maximum likelihood selection in order to get a deterministic solution.

## 7.2.6 Recurrent Input Selector with Frozen NALMs

This further modifies the recurrent input selector architecture of the previous section. Rather than allowing the module to learn input selection, a separate input mask is

<sup>1</sup>For example, if  $K=3$  and the returned index was 2 then the corresponding one-hot vector is  $[0,1,0]$ .

---

**Algorithm 1:** Pseudocode for Module selection in the Recurrent Input Selector.

---

**Data:** Scalar  $g\_ml$  representing the gate logit for using a deterministic router;  
Module outputs  $\mathbf{x} \in \mathbb{R}^{[B,M]}$ ; Module selection logits  $\mathbf{w} \in \mathbb{R}^{[M,1]}$ ;  $\tau = 10$ . B is batch size and M is number of modules.

**Result:** Selection of the output of a single module.

```

/* Generate probability of using deterministic gating for the soft
and hard selections. */
1 p_det_soft  $\leftarrow \sigma(g\_ml)$ 
2 if p_det_soft > 0.5 then
3 | p_det_hard  $\leftarrow 1$ 
4 else
5 | p_det_hard  $\leftarrow 0$ 
6 end
/* Generate module selection probabilities for the
deterministic/stochastic routes. */
7 p_m_det  $\leftarrow \text{st\_softmax}(\mathbf{w}, \tau)$ 
8 p_m_stoch  $\leftarrow \text{st\_gumbel\_softmax}(\mathbf{w}, \tau)$ 
/* Apply module selection for the deterministic/stochastic routes.
*/
9 m_det  $\leftarrow \mathbf{x} @ \mathbf{p\_m\_det}$ 
10 m_stoch  $\leftarrow \mathbf{x} @ \mathbf{p\_m\_stoch}$ 
/* Apply gating to select between the deterministic and stochastic
routes. */
11 out_soft = p_det_soft * m_det + (1 - p_det_soft) * m_stoch
12 out_hard = p_det_hard * m_det + (1 - p_det_hard) * m_stoch
/* Apply straight-through trick. */
13 out  $\leftarrow \text{out\_hard} - \text{out\_soft.detach()} + \text{out\_soft}$ 
14 return out

```

---

learnt which masks out irrelevant inputs such that the modules only require to apply their specialist operation. As a result, module parameters are not required to be learnt and can be preset to the desired operation.

The input mask will be a tensor of shape [Timesteps, Inputs + Scratchpad, Modules] so for each timestep, a different input selection for the modules can be learnt meaning the same modules can be reused each timestep. The weights of the input mask are transformed using a straight-through Softmax over the module dimension such that each input element can only be available to a single module. The input mask is initialised to zeros (since Softmax does not care about starting values; only the relative differences) but the input elements corresponding to the relevant line of working for a timestep are set to one to encourage the use of the relevant intermediary value at the relevant time steps. For example, an input mask corresponding to two timesteps with four inputs (and scratchpad with two lines of working) and 4 modules would be initialised to

$$\begin{bmatrix} \begin{bmatrix} 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \end{bmatrix} & \begin{bmatrix} 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 1. & 1. & 1. & 1. \\ 0. & 0. & 0. & 0. \end{bmatrix} \end{bmatrix} .$$

The module selection strategy remains the same as before, using the straight-through trick (see Algorithm 1) to select the output of a single module.

### 7.3 Results

TABLE 7.3: Results for learning multi-operation arithmetic. There are 16 combinations in total. \* indicates that division cannot be learnt so there are at most 9 combinations. The average interpolation and extrapolation errors are given along with their standard deviation. Types of architectures include non-modular networks, stacked NALMs, stacked gated NALMs, recurrent input with learnable NALMs and recurrent inputs with frozen NALMs.

Model	Correct (/16)	Interp. MSE	Extrap. MSE
MLP (6L;16HU)	0	0.002 ± 0.00	233.012 ± 744.44
Quadratic (2L;5HU)	0	0.001 ± 0.00	724.276 ± 696.68
Quadratic (2L; 2HU)	0	0.001 ± 0.00	558.707 ± 707.69
4L Stacked [iNAC,iMNAC, iNAC, iMNAC]	0	4.071 ± 7.57	140.695 ± 307.42
2L Stacked Gated [iNAC;iMNAC]	0	0.148 ± 0.17	832.505 ± 2870.44
4L Stacked [NAU,Real NPU,NAU,Real NPU]	0	0.39 ± 0.65	2320.03 ± 9072.02
2L Stacked Gated [NAU;Real NPU]	0	0.031 ± 0.04	189.576 ± 601.80
4L Stacked [NAU,NMU,NAU,NMU]	0*	2.815 ± 4.69	574.419 ± 1333.84
2L Stacked Gated [NAU;NMU]	4*	0.067 ± 0.067	32.491 ± 55.22
Recurrent Input with [iNAC, iMNAC, iNAC, iMNAC]	0	4.071 ± 7.58	416.987 ± 1054.30
Recurrent Input with [NAU, Real NPU, NAU, Real NPU]	3	3.891 ± 6.71	396.292 ± 1172.17
Recurrent Input with [NAU, NAU, NMU]	5*	0.196 ± 0.26	136.472 ± 220.06
Recurrent Input with Frozen NALMs	7	0.891 ± 2.03	99.76 ± 193.132

Results are shown in Table 7.3. We observe that the MLP and Quadratic architectures both exhibit overfitting on the interpolation data but no form of generalisation to the extrapolative data. In contrast, a majority of all the other architectures are found to not

TABLE 7.4: 2 layer Gated NAU-NMU success and failure breakdown of the expressions learnt. The  $o_i$  values represent the selected intermediary output (starting with index 1).

Success	Task	Target expression	Layer 2	Learnt expression
TRUE	add-sub	$-a - b - c$	$o_1$	$-a - b - c$
	add-mul	$(a + b) \times c$	$o_3 \times o_4$	$(a + b) \times c$
	sub-add	$-a - b + c$	$o_1$	$-a - b + c$
	mul-mul	$abc$	$o_3 \times o_4$	$ab \times c$
FALSE	add-add	$a + b + c$	$o_2 + o_3 + o_4 + o_5$	$1 + 1 + 1 + 1$
	sub-sub	$a + b - c$	$o_2 + o_3 - o_4$	$(b + ab) - (a - 0.5b + c)$
	sub-mul	$(-a - b) \times c$	$o_1 \times o_4$	$(-a - b - c) \times c$
	mul-sub	$-ab - c$	$o_1$	$0.88(-a - b - c)$
	mul-add	$ab + c$	$o_2 + o_3$	$b + ab$

have an average interpolation loss below 0.1 implying that the failures on the interpolation range are harsher and can be better indicated without relying on the extrapolation set. Using a deep stacking of singular NALMs also exhibits no success in any operation. Reasons can include the multiple ways of solving a combination that can result in confusion for the model. For example, sub-sub which requires learning  $a + b - c$  can be learnt as  $(a + b - c)$ , or  $(a + b) - c$ , or  $(a - c) + b$ , or  $(a + b + d) - (d + c)$ , etc. Additionally, requiring discrete weights (of which many will be zeros) across multiple layers can cause difficulty in acquiring good gradients for learning. For example, for the sub-sub task using stacked NALMs containing NAUs and NMUs were found to have three of the four modules all their weights collapsed to values around zero ( $< 10^{-4}$ ). A similar case occurs for the iNALU modules where the transformed weights have an average value of 0.17.

Reducing the depth by using gating to increase the breadth of a layer allows for replacing two stacked layers with a single layer of two modules that are gated. This architectural style can get successes, however, the choice of modules used matters. That is, the gated iNALU and Real NPU based units have no success whereas the NAU and NMU can learn mul-mul, add-mul, add-sub and sub-add. Observing the composition of the learnt expressions (see Table 7.4) shows that multi-step operations can be learnt. For example mul-mul requires both layers and add-mul does the addition of the sub-parts (carrying the singleton  $c$  to the next layer) and then multiplies. The purely summative tasks (add-sub and sub-add) learn the expression in the first layer instead of learning the expression in two parts and then combining. Looking at the remaining failure shows that either input or module selection can cause failures, as well as partial gating values. An interesting failure case is the add-add which learns to output 4. This is a result of the first layer using multiplication for each selected intermediary of layer 2, where each of the corresponding NMU weights is 0 resulting in the output being 1 (a result of the NMU architecture). If in the first layer, the NAU would have been selected instead then the output would have been  $2a + 2b + c + d$  which is quite different to the

TABLE 7.5: Recurrent Input Selection NALM using learnable NAUs and NMU for add-add, sub-sub and add-mul. The timestep columns are the output expressions (i.e., the selected module’s output) for the corresponding timesteps.

Success	Task	Target expression	Timestep 1	Timestep 2	Scratchpad accessed
TRUE	add-add	$a + b + c$	0	$a + b + c$	
	sub-sub	$a + b - c$	$-c$	$a + b - c$	✓
FALSE	add-mul	$(a + b) \times c$	$a + b + c$	$a + b + c$	

desired  $a + b + c$ . What may be occurring is an example of a shortcut as the interpolation range is  $\mathcal{U}[1,2)$  meaning the average output value for the expression  $a + b + c$  is 4.5 which is close to 4. In other words, the network had taken advantage of the NMUs bias of outputting the identity value of cumulative multiplication (i.e., any input with weight 0 will be converted to 1). Similar phenomena have been observed in Section 5.9 where division modules would unexpectedly leverage division rules such as the inverse rule or zero rule.

A gated architecture suggests there is an advantage in having all modules available every layer. If recurrent inputs with scratchpads are used with learnable NALMs then the NAU-NMU model can additionally learn the combinations add-add and sub-sub but no longer succeeds on add-mul. The learnt expressions for these combinations are found in Table 7.5 displaying examples of how the model can: (1) in add-add still favour single multi-element calculations over multi-step, (2) in add-sub can use the lines of working to store and access intermediate calculations and (3) in add-mul fail in all ways by not using the scratchpad and applying the incorrect input and module selection. The iNAC-iMNAC version of this model is still unable to learn any combination but the NAU-Real NPU version can learn three combinations.

Without loss of generality, we can separate the input selection from the NALMs such that there is only a single weight per module. Furthermore, if multi-op modules are separated into separate modules, such as a separate addition and subtraction module from the NAU, then the weights can be preset to be specialists meaning only the input mask (and module mask) requires to be learnt and not the NALM weights. Doing so results in successes in division combinations as well, as shown in Table 7.6. For the successes, the scratchpad can be used when required and when unused the scratchpad would simply store identity values such as 0 or 1. Of the failures, tasks requiring subtraction were particularly challenging, failing on  $\frac{6}{7}$  possible combinations.

TABLE 7.6: Recurrent Input Selection using frozen arithmetic modules. The timestep columns are the output expressions (i.e., the selected module’s output) for the corresponding timesteps.

Success	Task	Target expression	Timestep 1	Timestep 2	Scratchpad accessed?
TRUE	add-add	$a + b + c$	1	$a + b + c$	
	add-sub	$-a - b - c$	0	$-a - b - c$	
	mul-mul	$abc$	0	$abc$	
	mul-div	$\frac{1}{abc}$	$\frac{1}{abcd}$	$\frac{1}{abc}$	
	div-add	$\frac{1}{ab} + c$	$\frac{1}{ab}$	$\frac{1}{ab} + c$	✓
	div-mul	$\frac{c}{ab}$	$\frac{1}{c}$	$\frac{c}{ab}$	✓
	div-div	$\frac{ab}{c}$	$\frac{1}{cd}$	$\frac{abd}{cd}$	✓
FALSE	add-mul	$(a + b)c$	$-a - b - c$	$-a - b - c + a + b + c$	✓
	add-div	$\frac{1}{(a+b)c}$	$\frac{1}{abcd}$	$\frac{1}{abc}$	
	sub-add	$-a - b + c$	$\frac{1}{abcd}$	$-a - b$	
	sub-sub	$a + b - c$	$\frac{1}{abcd}$	$\frac{1}{cd}$	
	sub-mul	$(-a - b)c$	0	$-a - b - c$	
	sub-div	$\frac{1}{(-a-b)c}$	0	$0 \times d$	✓
	mul-add	$ab + c$	0	$0 \times 1$	✓
	mul-sub	$ab - c$	1	$-1 - a - b - c$	
div-sub	$\frac{1}{ab} - c$	$\frac{1}{abcd}$	$\frac{1}{cd}$		

## 7.4 Summary

This chapter has explored ways in which NALMs can be combined together to solve compositional arithmetic tasks. We draw influence from concepts such as the GWT (Baars, 1993), introducing a scratchpad which can only be written if a competition is won, and globally accessible information that is broadcasted each time step. Using such concepts results in our Recurrent Input Selection architecture which can build compositional expressions over multiple timesteps. Furthermore, by having a separation of concerns between the input selection, module operation and module selection we can remove the need for heavily engineered specialists, allowing for predefined modules. In contrast to approaches such as stacking the Recurrent Input Selection are able to build compositions with the use of its scratchpad. Though future work is required to improve the selection ability of the Recurrent Input Selection in being robust to different combinations, the results suggest that using a Recurrent Input Selection based architecture is a promising direction for combining NALMs.





## Chapter 8

# Conclusions

At the beginning of this thesis, we asked the question:

*How can we learn to discover basic mathematics using ML models in a generalisable manner?*

Throughout this thesis, we have set out to understand how to create neural networks with the ability to learn extrapolative mathematics. In other words, networks that learn an underlying mathematics relationship from an input distribution and still work when applied to a different distribution where the data also contains the same underlying pattern. To this end, we have focused on a specific family of interpretable architectures, named NALMs, which have in-built biases to learn arithmetic/logic operations. We have considered ways to evaluate and benchmark the networks under the assumption that there is a goal for extrapolative models. Appreciating the challenges in learning single operations, we further explored multiplication and division by investigating techniques to improve robustness and factors which influence learning such as the choice of loss. To finish, we considered ways to combine different specialist modules in order to do compositional arithmetic. In this final chapter, we will briefly summarise the key findings of this work and discuss the remaining directions for future work.

Chapter 3 introduced a benchmark for single-layered modules and an evaluation scheme, both of which build on previous works of [Madsen and Johansen \(2019, 2020\)](#). With such tools, we were able to evaluate a majority of arithmetic NALMs on the four key operations - addition, subtraction, multiplication and division over multiple training ranges. Analysing these results, the following observations were made. Firstly, there exists a challenge in robustness to learning on different training distributions. Secondly, of all the operations, division was found to be the most challenging. Thirdly, in most cases, NALMs with the capacity to do multiple operations were unable to perform well. This chapter also answered our first research question regarding benchmarking.

RQ 1.1: What evaluation strategies can be used to best analyse the different characteristics of a NALM?

RQ 1.2: Which NALMs are robust to learning on different training distributions?

Answer: We conclude that a reasonable evaluation of NALMs can measure their ability to extrapolate, specialise fast and be interpretable. This results in our benchmarks adopting three evaluation metrics - the configuration-sensitive success rates, the speed of convergence and the sparsity errors. Although no NALM is completely robust to different distributions, we are able to identify for a given operation the best possible NALM to use.

Chapter 4 took a deeper look into the robustness issue for multiplication, using the NMU as a case study. We found how the specialist architecture can result in biases resulting in convergence to local optima during training. After which, we introduced a form of reversible stochasticity to encourage escaping the local minima without affecting the NMU's inner mechanics. Our results showed that our approach to stochasticity is more effective than other stochastic approaches. However, we discovered there exist instances where the approach is not as successful as the NMU. This suggests that although there is room for improvement, reversible stochasticity is a useful direction to consider when tackling the robustness issue of NALMs.

Chapter 5 expanded on the analysis from Chapter 3 which identified that division is an especially difficult operation to learn. We proposed simple yet intuitive alterations to the Real NPU to improve robustness and introduced two novel division modules, the NRU and NMRU. Both the NRU and NMRU questioned if one can extend the multiplicative NMU for division under the intuition that division is the inverse operation of multiplication. Through a comprehensive set of studies, we discovered degradation in extrapolative performance when redundancy or different distributions are introduced. In particular, small values around zero and the Truncated Normal Distribution were especially challenging. When integrated with larger end-to-end modules for more challenging tasks, there was no direct correlation to the performance on the synthetic arithmetic tasks, implying that only using synthetic tasks as a proxy for generalisation ability is not wise.

Chapter 6 took a step away from NALM architectures and instead focused on the effect of feature scaling and loss function on extrapolation performance. Feature scaling, though known to be an effective method in symbolic regression, was found to only hinder NALMs. In particular, the arithmetic bias that the weights represent in a NALM was found to contradict the type of solutions found when feature scaling was applied. In contrast, using an alternative loss to MSE can have a positive impact on generalisation, but is task dependent. Therefore, if working with NALMs, experimenting with different objective functions should be considered.

---

The findings from Chapters 3-6 answered our questions for RQ2:

RQ 2.1: What are significant architecture choices which influence the ability to learn?

RQ 2.2: Can modifying the ML pipeline provide performance gains in robust extrapolation?

Answer: Significant architecture choices include: (1) using a form of discretisation to encourage weights to converge, (2) inducing well-behaved gradients through linear weights, clipping or relevance gating, (3) applying stochasticity for avoiding local optima and (4) having additional mechanisms to calculate the sign of outputs (this is especially valuable for division based NALMs). Although modifying the data via feature scaling was unable to find any empirical improvements, using different loss functions can (though this is task dependent).

With the objective of exploring the composition of NALMs, Chapter 7 experimented with different methods for combining NALMs. Simple approaches such as stacking were found to be ineffective. Instead, by taking inspiration from cognition, we integrated ideas of the GWT into our recurrent input architectures. Using selective writing to input memory and broadcasting information of intermediary calculations encouraged a scratchpad-like mechanism to build expressions. Furthermore, by employing the use of the straight-through trick, we were able to do hard selection whilst still training on soft differentiable weights. The work of this chapter answered RQ 3:

RQ 3.1: To what extent does the way in which NALMs are combined affect their ability to compose?

RQ 3.2: How can we build an architecture to compose NALMs for learning different combinations?

Answer: The combination strategy of NALMs can have a significant effect. Strategies such as stacking and gating are found to have little success, whereas our Recurrent Input Selector architecture can learn some combinations due to its ability to remember relevant intermediate calculations in a differentiable manner and broadcast them.

To conclude, our efforts have concentrated on teaching neural networks to do extrapolative mathematics as a means towards understanding how to build generalisable mathematical reasoning models. Specifically, we focused on NALMs which consist of specialised modules for extrapolative arithmetic. Our work has extended the NALM field in multiple directions including evaluation, novel architectures and techniques such as reversible stochasticity and compositionality. We have shown how our work can further improve the robustness of NALMs to different training ranges and distributions as well as areas where challenges still lie.

## 8.1 Directions for Future Work

This thesis can be considered a first for the field of NALMs. Being the first means various directions for future study remain open for exploration. Therefore, in this section, we present examples of such directions for future work including:

- Alternative methods for generation of selection masks for selecting inputs and modules for NALM composition
- Improving the expressiveness of NALMs by learning scaling coefficients
- Alternative input representations such as complex inputs and multitoken representations of numbers
- A more versatile evaluation suite to measure NALMs performance with other input modalities and compositionality with other families of neural networks (e.g., Transformers)

### 8.1.1 Input and Module Selection for Compositional NALMs

In Chapter 7, we explored some starting points in ways to combine NALMs for compositional arithmetic. We suggest that the Recurrent Input Selection architecture with frozen modules should be extended in two ways. Firstly, consider alternate strategies to generate the input and module selection masks. This can include exploring methods used in Mixture of Experts models which learn many experts that specialise on a subset of the input space and select the relevant expert via (input dependant) gating (Jacobs et al., 1991; Hazimeh et al., 2021), or further improving the creation of selection masks when using the straight-through (Gumbel) Softmax (Paulus et al., 2021; Huijben et al., 2023). Secondly, learning strategies regarding the speed and order to learn the discrete input and module selections. Investigating different speeds of learning can involve having a nested optimisation of learning the parameters for the inputs and module selection. Investigating the order to learn the selections would ask the question of if there exists an advantage in learning either the inputs or modules first, or if learning both at the same time is best. The question of the best order to learn remains an open question and therefore we believe is worth exploring. More generally speaking, we empathise with the need to research ways to do (discrete) routing/selection; a task critical to many applications (Rosenbaum et al., 2019).

### 8.1.2 Learning Coefficients

The extrapolative nature of NALMs enables them to be a good candidate for symbolic regression. This work focused on NALMs with discretisation regularisation applied in

order to force values towards exact arithmetic operations (e.g, weighting coefficients of 1 or -1). For NALMs to be used for applications such as equation discovery requires also being able to learn weighting coefficients on the input variables. Therefore, we suggest two approaches to consider. The first would be replacing discretisation with sparsity regularisation (such as  $L_0$  or  $L_1$ ), allowing for weighting coefficients to be learned through NALM weights (Louizos et al., 2018; Heim et al., 2020). However, this can result in a degrading generalisation performance since NALMs are able to fall into more local minima (because of the removal of the discretisation regularisation) especially if used in a compositional form (joining many NALMs together). The second would be keeping the discretisation regularisation but integrating additional weighting parameters to learn the coefficients (Schlör et al., 2020). The latter approach is also open to exploring alternate discretisation methods of NALM parameters, such as network compression regularisation in binary/ternary networks (Hubara et al., 2016; Zhu et al., 2017).

### 8.1.3 Alternative Encoding of Numbers

NALMs assume that the input data should be in a real-valued form (e.g. 2.16, -1.43, 2, etc.) encoded using a floating point format. However, a question worth asking is whether this is the best number representation in order to teach NALMs? Numbers (such as [-23.1]) could be encoded in different bases such as base 10 ([-,2,3,1]) or in decimal form ([-,20,3,0.1]). Multitoken input representations for numbers using a scientific form could reduce the affect of input magnitudes when learning to multiply and divide numbers. The resulting operations would require only multiplying/dividing the coefficients and summing/subtracting the exponents. For example, to calculate  $11 \times 120$  using a scientific token notation (i.e., 11 as [+1.1,1] and 120 as [+1.2, 2]) results in  $(+1.1 \times 10^1) \times (+1.2 \times 10^2)$ . The calculation can deal with the sign, coefficients and exponents separately as  $\text{sign}(+, +) = +$ ,  $1.1 \times 1.2 = 1.32$  and  $1 + 2 = 3$  resulting in the output [+1.32, 3] which is  $+1.32 \times 10^3 = 1320$ . As NALMs currently assume a floating-point format, having another form of numerical encoding would result in having to adapt the NALM's architecture. Using a different encoding may have inherent learning advantages over other bases in regard to robustness to different distributions. For example, for Transformers, it has been noted that the chosen embedding format can influence performance (Charton, 2022a). However, unlike Transformers which learn high-dimensional embeddings of the numeric encodings (that can be difficult to interpret), NALMs which can systematically modify the encodings can be designed. Although NALMs currently require inputs to be in real space, extending support to complex inputs would allow NALMs to solve a wider range of arithmetic such as modular arithmetic. Not only would the NALMs be more expressive, but learning expressions for arithmetic which require multiple intricate steps in real space could be completed in a few steps in complex space.

### 8.1.4 Extension of the Evaluation Suite

Finally, we observed in Chapter 5 that good performance in synthetic arithmetic tasks may not be reflected in the more complex tasks (such as MNIST division). Therefore, we suggest a need for a more elaborate suite of extrapolation based arithmetic experiments to be able to better evaluate a NALM's ability to learn complex networks. This can include working with alternative modalities such as speech (e.g., calculating the result of someone asking to add two numbers) or being integrated with other well-known architectures such as Transformers (e.g., in a symbolic regression task allowing a Transformer to output skeleton structures and have NALMs find the coefficients).

## Appendix A

# Inductive Biases for System 2 - Generalist Architectures with Modularity

In the following, we explain examples of ML architectures which incorporate IBs for System 2 processing including modularity, sparsity and composition.

Goyal et al. (2021c) introduces Recurrent Independent Mechanisms (RIMs) which assume that the task can be decomposed into independent mechanisms which compete against each other for access to information. The RIM architecture learns composable reusable ‘modules’ (where each module is represented by a subset of a Long Term Short Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) RNN’s weight matrices) with representations where change can be localised. The architecture consists of the following steps:

1. The RIMs will compete for access to the input, from which only a subset (the top-k) will be selected.
2. The selected RIMs update their knowledge with respect to the input.
3. The RIMs undergo information sharing (i.e., communication). Only the top-k RIMs are allowed to access information from the other RIMs. This is an example of sparse interaction.

RIMs use a top-down attention (competition) process to select the relevant modules for the input. Unlike RIMs, the GWT works with both bottom-up and top-down connections. Mittal et al. (2020) introduce bidirectional and multilayered information flow

controlled via attention to allow for such two-way connections with hierarchy. However, unlike the GWT which assumes macro modules are competing (e.g., face recognition, gait recognition, object recognition, etc.) their modules focus on smaller micro models.

A limitation of the above architectures is the need for a predefined number of allowed active modules (the  $k$  value for top- $k$ ) which can read an input/contribute to the workspace. Ideally, we want the number of selected modules to be flexible and context dependent. [Rahaman et al. \(2021b\)](#) proposes an alternative called Kernel Modulated Dot Product Attention (KMDPA) which removes the need for a predefined  $k$ . Instead of a  $k$ , a threshold value for a distance metric is set and the distances of modules to inputs are learnt. If the distance is within the threshold the module is active. In contrast to the RIM's top- $k$  communication between specialists, [Goyal et al. \(2022\)](#) relaxes the elitism of specialists by using a shared workspace for communication like what is found in the GWT. Specialists still compete to write to the workspace, but the workspace would be broadcasted to *all* the specialists. By replacing the top- $k$  communication with a shared workspace, it is possible to create global coherence between different specialists.

[Greff et al. \(2020\)](#) argue that the inability to flexibly bind information of a distributed network leads to the gap between humans and Deep Neural Networks (DNNs). This binding problem causes difficulty in the segregation of entities, representing the segregation and the novel composition using these entities. This problem is also reflected in the RIMs architectures as the dynamics are fixed. A RIM module's parameters are not shared with other modules and are specialised for a particular computation; hence different modules operate according to different dynamics and are not interchangeable. [Goyal et al. \(2021b\)](#) takes inspiration from the Object Orientated Programming paradigm in which a class (schema) can be shared between different instances (objects/modules) with their own states. Taking this idea, it is possible to encode factorisation of both declarative (properties) and procedural (dynamics) knowledge, allowing for interchangeable modules. This schema/object-file factorisation model is given the acronym SCOFF. SCOFFs, like RIMS, use modularity but are far more flexible. For example, imagine modelling a system with three uniquely coloured bouncy balls which share the same underlying dynamics. A RIM would learn different dynamics for each ball, but a SCOFF would learn to reuse the different module parameters even though each ball refers to a different state. The balls would share the same underlying dynamic but are treated as independent in each module.

[Goyal et al. \(2021a\)](#) furthers the SCOFF concept by introducing sparse interactions of objects through the Neural Production System (NPS). Instead of having direct object-to-object interactions using attention, the NPS makes the interaction between objects sparse i.e., schemas (rules) are only applied to a subset of the objects. The sparsity is more sample efficient, faster to train, and has better transfer capabilities in the face of



changes in distributions. These plug-and-play style compositions allow learning modular networks with interchangeable modules meaning the input dynamics do not need to be fixed allowing for improved adaptability and transfer to changes in distributions. Furthermore, such forms of factorisation do not require to be limited to the modules. [Mittal et al. \(2022b\)](#) apply the discussed biases directly to the attention mechanism to disentangle the search and retrieval. The result allows for compositional attention which can have dynamic specialisation on the type of retrieval required. Unlike the NPS, the compositional attention mechanism is not a niche architecture and can be used as a direct replacement for attention heads found in common architectures such as the Transformer ([Vaswani et al., 2017](#)).

Multiple regions in the brain use discrete encodings of variables such as objects, concepts and actions; for example, the hippocampus ([O'Reilly and Rudy, 2001](#); [Zeithamova et al., 2012](#)), the prefrontal cortex and the sensory cortical areas ([Tsao et al., 2006](#)). [Liu et al. \(2021\)](#) find that imposing discretisation in the communication between modules encourages more robustness and independence between specialists. Communication encodings are split into segments which get discretised using K-Nearest Neighbours (KNN) against a learnable codebook of allowed vectors and recombined together for the final discretised representation. Additionally, the discretisation can be made more flexible by having a dynamic selection of the discretisation tightness (number of segments) conditioned on the input ([Liu et al., 2022](#)). Instead of a single fixed-size discretisation function, a pool of discretisation functions with varying levels of coarseness can be used. However, such discretisation methods rely heavily on the assumption that the task to model requires communication between specialists.

Up to this point, architectures have focused on how to separate knowledge into composable modules and share it. There is also the question of how the architecture itself should learn. [Goyal and Bengio \(2022\)](#) suggest an IB for System 2 learning includes having several speeds of learning, “with more stable aspects learned more slowly and more non-stationary or novel ones learned faster” to allow for fast adaptation. [Goyal et al. \(2021c\)](#) applies such learning to RIMs such that different parameters will learn at different timescales; RIM modules undergo fast inner updates to capture the changing dynamics from the task distribution, whereas the parameters for the input and module communication have slow outer updates to learn stable connectivity structures. Similarly, in SCOFFs, the OFs are allowed to change rapidly while the schemata change at a slower stable rate over time ([Goyal et al., 2021b](#)).

[Rahaman et al. \(2021a\)](#) bring together many pieces of the above works to introduce a novel self-attention architecture composed of differentiable components analogous to common programming concepts such as functions, arguments, types etc. One can learn reusable functions (modules) which can be selected without the need to predefine a top-k value (using the KMDPA) and be dynamically combined depending on the input. Dynamic adaptation enables the addition and removal of functions depending on the

task, allowing for on-fly and faster adaptation. Unlike SCOFF and NPS, this work uses a differentiable interpreter which enables parameter sharing between functions.

## Appendix B

# Representation of Numbers in Humans, Animals and Computers

In this section, we consider the different ways numerical information is encoded in humans, animals and computers.

### B.1 Humans

Numbers, in humans, can be represented in different ways. Two such ways described by Nieder (2021) include symbolic and non-symbolic representations. Symbolic representations such as number words (e.g., 'three') and numerals are unique to humans and allow for arithmetic. Non-symbolic representations can be found in most species (including humans) and allow for counting. The development of such a representation can be associated with evolution and survival. Two mental systems come into play to allow for non-symbolic representations – the (conscious) approximate number system (ANS) and the (unconscious) object tracking system (OTS). The ANS enables the estimation of quantities of elements without the reliance on language/symbols and results in two characteristics called the *numerical distance effect* where the larger the distance between two numbers the easier it is to discriminate between them and the *number size effect* where it is harder to tell apart two numbers of the same distance the larger they are (Dehaene et al., 1998). These two effects can activate regions in the inferior parietal area of the brain. If quantified, these effects can be characterised by two laws – Weber's law and Fechner's law. Weber's law states that the just noticeable difference ( $\Delta I$ ) between two magnitudes is a constant ( $k$ ) proportion of the reference stimulus' magnitude ( $I$ ), i.e.,  $\Delta I = kI$  (Kacelnik and Brito e Abreu, 1998; Kandel et al., 2013). In other words, the error will increase linearly with numerosity. Fechner's law states that our sensation of magnitudes scales logarithmically to the stimulus (Dehaene and Changeux, 1993). The OTS, unlike the ANS, can only represent a small set of numbers

(between 1 and 3/4) and is associated with our subitizing ability, a fast, errorless procedure of being able to know a quantity without counting (Anobile et al., 2016).

### **B.1.1 How do Humans Process Numbers?**

Humans have dedicated areas of the brain for processing numbers (Cohen and Dehaene, 1995; Dehaene et al., 1999). Of the four cerebral cortex lobes, the frontal lobe and parietal lobes have the most important roles regarding human number sense and number manipulation. For example, the task of repeatedly subtracting the digit three results in the bilateral activation of the two lobes (Dehaene, 1999, p. 249). The communication between the prefrontal and parietal cortices occurs via the superior longitudinal fasciculus (Matejko and Ansari, 2015). The parietal cortex contains the intraparietal sulcus (IPS) responsible for number sense such as “2 is less than 4” and “5 is before 6” (Nieder, 2021) and the dorsolateral prefrontal cortex (DLPFC) can manipulate information in working memory (Barbey et al., 2013). The inferior parietal (IP) region of both cerebral hemispheres contains neuronal circuits dedicated to the mental manipulation of number quantities. The IP has been suggested to be subdivided into microregions which specialise towards numbers, writing, space, and fingers (Dehaene, 1999, p. 190). In particular, the inferior parietal cortex (IPC) containing the angular gyrus (AG) holds mental number representations and number quantities.

One way of learning about what specific regions of the brain are responsible for is through the study of cerebral lesions, where participants undergoing the task have damage to a specific region of the brain. Such studies have indicated that the arithmetic abilities of humans are based on many specialised neuronal networks (i.e., modules) which communicate through parallel pathways (Dehaene, 1999, pp. 194–199). This modular nature allows for a division of labour where each module can specialise in a particular role such as recognising digits or accessing arithmetic facts from memory. These lesion studies also find that algebraic knowledge such as the quadratic formula is processed in a distinct part of the brain in comparison to number knowledge. Those with damage to the left subcortical (in the basal ganglia) are unable to recall their addition or multiplication tables but can still do subtraction and algebra; although solutions through arithmetic recall are not possible, solutions via counting (like how children initially learn to do) is possible (Hittmair-Delazer et al., 1994). Such studies suggest that arithmetic circuitry is also separate from those for reading and writing in language. For example, participants who suffer from pure alexia cannot read words but can read number digits and do written calculations (Dejerine, 1892; Dehaene, 1999).

In the frontal lobe, the prefrontal areas contain multiple specialist networks related to working memory, planning and error detection, which are all important to arithmetic. Particularly, the prefrontal cortex acts like a working memory. Prefrontal lesions do

not affect elementary operations but can impair ordering in which operations are executed, e.g., adding when one should multiply or mixing up intermediary results (Dehaene, 1999, pp. 199-201). In contrast, there are also those who have the opposite problem; they can complete multidigit calculations but could not learn the multiplication table (Temple, 1991). In fact, when asked to multiply two digits, the participant was found to take several seconds, giving incorrect answers for a majority of the trials.

To summarise, there is not a single dedicated calculation region in the brain for arithmetic (unlike the arithmetic processors in computers) but a distribution of highly specialised modules which communicate through multiple pathways. Individually, these modules cannot do much but together they can solve complex problems. Note that even a task as simple as multiplying two numbers requires the connection of millions of neurons distributed throughout multiple brain areas (Dehaene, 1999, p. 221).

## B.2 Animals

Our sense of numerosity is an innate property for humans which can be detected even in infants (Xu et al., 2005). Other animals have a weaker arithmetic ability than humans. In most cases, the internal representations of numbers can only approximate simple operations like addition, subtraction, and comparison (Dehaene et al., 1998). Many animals including cats, dogs, birds, and fishes use an ANS to detect numerosity (Nieder, 2021). Unlike humans, animals do not use symbolic systems such as language but can demonstrate basic numeric abilities such as the size or distance effect (Dehaene et al., 1998). Such number perception abilities stem from survival advantages (Perona, 2020). For example, animals which travel in groups such as schools of fish or flocks of birds have advantages if travelling in larger groups to reduce chances of being targeted by predators. As for the predators, knowing if your group is at a quantity disadvantage from another can avoid losing battles and injury e.g., territory battles between lion prides. Newborn chicks can perceive numbers using a mental number line like how young children have a mental representation of number magnitude which emerges without the need for explicit teaching (Rugani et al., 2015). Ants can keep an internal counter of the number of steps taken (step integration) to calculate the distance to their nest (Wittlinger et al., 2006). Evidence suggests that their distance mechanism can adapt to factors such as changes in stride length implying generalisation abilities.

## B.3 Computers

In contrast to the fuzzy number representations, computers represent data using digital binary bits of 1's and 0's. To represent values larger than 1, several bits must be used.

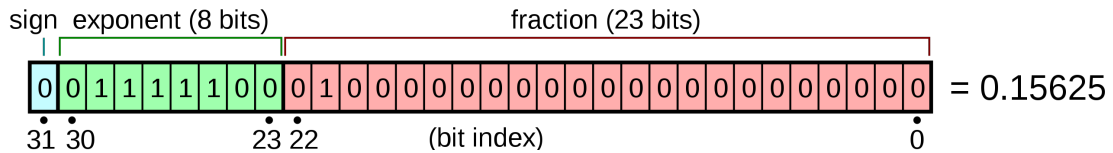


FIGURE B.1: Example of a 32-bit binary floating-point number (assuming a normalised form) representing the value  $0.15625 (= -1^S \times 2^{(E-127)} \times (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}))$ . Image sourced from [https://upload.wikimedia.org/wikipedia/commons/thumb/d/d2/Float\\_example.svg/2560px-Float\\_example.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/d/d2/Float_example.svg/2560px-Float_example.svg.png).

A collection of 8 bits, called a byte, can represent unsigned integer values from 0 to 255 or signed integer values from -127 to 127. The byte is the smallest unit which can be addressed in memory. Current computer architectures typically can have either 32-bit or 64-bit registers. To represent real numbers, a fixed-point number representation can be used. This requires having a fixed number of digits after the radix point (e.g., the decimal point if using a base 10 system). However, in modern software, a more common choice is an (IEEE-754 32-bit standard) floating-point number representation (Group et al., 2019). Unlike in fixed-point representations, the radix point is not fixed and can ‘float’ to either side depending on the magnitude of the number. The most common encoding requires three parts – the *sign* bit (S) representing if the number is positive or negative, the *exponent* (E), and the *mantissa* (F) representing the fraction. A floating-point number can be represented as  $-1^S \times F \times 2^E$  using radix 2 which is a binary base. The fractional part can be either in a normalised or denormalised form. A normalised form assumes there is an implicit leading one for the fraction, of the form  $1.F$ , meaning there is only one non-zero digit before the radix point. A denormalised form only occurs if  $E=0$  with an implicit leading 0 for the fraction allowing to represent the number 0. For the 32-bit floating-point value the breakdown would be a 1-bit sign, a 8-bit exponent and a 23-bit fraction (as shown in Figure B.1). This allows for precise arithmetic up to 6-7 digits (excluding accumulated rounding errors).

Real numbers can have an infinite number of digits (e.g.,  $\pi$ ), but numbers in computers can only be stored up to a finite number of digits using a floating-point representation. Such computational precision limitations result in rounding (round-off) errors. Rounding errors can be accumulated throughout a calculation leading to large rounding errors and a badly conditioned computation. In particular, applying floating-point operations can result in losing the correct digits of the mantissa. For an example, see Mørken (2013, p. 101, example 5.12). Two tricks can be used to avoid the accumulation of rounding errors. The first is simply using more bits to represent a real number. Converting from a 32-bit to a 64-bit floating-point representation can delay the magnitude of the rounding issue at the cost of increased memory for storing the number. The second is to rewrite the formulas used in calculations to avoid rounding errors such as  $\frac{1}{\sqrt{x^2+1}-x}$  into  $\sqrt{x^2+1}+x$  (Mørken, 2013, pp. 114-115).

## Appendix C

# Additional NALM Background Information

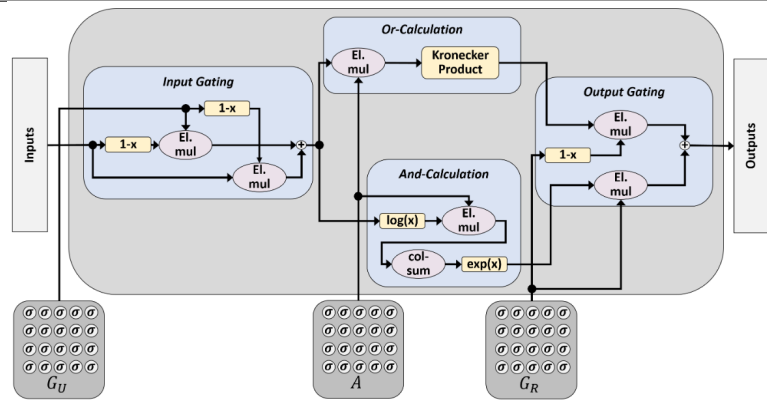
### C.1 Module Illustrations

Table C.1 displays, in chronological order, the module architecture illustrations given in their respective papers.

TABLE C.1: Module architecture illustrations taken from the original papers. \*Note that we modified the NALU architecture from Trask et al. (2018, Figure 2b) as the learned gate matrix ( $\mathbb{R}^{3 \times 4}$ ) represented as floating point numbers is mistakenly drawn as a vector ( $\mathbb{R}^3$ ) in the original figure.

Module	Architecture
NALU* (Trask et al., 2018)	

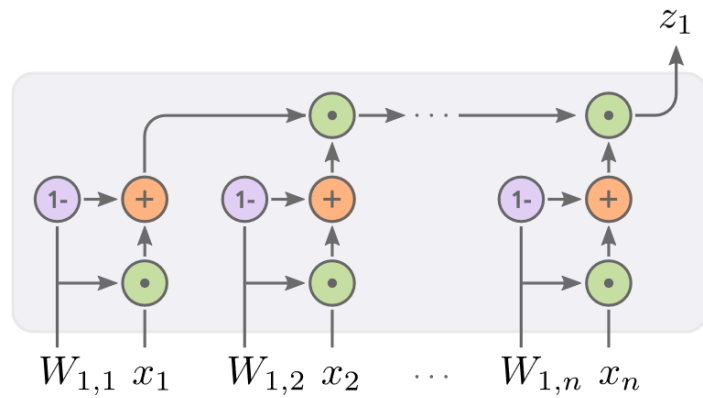
NLRL (Reimann and Schwung, 2019)



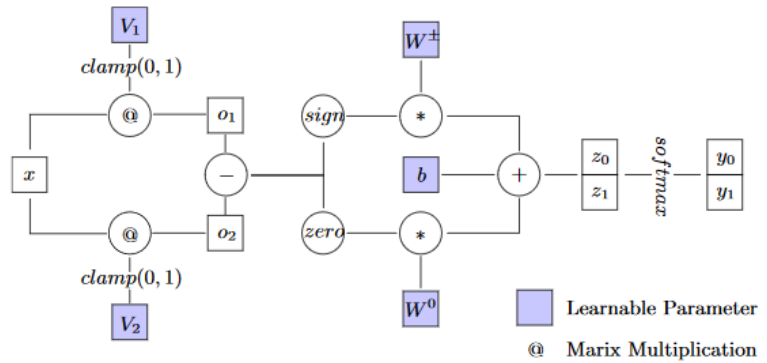
G-NALU (Rajaa and Sahoo, 2019) (No figure exists)

NAU (Madsen and Johansen, 2020) (No figure exists)

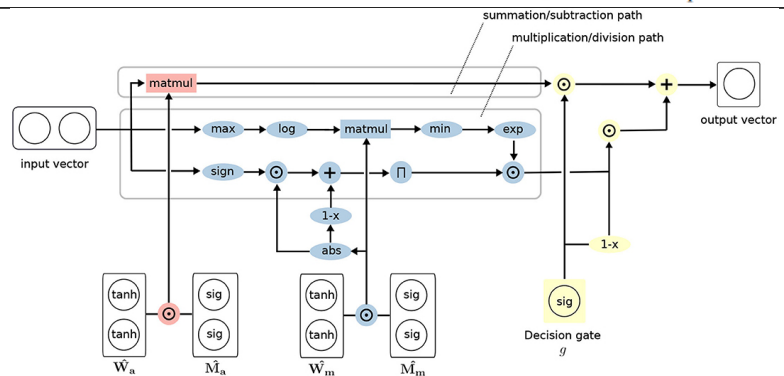
NMU (Madsen and Johansen, 2020)



NSR (Faber and Wattenhofer, 2020)

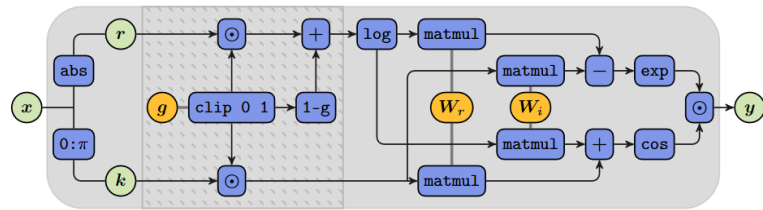


iNALU (Schlör et al., 2020)





NPU (Heim et al., 2020)



## C.2 Step-by-step Example using the NALU

To better understand how the internal process of a NALM, we provide a worked through example for subtraction using the NALU with parameters that can extrapolate.

**Task:** Subtract the second input value from the first where the input is  $\mathbf{x} = [2 \ 3 \ 4]$ . The output value should be  $[-1]$  (i.e.,  $2-3=-1$ ).

**Steps:**

1. Calculate  $\tanh(\widehat{\mathbf{W}})$ . The operation is  $+x_1 - x_2$  so  $\tanh(\widehat{\mathbf{W}}) = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$ .
2. Calculate  $\text{sigmoid}(\widehat{\mathbf{M}})$ . The first two input values are selected and the third is ignored, so  $\text{sigmoid}(\widehat{\mathbf{M}}) = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$ .
3. Calculate  $\tanh(\widehat{\mathbf{W}}) \odot \text{sigmoid}(\widehat{\mathbf{M}})$  to obtain  $\mathbf{W} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$ .
4. Calculate the result of the summative path.

$$\begin{aligned}
 \text{NAC}_+ &= \mathbf{x}\mathbf{W} \\
 &= [2 \ 3 \ 4] \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} \\
 &= [(2 \times 1) + (3 \times -1) + (4 \times 0)] \\
 &= [2 - 3 + 0] \\
 &= [-1].
 \end{aligned}$$

5. Calculate the result of the multiplicative path. (For simplicity, let us assume  $\epsilon = 0$ .)

$$\begin{aligned}
 \text{NAC}_\bullet &= \exp(\mathbf{W} \ln(|\mathbf{x}| + \epsilon)) \\
 &= \exp\left(\begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} \ln(|[2 \ 3 \ 4]|)\right) \\
 &= \exp\left(\begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} [\ln(2) \ \ln(3) \ \ln(4)]\right) \\
 &= \exp([\ln(2^1) + \ln(3^{-1}) + \ln(4^0)]) \\
 &= \exp(\ln([2^1 \times 3^{-1} \times 4^0])) \\
 &= \exp(\ln([2 \times \frac{1}{3} \times 1])) \\
 &= \exp(\ln([\frac{2}{3}])) \\
 &= [\frac{2}{3}].
 \end{aligned}$$

6. The target expression requires the summative path ( $\text{NAC}_+$ ) and ignores the multiplicative path ( $\text{NAC}_\bullet$ ), therefore gate is  $\text{sigmoid}(\mathbf{x}\mathbf{G}) = [1]$ .

7. Combine all the pieces to get the output.

$$\begin{aligned}\hat{\mathbf{y}} &= \mathbf{g} \odot \mathbf{a} + (\mathbf{1} - \mathbf{g}) \odot \mathbf{m} \\ &= [1] \odot [-1] + ([1] - [1]) \odot \left[\frac{2}{3}\right] \\ &= [-1] + [0] \\ &= [-1].\end{aligned}$$

### **C.3 Naive NPU Derivation**

Derivation 1 shows how the NAC $\bullet$  can be transformed into the NPU.

$$\text{NAC}_{\bullet} : y_o = \exp \left( \sum_{i=1}^I (W_{i,o} \cdot \ln(|x_i| + \epsilon)) \right)$$

Assume use of complex logarithm and separation of weights  $W_{i,o}$  to  $W^{\text{RE}}$  and  $W^{\text{IM}}$ .

$$= \exp \left( \sum_{i=1}^I ((W_{i,o}^{\text{RE}} + iW_{i,o}^{\text{IM}}) \cdot \ln(x_i)) \right)$$

Rewrite  $\ln(x_i)$  in polar form  $\ln(x_i) = \ln(r_i) + ik_i$  where  $r_i$  is the magnitude and  $k_i$  (Equation 2.19) is the phase.

$$= \exp \left( \sum_{i=1}^I ((W_{i,o}^{\text{RE}} + iW_{i,o}^{\text{IM}}) \cdot (\ln(r_i) + ik_i)) \right)$$

Expand the expression using FOIL (first, outer, inner and last).

$$= \exp \left( \sum_{i=1}^I (W_{i,o}^{\text{RE}} \ln(r_i) + iW_{i,o}^{\text{RE}} k_i + iW_{i,o}^{\text{IM}} \ln(r_i) + i^2 W_{i,o}^{\text{IM}} k_i) \right)$$

Collect real and imaginary terms.

$$= \exp \left( \sum_{i=1}^I (W_{i,o}^{\text{RE}} \ln(r_i) + i^2 W_{i,o}^{\text{IM}} k_i) \right) \cdot \exp \left( \sum_{i=1}^I (iW_{i,o}^{\text{IM}} \ln(r_i) + iW_{i,o}^{\text{RE}} k_i) \right)$$

Rewrite using  $i^2 = -1$

$$= \exp \left( \sum_{i=1}^I (W_{i,o}^{\text{RE}} \ln(r_i) - W_{i,o}^{\text{IM}} k_i) \right) \cdot \exp \left( \sum_{i=1}^I (i(W_{i,o}^{\text{IM}} \ln(r_i) + W_{i,o}^{\text{RE}} k_i)) \right)$$

Use Euler's formula  $e^{ix} = \cos x + i \sin x$

$$= \exp \left( \sum_{i=1}^I (W_{i,o}^{\text{RE}} \ln(r_i) - W_{i,o}^{\text{IM}} k_i) \right) \cdot \left( \cos \left( \sum_{i=1}^I (W_{i,o}^{\text{IM}} \ln(r_i) + W_{i,o}^{\text{RE}} k_i) \right) + i \sin \left( \sum_{i=1}^I (W_{i,o}^{\text{IM}} \ln(r_i) + W_{i,o}^{\text{RE}} k_i) \right) \right)$$

Remove the remaining imaginary terms as we only want real weights.

$$\exp \left( \sum_{i=1}^I (W_{i,o}^{\text{RE}} \ln(r_i) - W_{i,o}^{\text{IM}} k_i) \right) \cdot \left( \cos \left( \sum_{i=1}^I (W_{i,o}^{\text{IM}} \ln(r_i) + W_{i,o}^{\text{RE}} k_i) \right) + i \sin \left( \sum_{i=1}^I (W_{i,o}^{\text{IM}} \ln(r_i) + W_{i,o}^{\text{RE}} k_i) \right) \right)$$

The resulting expression is the NPU (Equation 2.17) assuming  $r_i$  is relevance gating (Equation 2.18).

$$\text{NPU} : y_o = \exp \left( \sum_{i=1}^I (W_{i,o}^{\text{RE}} \ln(r_i) - W_{i,o}^{\text{IM}} k_i) \right) \cdot \cos \left( \sum_{i=1}^I (W_{i,o}^{\text{IM}} \ln(r_i) + W_{i,o}^{\text{RE}} k_i) \right)$$

DERIVATION 1: Deriving the NPU from the NALU's  $\text{NAC}_{\bullet}$ .



## Appendix D

# NALM Benchmarking - Experiments and Cross Module Comparisons of Existing Works

This Appendix covers other types of experiments covered in previous NALM papers and cross module comparison prior to running our benchmark experiments.

### D.1 Additional Experiments

This section briefly summarises additional experiments given in the arithmetic NALM papers. We do not cross-compare papers for each experiment as there is too little similarity between experiments.

Trask et al. (2018) carries out a recurrent version of their static task experiment to test the  $NAC_+$  where the subsets  $a$  and  $b$  are accumulated over multiple timesteps. The purpose of this task is to generate much larger output values to test the NALU on. Furthermore, as well as pure arithmetic tasks, Trask et al. (2018) tests NALU in other settings such as: translating numbers in text form into the numerical form (for example ‘two hundred and one’ to ‘201’), a block grid-world which requires travelling from point A to B in exactly  $n$  timesteps, and program evaluation for programs with arithmetic and control operations. However, the NALU is not utilised for its capabilities as a NALM in the text-to-number task as the NALU is applied to a LSTM’s hidden state vector; therefore it is questionable if the arithmetic capabilities of NALU are being used, as the NALU may also have to decode the numerical values from the LSTM vector. The MNIST dataset is also used to evaluate NALU’s abilities in being part of end-to-end applications. This includes exploring counting the occurrence of different digits, the addition of a sequence of digits, and parity prediction.

Madsen and Johansen (2020) also uses MNIST for testing the module’s abilities to act as a recurrent module for adding/multiplying the digits. Madsen and Johansen (2020) additionally provide experiments to express the validity of their modules. This includes modifying the number of redundant hidden units, different input training ranges, ablation on multiplication, stress testing the stacked NAU-NMU against difference input sizes, overlap ratios and subset ratios, showing the failure of gating in convergence, and parameter tuning regularisation parameters.

Schlör et al. (2020) provide three additional experiments. Experiment 1 (‘Minimal Arithmetic Task’) uses a single layer to do a single operation with no redundancy to see the effect of different input distributions. Experiment 2 (‘Input Magnitude’) sees the effect of training data by controlling the magnitude of the interpolation data. The NALU is found to fail on magnitudes greater than 1. The iNALU remains unaffected for addition and subtraction. Multiplication performance is coupled to magnitude where extrapolation error increases with magnitude. Division is uncorrelated to the input magnitude. To increase problem difficulty, experiment 3 (‘Simple Arithmetic Task’) introduces redundancy where from 10 inputs only 2 are relevant. NALU improves performance for exponentially distributed data when redundant inputs are introduced. iNALU shows improvements for multiplication where the module is able to succeed on previously failed training ranges such as an exponential distribution with a scale parameter of 5 (i.e.  $\lambda=0.2$ ) but worsens for division.

Heim et al. (2020) highlights the relevance gate’s use via a toy experiment to select one of the two inputs. They show the relevance gate transforms regions away from the solution which contains no gradient information into regions with more instructive gradients (Heim et al., 2020, Figure 3). Additionally, they demonstrate an application of a stacked NAU-NPU module for equation discovery for an epidemiological model.

## D.2 Cross Module Comparison

Comparing the existing findings across modules, the NALU is no longer considered state-of-the-art for neural arithmetic operation learning. For each operation the best module is as follows - **addition or subtraction**: NAU, **multiplication**: NMU, **division**: NPU (or Real NPU if the task is trivial).

The iNALU generally outperforms the NALU at the cost of additional parameters and complexities to the model. The magnitude of iNALU’s improvement varies, as Schlör et al. (2020) claims vast improvements, while Heim et al. (2020) claim minor. For division both the iNALU and NALU performances remain comparable. Success on multiplication is dependent on the input training range. Heim et al. (2020) states the NMU outperforms the iNALU on multiplication (as expected), but also addition and



subtraction. The reason lies in the architecture used. The model is a stacked NAU-NMU meaning the addition/subtraction would be modelled by the NAU. Therefore, the NMU would only be required to act as a selector, selecting the output of the summation (that is, have a single weight at 1 and the rest at 0). Therefore, if two NMUs are stacked together we expect the failure in a pure addition/subtraction task as shown in Madsen and Johansen (2020, Appendix C.7). Heim et al. (2020)'s results show that the NPU outperforms the iNALU for multiplication and division. When stacked on top of a NAU, the NPU performs similarly to the NMU for addition and subtraction. The NPU is outperformed by the NMU for multiplication, however it is more consistent in convergence against different runs. For addition and subtraction, the NAU-NMU is the sparsest module (having the least number of non-zero weights). The NPU uses  $L_1$  regularisation for arithmetic tasks, encouraging sparsity over discretisation due to its ability to express fractional powers. However, the main influence causing sparsity in the NPU modules is from using the relevance gating. If this gating is removed (denoted by the NaiveNPU in the experiments), models are consistently less sparse for all operations (Heim et al., 2020, Figure 7).

## D.3 Experiments and Findings of Modules for Logic Tasks

This section summarises the experiments provided in two existing logic based NALMs—the NLRL and the NSR.

### D.3.1 NLRL

Preliminary results are given which test basic logic and arithmetic operations: AND, OR, NOT and XOR, multiplication, addition with division, identity and constant selection. Each model consists of a stacked NLRL. Different numbers of intermediary units per layer are tested and the authors conclude increasing the units improves performance until a saturation point (found to be 8) is reached. A multi-operation based task requires stacking layers of NLRL which introduces redundancy as the stacked output and input layers both use negation gating. Therefore, if stacking is required, it is suggested to remove cases with consecutive negation gating layers and only have a single layer. Using a module with both AND and OR results in faster convergence (fewer iterations), compared to using a module only using AND operations, but has a longer computation time to train each iteration.

### D.3.2 NSR

Faber and Wattenhofer (2020) first check if the NSR can learn comparison operations on both an integer and floating-point input setting. Results show that the NSR can learn the comparison functions with both input types and can extrapolate well. Modules struggle with learning the  $=$  and  $\neq$  operations, but performance can be improved by introducing redundancy through additional sets of weights during training (Faber and Wattenhofer, 2020, Section 4.4). The NSR can be attached to a NAU to learn piecewise functions. Findings suggest a simple continuous function (such as the absolute difference between two inputs) can be learnt with extrapolation capabilities, but a non-continuous function cannot. The NSR can be converted into a recurrent module to find the minimum of a list and to count the occurrence of a number in a sequence. The minimum task performs perfectly on all extrapolation settings however the counting task's performance reduces as sequence length increases. An additional task requires finding the shortest paths in a Graph Neural Network where the network should learn to imitate the Bellman-Ford algorithm. This is used to show that the recurrent NSR can learn to aggregate numbers to a minimum. When extrapolating to larger graphs, performance improved with larger edge weights. Finally, a MNIST digit comparison task was tested to see if the NSR can be used as a downstream module for a CNN in an end-to-end manner. Findings show that the NSR based network cannot outperform a vanilla CNN but is comparable to a MLP based network, where the underperformance was suggested to be a result of a weak learning signal.

## Appendix E

# Experiment Details

This appendix provides details regarding experiment setups used throughout the thesis including parameters, hardware and runtimes.

### E.1 Benchmark Synthetic Arithmetic Tasks

This section presents additional experiment details used for the two layer Arithmetic Dataset Task and single layer Single Module Arithmetic Task.

#### E.1.1 Experiment Parameters

The module specific hyperparameters for the NAU/NMU, (Real)NPU and iNALU can be found in Tables E.1, E.2 and E.3 respectively.

TABLE E.1: Additional parameters for the NMU (and NAU) for the Single Module and Arithmetic Dataset Task. The  $\hat{\lambda}$ ,  $\lambda_{start}$ ,  $\lambda_{end}$ .

Parameter	Arithmetic Dataset Task	Single Module Task
$\hat{\lambda}$	10	10
$\lambda_{start}$	1 million	20000
$\lambda_{end}$	2 million	35000
Learning rate	$10^{-3}$	$10^{-3}$

TABLE E.2: Parameters specific to the NPU and Real NPU modules for the Single Module Task.

Parameter	Value
$(\beta_{start}, \beta_{end})$	$(10^{-7}, 10^{-5})$
$\beta_{growth}$	10
$\beta_{step}$	10000
Learning rate	$5 \times 10^{-3}$

TABLE E.3: Parameters specific to the iNALU for the Single Module Task.

Parameter	Single Module Task
$\omega$	20
$t$	20
Gradient clip range	[-0.1,0.1]
Max stored losses (for reinitialisation check)	5000
Minimum number of epochs before regularisation starts	10000

### E.1.2 Hardware and Runtimes

All experiments for the Single Module Task and the Arithmetic Dataset Tasks were trained on the CPU, as training on GPUs takes considerably longer, using a 16 core/32 thread CPU server with 128 GB memory and 2.4 GHz processors. For any model, a single seed for a single training range can be completed within 5 minutes for the Single Module Task and within 4.5 hours for the Arithmetic Dataset Task. Timings are based on a single run rather than the runtime of the script execution because the queuing time from jobs when executing scripts is not relevant to the experiment timings.

## E.2 Multiplication MNIST Experiments

### E.2.1 Experiment Parameters

For further details, refer to the Sequential MNIST Product experiment details from [Madsen and Johansen \(2020\)](#).

### E.2.2 Hardware and Runtimes

All experiments for the MNIST based tasks were trained using a single GeForce GTX 1080 GPU. For the Static MNIST experiments, a single fold can be completed in approximately 5 hours for the Isolated Digit setup experiment and 10.5 hours for the Colour Channel Concatenated Digit setup. The Sequential MNIST experiments runtimes and memory usage are found in Table E.4.

TABLE E.4: Time taken and GPU memory required to run Sequential MNIST experiments. Experiments are run over 10 seeds.

Experiment	Model	Criterion	Device	Epochs	Approximate time for completing 1 seed/- fold (hh:mm:ss)	GPU memory (MiB)
Sequential MNIST Product	Reference				02:00:00	
	NMU	MSE	GPU	1000	02:55:00	679
	sNMU $\mathcal{U}[1,5]$				03:00:00	
	sNMU $\mathcal{U}[1,1+1/sd(x)]$				03:10:00	

### E.3 Division Experiments

Table E.5 shows the Real NPU parameters are taken from Heim et al. (2020, Section 4.1) which we confirm work empirically in Figure 5.4b.

TABLE E.5: Parameters specific to the Real NPU modules for the Single Module Tasks.

Parameter	Value
$(\beta_{start}, \beta_{end})$	$(10^{-9}, 10^{-7})$
$\beta_{growth}$	10
$\beta_{step}$	10000
$\hat{\lambda}$	1

#### E.3.1 Parameter Initialisation

We give the initialisations used on the different module parameters:

**Real NPU:** The real weight matrix uses Pytorch’s Xavier Uniform initialisation. The gate vector initialises all values to 0.5. This is the same initialisation used in Heim et al. (2020).

**NPU:** The imaginary weight matrix is initialised to 0. The rest of the parameters are initialised the same as the Real NPU. This is the same initialisation used in Heim et al. (2020).

**NRU:** The weight matrix uses a Xavier Uniform initialisation which can have a maximum range between -0.5 to 0.5 (depending on the network sizes). This is the same initialisation the Neural Addition Unit uses (Madsen and Johansen, 2020).

**NMRU:** The weight matrix uses a Uniform initialisation which can have a maximum range between 0.25 to 0.75 (depending on the network sizes). This is the same initialisation the Neural Multiplication unit uses (Madsen and Johansen, 2020).

#### E.3.2 Hardware and Runtimes

All synthetic arithmetic experiments were trained on the CPU, as training on GPUs takes considerably longer. All Real NPU experiments were run on Iridis 5 (the University of Southampton’s supercomputer), where a compute node has 40 CPUs with 192 GB of DDR4 memory which uses dual 2.0 GHz Intel Skylake processors. All NRU and NMRU experiments were run on a 16 core CPU server with 125 GB memory 1.2 GHz processors.

Table E.6 displays the time taken for each experiment to run a single seed for a single range. Timings are based on a single run rather than the runtime of a script execution because the queuing time from jobs when executing scripts is not relevant to the experiment timings. For a single model, a single experiment would have 225 runs (for 9 training ranges and 25 seeds).

TABLE E.6: Timings of experiments.

<b>Experiment</b>	<b>Model</b>	<b>Approximate time for completing 1 seed (mm:ss)</b>
No redundancy (size 2)	Real NPU	03:20
	NRU	02:00
	NMRU	03:00
With redundancy (size 10)	Real NPU	05:30
	NRU	05:00
	NMRU	05:15

### **E.3.3 Summary Table of the Ranges Used for the Single Layer Task**

Table E.7 shows the ranges used to generate the summary statistics. Note that even though the interpolation ranges are given (to make it easier to compare against the relevant Figures), it is the success rate on the extrapolation range which is used in the table summary.



TABLE E.7: The relevant ranges used to calculate the summary statistics in Table 5.6.

Redun- dancy?	Input type	Distribution	Figure	Interpolation Ranges
No	Mixed-signs	Uniform	5.8	All 5 ranges: $\mathcal{U}[-2,-0.1)$ & $\mathcal{U}[0.1,2)$ , $\mathcal{U}[-2,-1)$ & $\mathcal{U}[1,2)$ , $\mathcal{U}[-2,2)$ , $\mathcal{U}[0.1,2)$ & $\mathcal{U}[-2,-0.1)$ and $\mathcal{U}[1,2)$ & $\mathcal{U}[-2,-1)$
	Mixed-signs	Truncated Normal	5.10	All 3 TN ranges: TN(-1,3): [-5,10), TN(0,1):[-5,5) and TN(1,3):[-10,5)
	Negative	Uniform	5.6	Only pure negative ranges: $\mathcal{U}[-0.2,-0.1)$ , $\mathcal{U}[-1.2,-1.1)$ , $\mathcal{U}[-2,-1)$ and $\mathcal{U}[-20,-10)$
	Positive	Uniform	5.6	Only pure positive ranges: $\mathcal{U}[0.1,0.2)$ , $\mathcal{U}[1,2)$ , $\mathcal{U}[1.1,1.2)$ and $\mathcal{U}[10,20)$
	Large magnitude	Uniform	5.10	$\mathcal{U}[-100,100)$ and $\mathcal{U}[-50,50)$
	Large magnitude	Benford	5.10	B[10,100)
	Close to 0	Uniform	5.15	All 5 ranges: $\mathcal{U}[0,0.0001)$ , $\mathcal{U}[0,0.001)$ , $\mathcal{U}[0,0.01)$ , $\mathcal{U}[0,0.1)$ and $\mathcal{U}[0,1)$
	Close to 0	Truncated Normal	5.10	TN(0,1)[-5, 5)
Yes	Mixed-signs	Uniform	5.9	All 5 ranges: $\mathcal{U}[-2,-0.1)$ & $\mathcal{U}[0.1,2)$ , $\mathcal{U}[-2,-1)$ & $\mathcal{U}[1,2)$ , $\mathcal{U}[-2,2)$ , $\mathcal{U}[0.1,2)$ & $\mathcal{U}[-2,-0.1)$ and $\mathcal{U}[1,2)$ & $\mathcal{U}[-2,-1)$
	Mixed-signs	Truncated Normal	5.11	All 3 TN ranges: TN(-1,3): [-5,10), TN(0,1):[-5,5) and TN(1,3):[-10,5)
	Negative	Uniform	5.7	Only pure negative ranges: $\mathcal{U}[-0.2,-0.1)$ , $\mathcal{U}[-1.2,-1.1)$ , $\mathcal{U}[-2,-1)$ and $\mathcal{U}[-20,-10)$
	Positive	Uniform	5.7	Only pure positive ranges: $\mathcal{U}[0.1,0.2)$ , $\mathcal{U}[1,2)$ , $\mathcal{U}[1.1,1.2)$ and $\mathcal{U}[10,20)$
	Large magnitude	Uniform	5.11	$\mathcal{U}[-100,100)$ and $\mathcal{U}[-50,50)$
	Large magnitude	Benford	5.11	B[10,100)
	Close to 0	Truncated Normal	5.11	TN(0,1)[-5, 5)

## E.4 MNIST Product Tasks: Architecture Details

This section details the architectures used and further explores the learnt models from the MNIST tasks.

### E.4.1 Isolated Digits

The digit classification network can be found in Figure E.1. The application of the fc2 Linear layer will result in logits for each MNIST digit in the batch. At this point, the two MNIST digits are still separated and treated as independent batch items. A softargmax is applied to convert logits into a soft selection of the index for the most likely MNIST digit value. This is used to select a digit value and then the two digits are recombined into their original two-digit format. The output of the digit classifier (of shape [batch size, 2]) is passed to a multiplication module which returns the final output predictions.

```
DigitClassifier(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (dropout1): Dropout(p=0.25, inplace=False)
  (fc1): Linear(in_features=9216, out_features=128, bias=True)
  (dropout2): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)
```

(A)

Layer (type)	Output Shape	Param #
DigitClassifier		
Conv2d-1	[128, 32, 26, 26]	320
Conv2d-2	[128, 64, 24, 24]	18,496
Dropout-3	[128, 64, 12, 12]	0
Linear-4	[128, 128]	1,179,776
Dropout-5	[128, 128]	0
Linear-6	[128, 10]	1,290
Classifier-7	[128, 2]	0

(B)

FIGURE E.1: Digit classification network structure and summary used in the Isolated Digits MNIST task

### E.4.2 Colour Channel Concatenated Digits

We use the rotated, translated, and scaled (RTS) dataset described in Jaderberg et al. (2015, Appendix A.4). The RTS dataset is generated by randomly rotating an MNIST digit by +45 and -45 degrees, randomly scaling the digit by a factor of between 0.7 and 1.2, and placing the digit in a random location in a 42×42 image.

Given an image which is distorted via random scaling, rotation and translation, the Spatial Transformer Network can learn to locate the digit of interest and transform the source image to produce a version of the digit more like its non-distorted form. First, a localisation network learns a set of  $K$  control coordinates which are normalised between  $[-1,1]$ . These control points learn a grid around the point of interest which in this case is the digit. As there are two digits, two localisations are learnt. A localisation network consists of a convolutional network (see Figure E.2) with a tanh transformation at the end so the output of the network is between  $[-1,1]$ , making it bounded (Shi et al., 2016). The Thin Plate Spline (TPS) transformation parameters are calculated using the control points from a localisation network. The TPS transformation will transform the target image's pixel coordinates to the source image's pixel coordinates. To generate the TPS transformation matrix, we follow Shi et al. (2016, Section 3.1.2). Finally, a sampling grid will take the source image and its pixel locations to produce the transformed image. The transformed image is then passed to a classification network (see Figure E.3) to produce logits for digit classification. The classified digits are then passed to the relevant multiplication network.

```
LocalisationConvNet(
  AvgPool2d(kernel_size=2, stride=2, padding=0)
  Conv2d(2, 20, kernel_size=(5, 5), stride=(1, 1))
  ReLU(inplace=True)
  MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  Conv2d(20, 20, kernel_size=(5, 5), stride=(1, 1))
  ReLU(inplace=True)
  Linear(in_features=320, out_features=32, bias=True)
  Linear(in_features=32, out_features=32, bias=True)
)
```

(A)

Layer (type)	Output Shape	Param #
-----		
LocalisationConvNet		
AvgPool2d-1	[256, 2, 21, 21]	0
Conv2d-2	[256, 20, 17, 17]	1,020
ReLU-3	[256, 20, 17, 17]	0
MaxPool2d-4	[256, 20, 8, 8]	0
Conv2d-5	[256, 20, 4, 4]	10,020
ReLU-6	[256, 20, 4, 4]	0
Linear-7	[256, 32]	10,272
Linear-8	[256, 32]	1,056

(B)

FIGURE E.2: Localisation network structure and summary used in the Colour Channel Concatenated Digits MNIST task.

```

DigitClassifier(
  Conv2d(2, 32, kernel_size=(3, 3), stride=(1, 1))
  ReLU(inplace=True)
  Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  ReLU(inplace=True)
  MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  Dropout(p=0.25, inplace=False)
  Flatten()
  Linear(in_features=4096, out_features=128, bias=True)
  ReLU(inplace=True)
  Dropout(p=0.5, inplace=False)
  Linear(in_features=128, out_features=10, bias=True)
)

```

(A)

Layer (type)	Output Shape	Param #
DigitClassifier		
Conv2d	[256, 32, 19, 19]	608
ReLU	[256, 32, 19, 19]	0
Conv2d	[256, 64, 17, 17]	18,496
ReLU	[256, 64, 17, 17]	0
MaxPool2d	[256, 64, 8, 8]	0
Dropout	[256, 64, 8, 8]	0
Flatten	[256, 4096]	0
Linear	[256, 128]	524,416
ReLU	[256, 128]	0
Dropout	[256, 128]	0
Linear	[256, 10]	1,290

(B)

FIGURE E.3: Digit classification network structure and summary used in the Colour Channel Concatenated Digits MNIST task.

### E.4.3 Sequential MNIST

The network for the Sequential MNIST task can be found in Figure E.4. There are two parts to the network - the regression network to convert MNIST image digits to numbers and the NALM to do the calculation. Note that for the baseline the NALM would represent a fixed operation so would not consist of any learnable parameters.

```

SequentialMnistNetwork(
  (image2label): RegressionMnistNetwork(
    (conv1): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
    (conv2): Conv2d(20, 50, kernel_size=(5, 5), stride=(1, 1))
    (fc1): Linear(in_features=800, out_features=500, bias=True)
    (fc2): Linear(in_features=500, out_features=1, bias=True)
  )
  (recurrent_cell): GeneralizedCell(
    input_size=1, hidden_size=1, unit_name=NALM
    (cell): NALM(
      input_size=1, hidden_size=1
      (op): NALM(in_features=2, out_features=1)
    )
  )
)

```

FIGURE E.4: Network structure used in the Sequential MNIST task, assuming the regression network is a NALM.

## Appendix F

# Multiplication: Static MNIST Analysis

### F.1 Class Accuracies

This section plots the class accuracies of the models for a fold, evaluated on the test dataset. Doing so helps assess the learnt representations of the digit classifier network. The accuracy for classifying each digit over the test set is plotted with a further breakdown of the decisions over each digit using an unnormalised confusion matrix. The distribution of digit labels will be non-uniform.

### F.2 Isolated Digits

See Figures F.1 and F.2. The baseline is unable to classify zeros, mistaking all occurrences except 1 as the number 1. The FC model completely misclassifies digits 6, 7 and 8 as 7, 8 and 9 respectively. Both NMU and sNMU variants have strong classifiers with each digit getting at least 96% success in classification.

### F.3 Colour Channel Concatenated Digits.

See Figures F.3 and F.4. Results are shown for a fold which models found especially challenging. Only the baseline and the sNMU using batch statistics are able to learn classifiers which can provide a distinct diagonal over the confusion matrix. For this fold, the batch sNMU can outperform the baseline's classifier for every digit, implying

the learnable multiplication layer provides a better optimisation landscape. The remaining multiplication models have no sign of convergence, with the FC model learning to have a high bias towards classifying the digit 3.

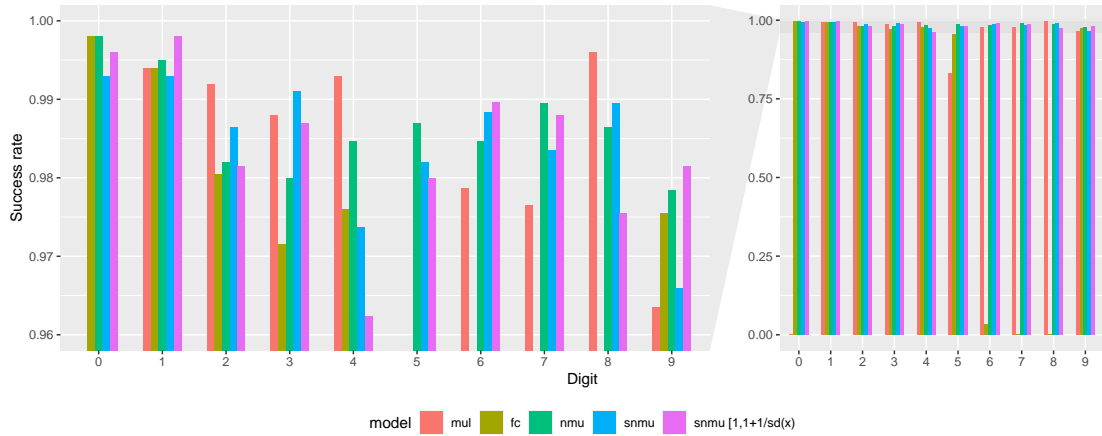


FIGURE F.1: Success rates for classifying each digit in the test dataset for a single seed. (Left) Zoom-in for success in range 0.95-1. (Right) The full plot from success rate 0-1.

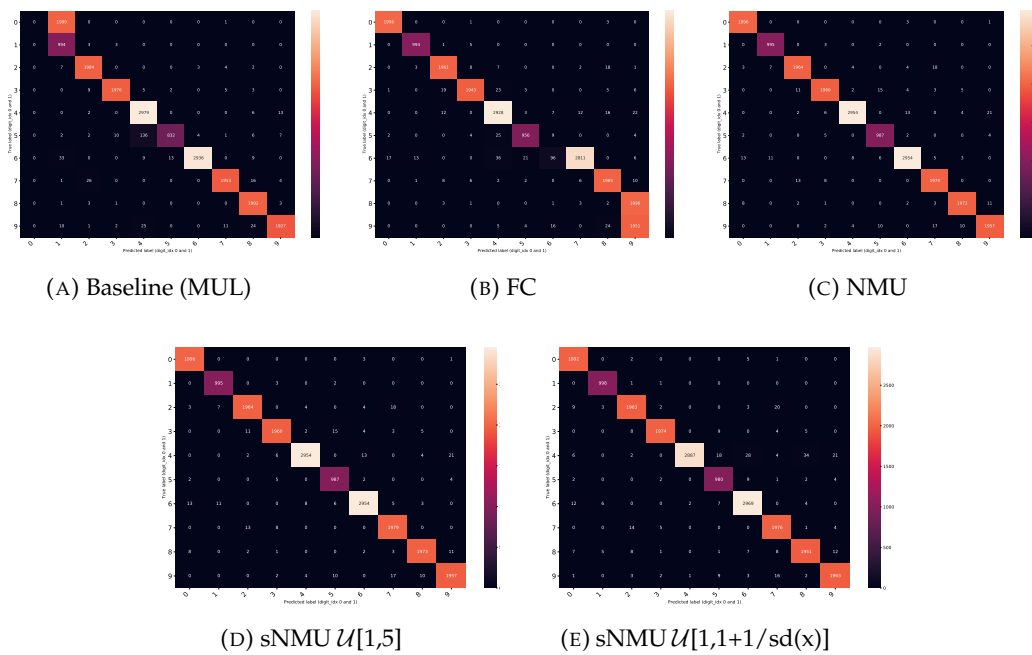


FIGURE F.2: Confusion matrices for intermediate label classification

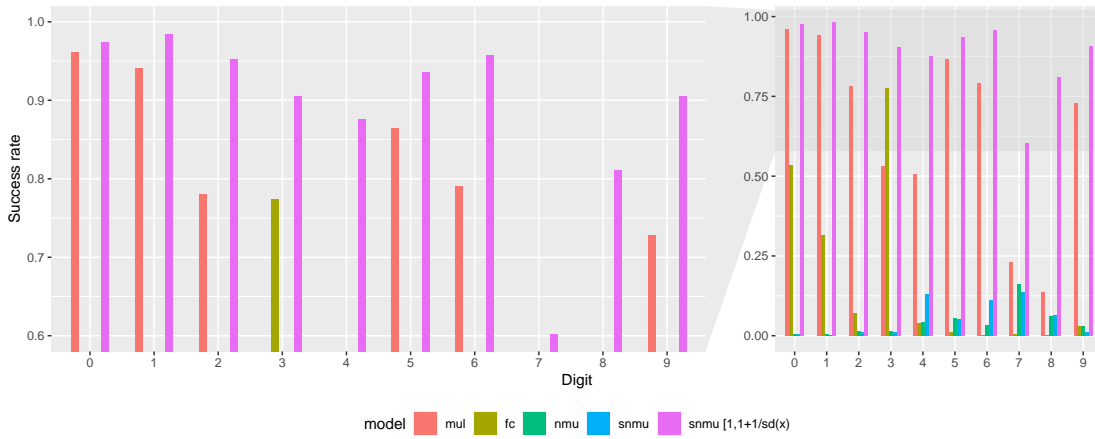


FIGURE F.3: Success rates for classifying each digit (with rounding) in the test dataset for a single seed. (Left) Zoom-in for success in range 0.95-1. (Right) The full plot from success rate 0-1.

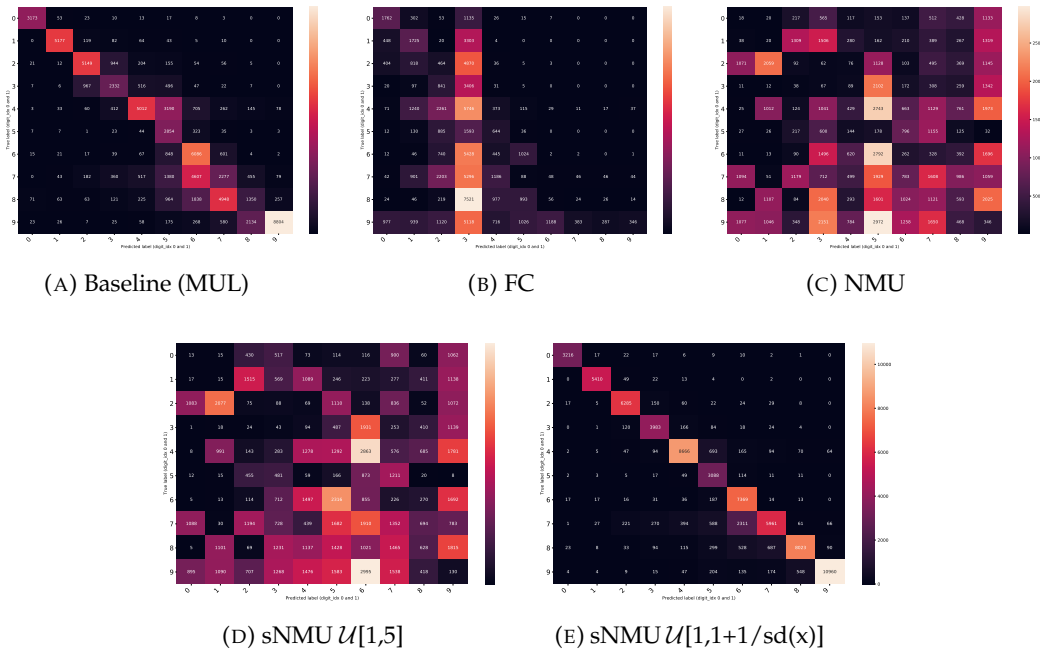


FIGURE F.4: Confusion matrices for intermediate label classification.

## F.4 Digit Classification Accuracy over Epochs

This section shows how accuracies for classifying each digit evolves over the epochs. The average values over all folds are shown (with 95% confidence intervals).

**Isolated digits.** Figure F.5 shows similar learning for both digits, with the sNMU modules providing tight confidence bounds. The sNMU with range  $\mathcal{U}[1,5]$  also shows better accuracy over the baseline after approximately 600 epochs.

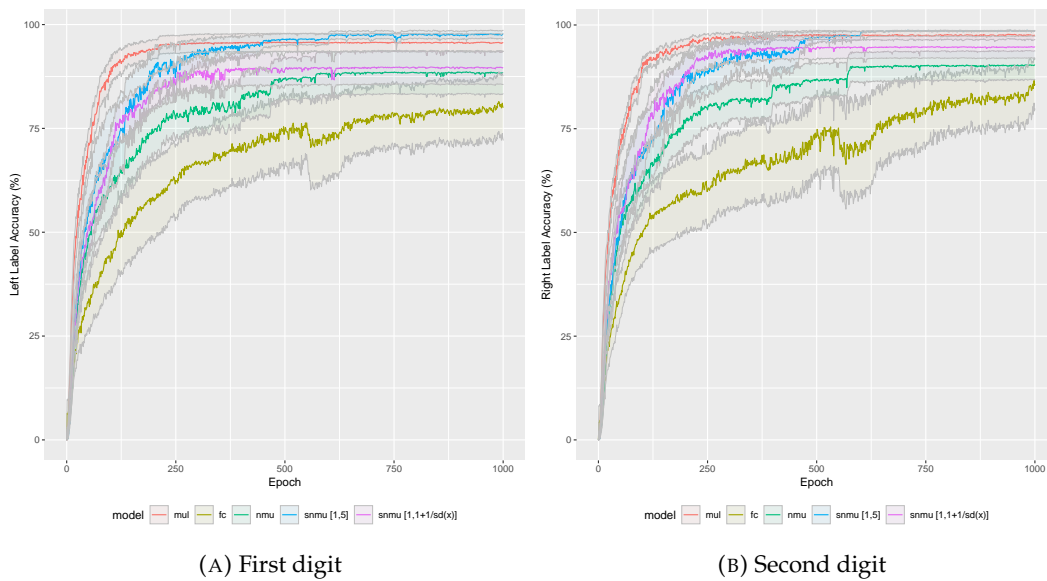


FIGURE F.5: Label accuracy vs epoch of the two digits for the Isolated digit variant of the Static MNIST Product task.

**Colour Channel Concatenated Digits.** Figure F.6 shows a greater variation in performance over the different models and folds in comparison to the isolated digits' results. The difference can be explained by the increased difficulty of this task, where localisation for each digit must be completed by the image classifier network. The solved baseline model shows challenges in robustness from the large confidence bounds while the batch sNMU provides much tighter bounds. The importance of having a reasonable noise range is also reflected in this task, with the sNMU using  $\mathcal{U}[1,5]$  noise unable to learn any reasonable image classifiers. It is also clear even with a bad noise interval, using stochasticity is better than not using stochasticity (i.e., sNMU vs NMU).

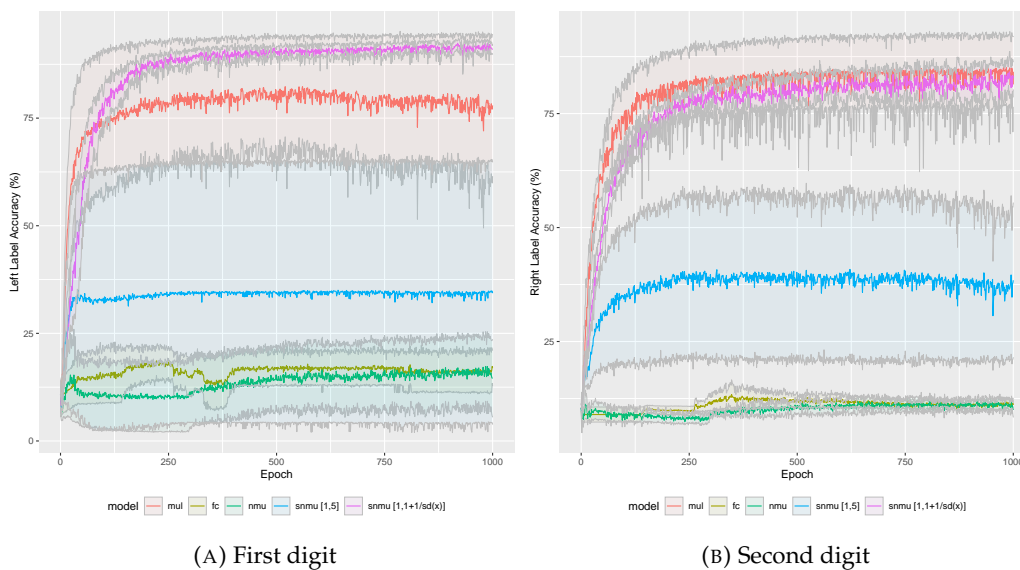


FIGURE F.6: Label accuracy vs epoch of the two digits for the Colour channel concatenated digit variant of the Static MNIST Product task.



## Appendix G

# Division: Additional Analysis

### G.1 Properties of a Division Module

When building a division module, the following properties should be included:

**Ability to multiply:** Without multiplication the module is limited to expressing reciprocals.

**Interpretable weights:** A good division module should produce generalisable solutions to OOD data. Using interpretable weights to represent exact operations is one way of doing so, e.g., -1 to divide, 1 to multiply, and 0 to not select. For the scope of this paper, we focus on discrete weights, however fractional weights can also be considered interpretable. For example, the Real NPU can express  $\frac{1}{\sqrt{x_i}}$  using a weight value of -0.5.

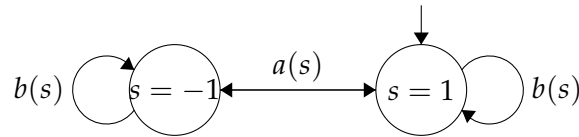
**Calculating the output:** This can be decomposed into three tasks: magnitude calculation, sign calculation and input selection.

**Magnitude calculation:** Refers to calculating the output value for a calculation. This is achieved using discrete weight parameters. For example, the Real NPU and NRU use a weight value of -1 for calculating reciprocals of the selected input and 1 for multiplication, while the NMRU uses 1 for selecting an input element resulting in either a multiplication or reciprocal depending on the weight's position index.

**Sign of the output:** Calculating the sign value (1/-1) of the output can occur at an element level in which the sign is calculated for each intermediary value as each input element is being processed, or at the higher input level in which the sign is calculated separately from the magnitude and then applied once the final output magnitude is calculated. The NRU uses the prior method while the Real NPU and NMRU use the latter method. If an input is 0 or considered irrelevant then the output sign will be 1.

(Ablation studies on the NMRU, Figure G.6, suggest the latter option which separately calculates the sign to be more beneficial).

The Real NPU and NMRU use the cosine function to calculate the final sign of the module's output neuron. Below shows the state diagram of how the sign value (i.e., the state) of the output would change depending on the inputs and relevant parameters being processed. We only consider the discrete parameters for simplicity. Both the Real NPU and NMRU use the same state diagram but have different conditions for a state transition to occur.



The conditions for the Real NPU transition functions  $a(s) = -s$  and  $b(s) = s$ , where  $s$  is the state value -1, or 1, are defined as follows:

$$a(s) : x_i < 0 \wedge w_{i,o} \in \{-1, 1\} \wedge g_i = 1 ,$$

$$b(s) : x_i \geq 0 \vee w_{i,o} = 0 \vee g_i = 0 .$$

Transitioning from one sign to another only occurs if the input element ( $x_i$ ) is negative and is considered relevant i.e., the gate ( $g_i$ ) and weight value ( $w_{i,o}$ ) is non-0. In contrast, to remain at a state requires either the input element to be  $\geq 0$  or not be considered relevant.

The conditions for the NMRU transition functions  $a(s) = -s$  and  $b(s) = s$ , where  $s$  is the state value -1, or 1, are defined as follows:

$$a(s) : x_i < 0 \wedge w_{i,o} = 1 ,$$

$$b(s) : x_i \geq 0 \vee w_{i,o} = 0 .$$

Transitioning from one sign to another only occurs if the input element ( $x_i$ ) is negative and is considered relevant i.e., the weight value ( $w_{i,o}$ ) is 1. To remain at a state requires either the input element to be  $\geq 0$  or the weight value to not select the input.

**Selection:** Not all inputs are relevant to the output value. To process any irrelevant input elements can be interpreted as converting to the identity value of multiplication/division (=1). The identity property means that any value multiplied/divided by the identity value remains at the original number. Hence, irrelevant inputs are converted into 1 (rather than being masked out to 0). For the multiplication case, this stops the output from becoming 0, and for division it avoids the divide by 0 case. For all the explored modules, a weight value of 0 will deal with the irrelevant input case. However, the Real NPU goes a step further by also having an additional gate vector with

the purpose of learning to select relevant inputs. Such gating has been proven to be helpful for an NPU based module (Heim et al., 2020), but may not be necessary when dealing with weights between  $[0,1]$  like in the NRMU (see Appendix G.6).

## G.2 NRU; Single Module Task (without Redundancy): Tanh Scale Factor

Figure G.1 shows the impact of changing the tanh scale factor. We find larger scale factors work better with a factor of 1000 being the best. This correlates to the findings in Faber and Wattenhofer (2020, Figure 5).

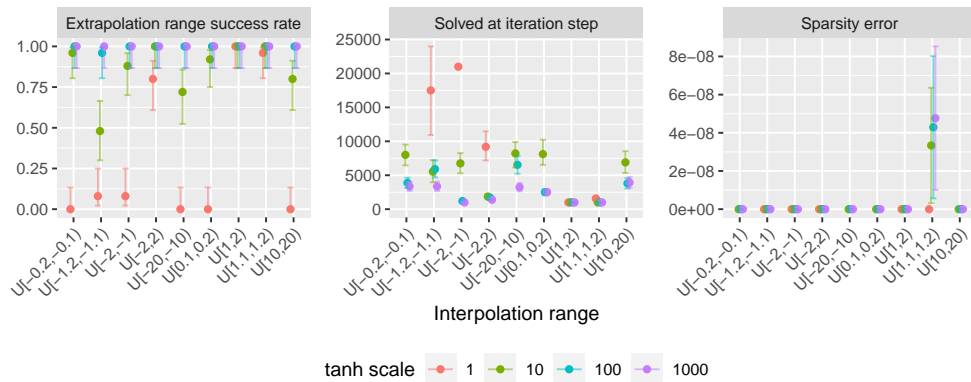


FIGURE G.1: Effect of the tanh scale factor for the NRU on the 2-input setting.

### G.3 Real NPU; Single Module Task (without Redundancy): Additional Experiments

Figure G.2 shows the results of using the NPU for the 2-input task. Of the 9 tested ranges,  $L_2$  has a lower success rate than  $L_1$  for 5 ranges and has the same success rate for the remaining 4 ranges. If  $L_2$  regularisation is used instead of no regularisation, it performs worse in 3 (of the 9) ranges, better in 3 ranges and the same in the remaining 3 ranges.

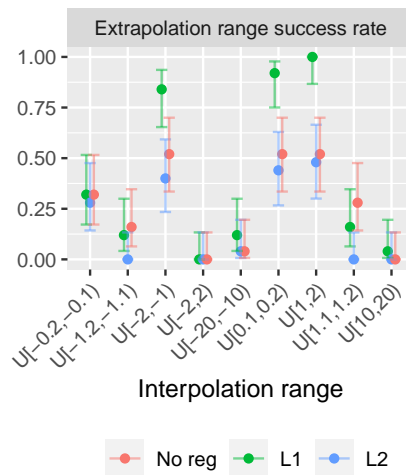


FIGURE G.2: Applying no regularisation,  $L_1$  regularisation and  $L_2$  regularisation on the Real NPU for the 2 input tasks.

Figure G.3 displays the effect of different learning rates for the modified Real NPU. A learning rate of  $5 \times 10^{-3}$  has the best performance over all ranges.

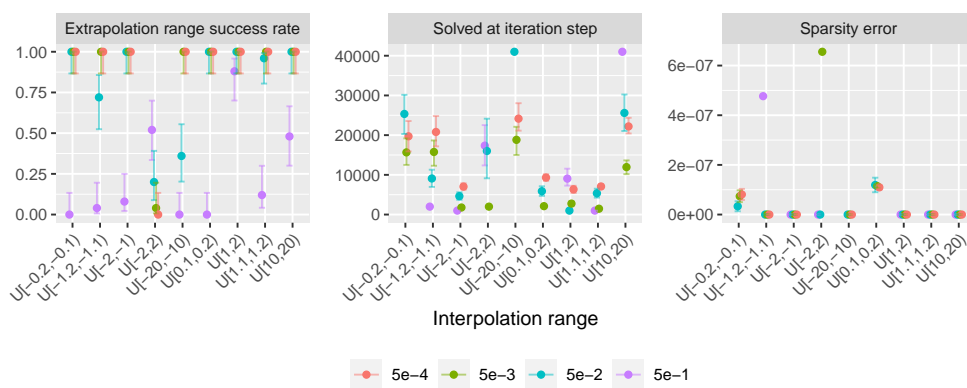


FIGURE G.3: Different learning rates on the Real NPU (mod) for the Single Module Task (no redundancy).

## G.4 NRU; the Single Module Task (without Redundancy): Effect of Learning Rate

Figure G.4 displays the effect of different learning rates for the NRU. A learning rate of 1 gets full success on all ranges with performance deteriorating as the learning rate reduces.

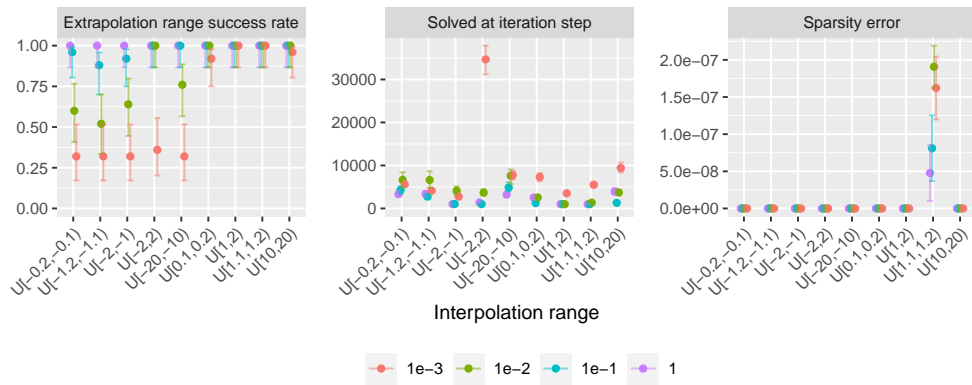


FIGURE G.4: Different learning rates on the NRU for the Single Module Task (no redundancy)

## G.5 Real NPU; Single Module Task (with Redundancy): Additional Experiments

We test the NPU module with all the modifications used on the real weight matrix. Also, assuming the global solution only uses the real weights, we enforce the complex weights to be clipped between  $[-1,1]$  and to go to 0 during the regularisation stage using a  $L_1$  penalty. Figure G.5 shows the complex weights without any constraints, hindering success and convergence speeds of negative ranges. Applying clipping and regularisation constraints does not result in any significant improvements against the Real NPU results.

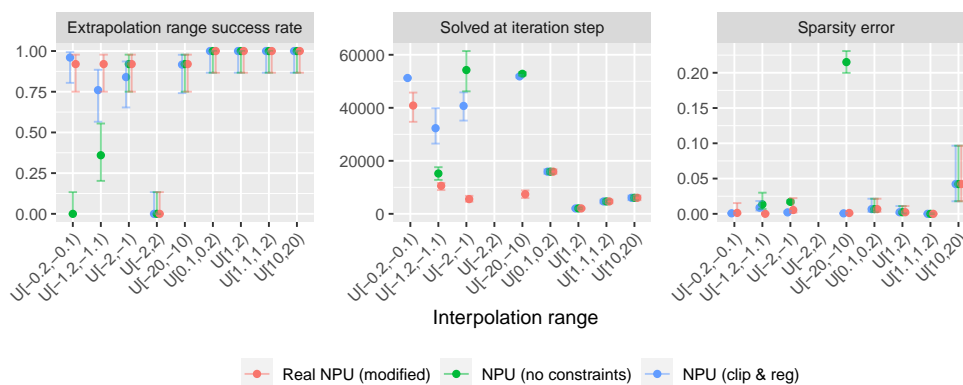


FIGURE G.5: Adapting the Real NPU to use complex weights (NPU) on the Single Module Task with redundancy. Compares the NPU architecture with the Real NPU modifications (i.e. NPU (no constraints)) and the same model but with the imaginary weights clipped to  $[-1,1]$  and  $L_1$  sparsity regularisation on the complex weights (i.e. NPU (clip & reg)).

## G.6 NMRU; Single Module Task (with Redundancy): Additional Experiments

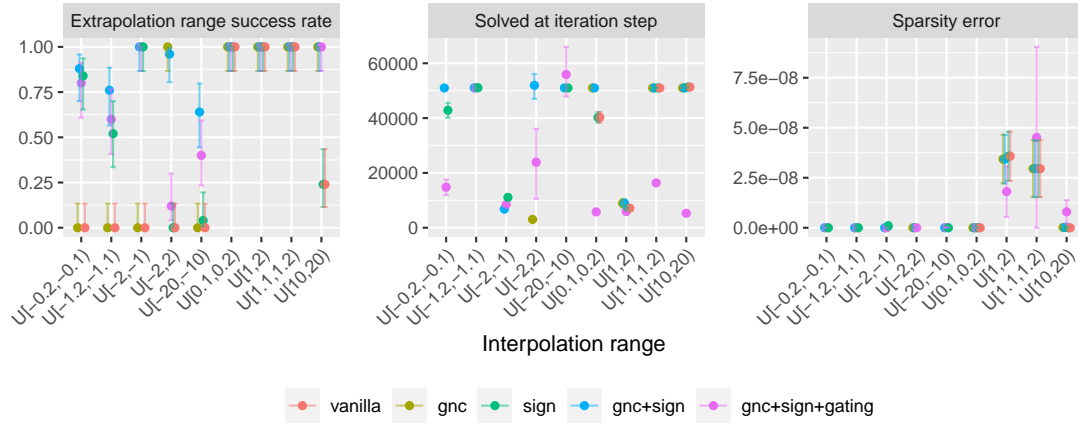


FIGURE G.6: Ablation study for the NMRU. gnc = Gradient norm clipping; sign = sign retrieval mechanism; gating = NPU-style gating.

This section further explores the NMRU architecture.

Figure G.6 shows an ablation study on different components of the NMRU architecture. Removing both the sign retrieval and grad norm clipping performs poorly over a majority of ranges (including positive ranges). Gradient norm clipping alone is unable to solve the issue of learning negative ranges, however, fully succeeds on the  $\mathcal{U}[-2,2]$  range. Using the sign retrieval without the gradient clipping gains successes for the negative ranges, though performance on  $\mathcal{U}[2,-2]$  is affected. However, including both gradient clipping and sign retrieval results in separating the calculation of the magnitude of the output and its sign while having reasonable gradients, gaining the most improvement over the vanilla NMRU. Further including a learnable gate vector (like the Real NPU), which is applied to the input vector, hinders performance. The largest solved at iteration step seems to be bounded at approximately 50,000 iterations which correlates to the point at which the sparsity regularisation begins, which highlights the importance of discretisation. Even with the different ablations, the sparsity errors of the successful seeds remain extremely low.

Figure G.7 shows the effect of using different learning rates on the NMRU (with grad norm clipping and sign retrieval) using an Adam optimiser. Too low a learning rate struggles on the mixed-sign range  $\mathcal{U}[-2,2]$ . Too high a learning leads to no success on multiple ranges.

Figure G.8 compares training the NMRU with either an Adam and SGD optimiser. As expected, Adam outperforms SGD in all ranges (except two, where both perform equally). This difference in performance can be accounted for by Adam's ability to



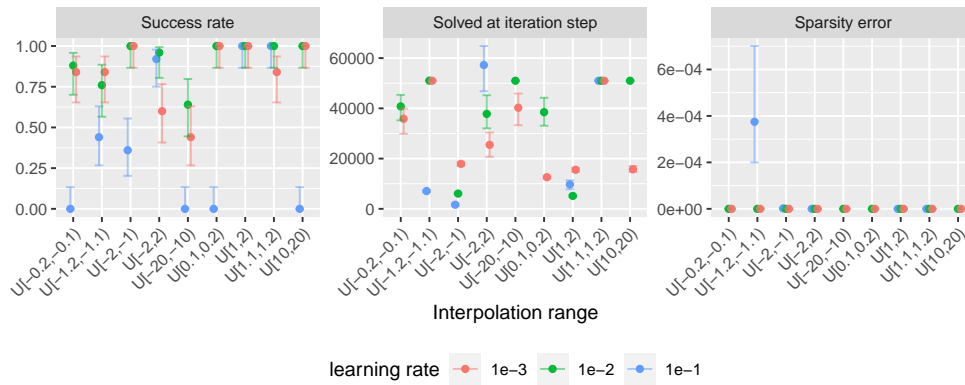


FIGURE G.7: Effect of different learning rates on the NMRU

scale the step size of each weight, which can complement the clipped gradient norm of the NMRU, in contrast to the SGD’s global step size.

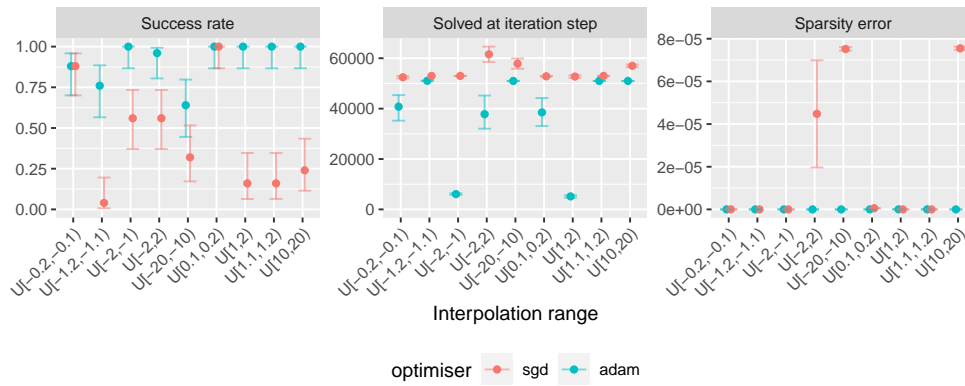


FIGURE G.8: Effect of optimiser on the NMRU. SGD = Stochastic Gradient Descent.

## G.7 NRU; Single Module Task (with Redundancy): Calculating the Sign Separately

The ‘separate NRU’ module calculates the magnitude and sign separately and then combines them using multiplication together once all input elements are accounted for. The following definition is used to calculate a NRU with separate magnitude and sign calculation,

$$z_o = \prod_{i=1}^I \left( |x_i|^{W_{i,o}} \cdot |W_{i,o}| + 1 - |W_{i,o}| \right) \cdot \prod_{i=1}^I \text{sign}(x_i)^{\text{round}(W_{i,o})}. \quad (\text{G.1})$$

Figure G.9 shows results, where the separate sign method shows no difference in success to the original NRU architecture.

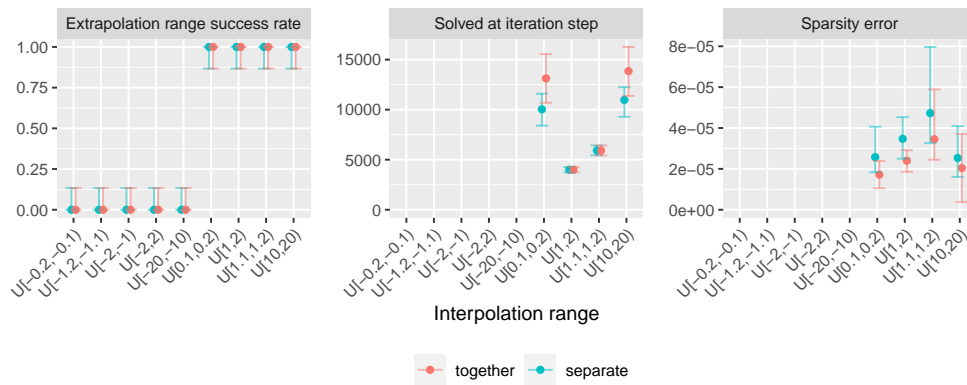


FIGURE G.9: NRU on the redundancy experiment comparing a module which calculates the magnitude and sign together vs calculating the magnitude and sign separately and then combining them.

## G.8 Division MNIST Arithmetic Task: Effect of Gradient Norm Clip

This section sees the effect of applying gradient norm clip (GNC) to the division NALMs for the two-digit MNIST task. Results (see Table G.1 and Figure G.10) indicate that using GNC is detrimental for the NRU but is advantageous for the Real NPU and NMRU.

TABLE G.1: Test accuracies of the output label for the MNIST task. The predictions and targets are rounded to 5 d.p. before the accuracy is calculated. The mean accuracy over 10-folds is given with the standard error. ‘gnc’ stands for grand norm clip.

	DIV	Real NPU (mod.)	NRU	NMRU
No gnc	<b>97.497±0.183</b>	97.147±0.242	97.517±0.291	44.69±13.841
With gnc=1	-	97.982±0.092	94.215±3.627	46.891±13.969

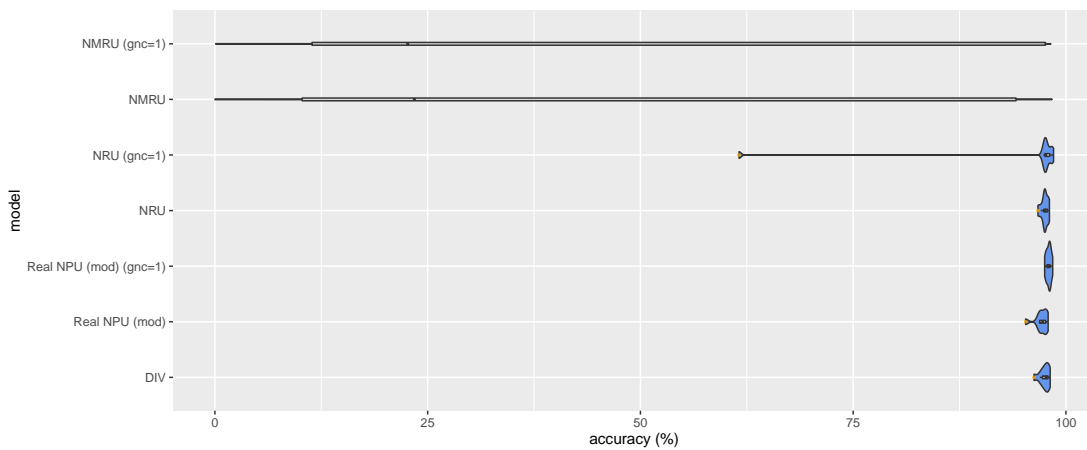


FIGURE G.10: Test accuracies on the 5 d.p. output values comparing the effect of clipping the gradient norm on NALMs.



## Appendix H

# Gradients of the Arithmetic Dataset Task

This section derives the generalised partial derivatives for the stacked NAU-NMU and NAU-sNMU. To keep derivations as simple as possible, we assume formulations of models without the use of regularisation or clipping.

Throughout this section, we assume the following notations:

- Superscript A ( $W^A$ ) = Weight matrix of a summative module (i.e. NAU)
- Superscript M ( $W^M$ ) = Weight matrix of a multiplicative module (i.e. NMU or sNMU)
- Weight matrix indexing  $W_{r,c}$  where  $r$  = row index and  $c$  = column index (starting at 1)
- $I$  = total number of input elements for the respective module
- $O$  = output size of the NAU weight matrix (or number of elements in the intermediate vector)
- $l$  = index for an output element
- $i$  = index for an input element

### H.1 MSE Loss for the Arithmetic Dataset Task

We define the MSE loss specific to the two-layer task below.  $N$  is the number of batch items.  $X_n$  is the input vector for batch item  $n$  with target scalar  $y_n$  and predicted scalar

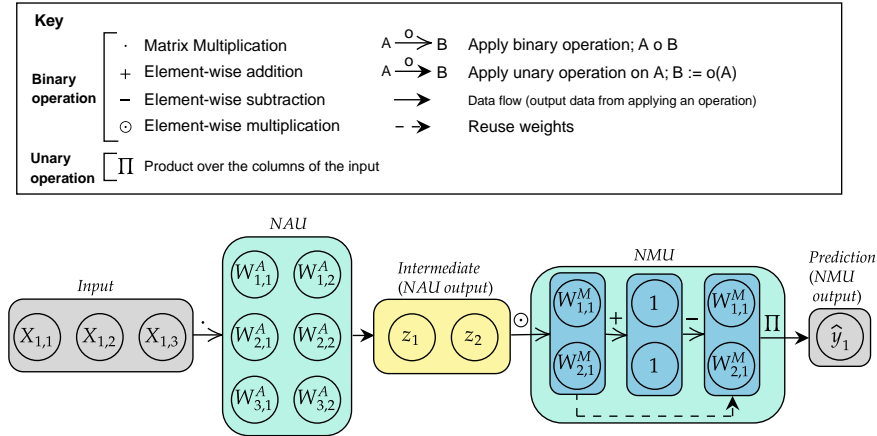


FIGURE H.1: Illustration of the data flow of a NAU-NMU module. The annotation of weights will be consistent with the gradient calculations.

$\hat{y}_n$ .  $I$  is the number of input elements ( $=100$ ) in the input vector.

$$L = \frac{1}{N} \sum_n (y_n - \hat{y}_n)^2$$

$$L = \frac{1}{N} \sum_n (y_n - \text{NMU}(\text{NAU}(\mathbf{X}_n)))^2$$

$$z_1 = \sum_i^I (X_{n,i} \cdot W_{i,1}^A)$$

$$z_2 = \sum_i^I (X_{n,i} \cdot W_{i,2}^A)$$

$$L = \frac{1}{N} \sum_n [y_n - ((1 + W_{1,1}^M \cdot z_1 - W_{1,1}^M)(1 + W_{2,1}^M \cdot z_2 - W_{2,1}^M))]^2$$

## H.2 Explicit Gradients

To help improve familiarity with notation, we first work through an example using predefined network sizes. For the following (simplified) two module example using the baseline stacked NAU-NMU, we assume an input vector size 3, intermediate size 2, and output size 1. As a further simplification, we only consider the loss for a single data-label pair  $(X_1, y_1)$ . We annotate the weights and components explicitly in Figure H.1. Matrix indexing follows the standard (row, column) convention, with indexing starting from 1.

### H.2.1 MSE Loss Partial Derivatives:

Derivation 2 calculates the loss derivative with respect to (wrt) three different weights values:  $W_{1,1}^A$ ,  $W_{1,2}^A$ , and  $W_{1,1}^M$  indicated by colours (yellow, purple, and teal). Colours red and blue are used to identify the parts of the derivative which are derived from the original equation (i.e., the predicted value  $\hat{y}_1$ ). Using the chain rule on the loss requires calculating the partial derivative of the predicted value wrt the weight, which is calculated via the product rule using the underlined red and blue terms as the parts. To differentiate each part further also requires another application of the product rule.

$$\begin{aligned}
 L &= (y_1 - \hat{y}_1)^2 \\
 \hat{y}_1 &= \frac{(W_{1,1}^M \cdot (X_{1,1} \cdot W_{1,1}^A + X_{1,2} \cdot W_{2,1}^A + X_{1,3} \cdot W_{3,1}^A) + 1 - W_{1,1}^M) \cdot (W_{2,1}^M \cdot (X_{1,1} \cdot W_{1,2}^A + X_{1,2} \cdot W_{2,2}^A + X_{1,3} \cdot W_{3,2}^A) + 1 - W_{2,1}^M)}{1} \\
 \frac{\partial L}{\partial W_{1,1}^A} &= -2 \cdot \frac{\partial \hat{y}_1}{\partial W_{1,1}^A} \cdot (y_1 - \hat{y}_1) \\
 \frac{\partial \hat{y}_1}{\partial W_{1,1}^A} &= W_{1,1}^M \cdot X_{1,1} \cdot (W_{2,1}^M \cdot (X_{1,1} \cdot W_{1,2}^A + X_{1,2} \cdot W_{2,2}^A + X_{1,3} \cdot W_{3,2}^A) + 1 - W_{2,1}^M) \\
 &= W_{1,1}^M \cdot X_{1,1} \cdot (W_{2,1}^M \cdot (\sum_i^I X_{1,i} \cdot W_{i,2}^A) + 1 - W_{2,1}^M) \\
 \frac{\partial L}{\partial W_{1,2}^A} &= -2 \cdot \frac{\partial \hat{y}_1}{\partial W_{1,2}^A} \cdot (y_1 - \hat{y}_1) \\
 \frac{\partial \hat{y}_1}{\partial W_{1,2}^A} &= (W_{1,1}^M \cdot (X_{1,1} \cdot W_{1,1}^A + X_{1,2} \cdot W_{2,1}^A + X_{1,3} \cdot W_{3,1}^A) + 1 - W_{1,1}^M) \cdot W_{2,1}^M \cdot X_{1,1} \\
 &= (W_{1,1}^M \cdot (\sum_i^I X_{1,i} \cdot W_{i,1}^A) + 1 - W_{1,1}^M) \cdot W_{2,1}^M \cdot X_{1,1} \\
 \frac{\partial L}{\partial W_{1,1}^M} &= -2 \cdot \frac{\partial \hat{y}_1}{\partial W_{1,1}^M} \cdot (y_1 - \hat{y}_1) \\
 \frac{\partial \hat{y}_1}{\partial W_{1,1}^M} &= ((X_{1,1} \cdot W_{1,1}^A + X_{1,2} \cdot W_{2,1}^A + X_{1,3} \cdot W_{3,1}^A) - 1) \cdot (W_{2,1}^M \cdot (X_{1,1} \cdot W_{1,2}^A + X_{1,2} \cdot W_{2,2}^A + X_{1,3} \cdot W_{3,2}^A) + 1 - W_{2,1}^M) \\
 &= (\sum_i^I (X_{1,i} \cdot W_{i,1}^A) - 1) \cdot (W_{2,1}^M \cdot (\sum_i^I (X_{1,i} \cdot W_{i,2}^A) + 1 - W_{2,1}^M)
 \end{aligned}$$

DERIVATION 2: Partial derivatives on the MSE Loss of the NAU-NMU wrt weight elements  $W_{1,1}^A$ ,  $W_{1,2}^A$ , and  $W_{1,1}^M$ .

**The partial derivative of the prediction wrt to a NAU weight is the product of two terms:** One term is the NMU weight (whose row index matches the column index of the target NAU weight) multiplied with the input (whose column index corresponds to the target NAU weight's row index). The other term is the result of what would be the output of the NMU if it is only applied to intermediate  $z_i$  where  $i$  is the value which is not the value of the column of the target NAU weight. E.g.  $W_{1,2}^A$  considers  $z_1$ .

The partial derivative of the prediction wrt to a NMU weight is the product of two terms: One term is the intermediate element which corresponds to the row value of the target NMU weight minus 1, e.g.  $W_{1,1}^M$  would have  $z_1 - 1$ . The other term is the result of what would be the output of the NMU if only applied to the intermediate  $z_i$  where  $i$  is the value which is not the value of the column of the target NAU weight. E.g.  $W_{1,1}^M$  considers  $z_2$ . This term also occurs in the partial derivative when the target weight being derived to is a NAU weight.

### H.3 Generalised NAU and NMU Partial Derivatives of the loss for a NAU-NMU

The derivative of the loss wrt either a NAU or NMU weight can then be derived using the chain rule. We formulate these gradients for the generalised case. The expression is generalised such that it can be applied to any element in the NAU weight matrix regardless of the matrix's size, and the NMU weight matrix regardless of the matrix's row size. Like before, we assume derivatives for a single data-label pair  $(X_1, y_1)$ .

To reiterate, the NAU weight matrix is denoted as  $W_{i,l}^A$  where the A represents a summative module (for adding/subtracting),  $l$  is the output element index for the output applying the NAU, and  $i$  is the index to select an element from the input.

$$\frac{\partial L}{\partial W_{i,l}^A} = -2(y_1 - \hat{y}_1) \cdot W_{l,1}^M X_{1,i} \cdot \prod_j^{\{O \setminus l\}} (W_{j,1}^M (\sum_{k=1}^I X_{1,k} W_{k,j}^A) + 1 - W_{j,1}^M)$$

$$\frac{\partial L}{\partial W_{l,1}^M} = -2(y_1 - \hat{y}_1) \cdot (\sum_{i=1}^I X_{1,i} W_{i,l}^A - 1) \cdot \prod_j^{\{O \setminus l\}} (W_{j,1}^M (\sum_{k=1}^I X_{1,k} W_{k,j}^A) + 1 - W_{j,1}^M)$$

$\{O \setminus l\}$  represents the indices of all output elements from applying the module excluding the index corresponding to the output for the weight element you are calculating the partial derivative of.

### H.4 Generalised NAU and NMU Partial Derivatives for a NAU-sNMU

We derive the generalised gradients as before but now using a sNMU rather than a NMU. Gradients are derived using the quotient rule. Let  $N$  be the noise matrix (same shape as input  $X$ ).



### H.4.1 MSE Loss Definition

$$\begin{aligned} L &= (y_1 - \hat{y}_1)^2 \\ &= (y_1 - \text{sNMU}(\text{NAU}(\mathbf{X}_1)))^2 \end{aligned}$$

### H.4.2 Loss derivatives wrt NAU and sNMU weights

Let

$$A = \prod_j^{\{O\}} N_{1,j} W_{j,1}^M \cdot \left( \sum_{k=1}^I X_{1,k} W_{k,j}^A \right) + 1 - W_{j,1}^M$$

be the result of the sNMU applied only to the output values of the NAU whose index is not the value of the column of the target NAU weight. Let

$$D = \prod_i^O N_{1,i} W_{i,1}^M + 1 - W_{i,1}^M,$$

be the denoising term. Therefore,

$$\begin{aligned} \frac{\partial L}{\partial W_{i,l}^A} &= -2(y_1 - \hat{y}_1) \cdot \frac{A}{D} \cdot W_{i,1}^M N_{1,i} X_{1,i} \\ \frac{\partial L}{\partial W_{l,1}^M} &= -2(y_1 - \hat{y}_1) \cdot \frac{A}{D^2} \cdot [D(N_{1,l} z_l - 1) \\ &\quad - (W_{l,1}^M N_{1,l} z_l + 1 - W_{l,1}^M)(N_{1,l} - 1) \left( \prod_j^{\{O\}} (N_{1,j} W_{j,1}^M + 1 - W_{j,1}^M) \right)] \end{aligned}$$

Although the residual term  $(y_1 - \hat{y}_1)$  remains, during training, the NAU-sNMU's gradients of the weights can take a different trajectory to the NAU-NMU. Hence, the use of noise in the sNMU may aid in alleviating the local minima issues of the NMU.



## References

- Samira Abnar, Mostafa Dehghani, and Willem H. Zuidema. Transferring inductive biases through knowledge distillation. *CoRR*, 2020. doi:10.48550/ARXIV.2006.00555.
- Guozhong An. The Effects of Adding Noise During Backpropagation Training on a Generalization Performance. *Neural Computation*, 8(3):643–674, 04 1996. ISSN 0899-7667. doi:10.1162/neco.1996.8.3.643.
- James A Anderson, Kathryn T Spoehr, and David J Bennett. A study in numerical perversity: Teaching arithmetic to a neural network. In *Neural networks for knowledge representation and inference*, pages 311–335. Lawrence Erlbaum Associates, Inc, 1994. ISBN 0-8058-1158-3 (Hardcover); 0-8058-1159-1 (Paperback).
- Cem Anil, Yuhuai Wu, Anders Johan Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Venkatesh Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. doi:10.48550/ARXIV.2207.04901.
- Giovanni Anobile, Guido Marco Cicchini, and David C Burr. Number as a primary perceptual attribute: A review. *Perception*, 45(1-2):5–31, 2016. doi:10.1177/0301006615602599.
- Mark H Ashcraft. Cognitive arithmetic: A review of data and theory. *Cognition*, 44(1-2):75–106, 1992. doi:10.1016/0010-0277(92)90051-I.
- Bernard J Baars. *A cognitive theory of consciousness*. Cambridge University Press, 1993.
- Bernard J Baars. In the theatre of consciousness. global workspace theory, a rigorous scientific theory of consciousness. *Journal of consciousness Studies*, 4(4):292–309, 1997.
- Dzmitry Bahdanau, Shikhar Murty, Michael Noukhovitch, Thien Huu Nguyen, Harm de Vries, and Aaron Courville. Systematic generalization: What is required and can it be learned? In *International Conference on Learning Representations (ICLR)*, 2019. doi:10.48550/ARXIV.1811.12889.

- Aron K. Barbey, Michael Koenigs, and Jordan Grafman. Dorsolateral prefrontal contributions to human working memory. *Cortex*, 49(5):1195–1205, 2013. ISSN 0010-9452. doi:10.1016/j.cortex.2012.05.022.
- Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, 2018. doi:10.48550/ARXIV.1806.01261.
- Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, 2013. doi:10.48550/ARXIV.1308.3432.
- Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales. In *International Conference on Machine Learning (ICML)*, pages 936–945. PMLR, 2021. URL <https://proceedings.mlr.press/v139/biggio21a.html>.
- Marcus Bloice, Peter M. Roth, and Andreas Holzinger. Performing arithmetic using a neural network trained on images of digit permutation pairs. *Journal of Intelligent Information Systems*, 08 2021. doi:10.1007/s10844-021-00662-9.
- Fred L. Bookstein. Principal warps: Thin-plate splines and the decomposition of deformations. *IEEE Transactions on pattern analysis and machine intelligence*, 11(6):567–585, 1989. doi:10.1109/34.24792.
- Matthew M Botvinick, Todd S Braver, Deanna M Barch, Cameron S Carter, and Jonathan D Cohen. Conflict monitoring and cognitive control. *Psychological review*, 108(3):624, 2001. doi:10.1037/0033-295X.108.3.624.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1877–1901, 2020. URL <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf>.

- Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016. doi:10.1073/pnas.1517384113.
- Sergio A. Cannas. Arithmetic Perceptrons. *Neural Computation*, 7(1):173–181, 01 1995. ISSN 0899-7667. doi:10.1162/neco.1995.7.1.173.
- William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabricio Olivetti de Franca, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H. Moore. Contemporary symbolic regression methods and their relative performance. In *35th Conference on NeurIPS Track on Datasets and Benchmarks*, 2021. URL <https://openreview.net/forum?id=xVQMrDLyGst>.
- Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014. ISSN 0045-7906. doi:10.1016/j.compeleceng.2013.11.024.
- Francois Charton. Linear algebra with transformers. *Transactions on Machine Learning Research (TMLR)*, 2022a. URL <https://openreview.net/forum?id=Hp4g7FAXXG>.
- Francois Charton. Leveraging maths to understand transformers, 2022b. URL <https://neurips.cc/virtual/2022/workshop/50015#wse-detail-63846>. Invited Talk at the NeurIPS 2nd Workshop on MATH-AI.
- Francois Charton, Amaury Hayat, and Guillaume Lample. Learning advanced mathematical computations from examples. In *International Conference on Learning Representations (ICLR)*, 2021. URL <https://openreview.net/forum?id=-gfhS00XfKj>.
- Gang Chen. Learning symbolic expressions via gumbel-max equation learner network. *CoRR*, 2020. doi:10.48550/ARXIV.2012.06921.
- Gopinath Chennupati, Nandakishore Santhi, Phillip Romero, and Stephan J. Eidenbenz. Machine learning enabled scalable performance prediction of scientific codes. *CoRR*, 2020. doi:10.48550/ARXIV.2010.04212.
- V. Cherkassky and Yunqian Ma. Comparison of loss functions for linear regression. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, volume 1, pages 395–400, 2004. doi:10.1109/IJCNN.2004.1379938.
- Davide Chicco, Matthijs J Warrens, and Giuseppe Jurman. The coefficient of determination r-squared is more informative than smape, mae, mape, mse and rmse in regression analysis evaluation. *PeerJ Computer Science*, 7:e623, 2021. doi:10.7717/peerj-cs.623.

- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayanan Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, 2022. doi:10.48550/ARXIV.2204.02311.
- Samuel Cognolato and Alberto Testolin. Transformers discover an elementary calculation system exploiting local attention and grid-like problem representation. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022. doi:10.1109/IJCNN55064.2022.9892619.
- Laurent Cohen and S Dahaene. Towards an anatomical and functional model of number processing. *Math. Cognit.*, 1:83–120, 1995.
- Mark William Craven. *Extracting comprehensible models from trained neural networks*. PhD thesis, The University of Wisconsin-Madison, 1996. URL <https://pages.cs.wisc.edu/~shavlik/abstracts/craven.thesis.abstract.html>.
- Róbert Csordás, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Are neural nets modular? inspecting functional modularity through differentiable weight masks. In *International Conference on Learning Representations (ICLR)*, 2021. doi:10.48550/ARXIV.2010.02066.
- Wang-Zhou Dai and Stephen H. Muggleton. Abductive knowledge induction from raw data. *CoRR*, 2020. doi:10.48550/ARXIV.2010.03514.
- Stéphane D’Ascoli, Pierre-Alexandre Kamienny, Guillaume Lample, and Francois Charton. Deep symbolic regression for recurrent sequences. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning (ICML)*, volume 162 of *Proceedings of Machine Learning Research*, pages 4520–4536. PMLR, 17–23 Jul 2022. URL <https://proceedings.mlr.press/v162/d-ascoli22a.html>.
- Ernest Davis. The use of deep learning for symbolic integration: A review of (Lample and Charton, 2019). *CoRR*, 2019. doi:10.48550/ARXIV.1912.05752.

- Stanislas Dehaene. Varieties of numerical abilities. *Cognition*, 44(1):1–42, 1992. doi:[https://doi.org/10.1016/0010-0277\(92\)90049-N](https://doi.org/10.1016/0010-0277(92)90049-N).
- Stanislas Dehaene. *The number sense: How the mind creates mathematics*. Oxford University Press, 1999.
- Stanislas Dehaene. *Consciousness and the brain: Deciphering how the brain codes our thoughts*. Penguin, 2014.
- Stanislas Dehaene and Jean-Pierre Changeux. Development of elementary numerical abilities: A neuronal model. *Journal of cognitive neuroscience*, 5(4):390–407, 1993. doi:10.1162/jocn.1993.5.4.390.
- Stanislas Dehaene, Ghislaine Dehaene-Lambertz, and Laurent Cohen. Abstract representations of numbers in the animal and human brain. *Trends in neurosciences*, 21(8): 355–361, 1998. doi:10.1016/S0166-2236(98)01263-6.
- Stanislas Dehaene, Elizabeth Spelke, Philippe Pinel, Ruxandra Stanescu, and Sanna Tsivkin. Sources of mathematical thinking: Behavioral and brain-imaging evidence. *Science*, 284(5416):970–974, 1999. doi:10.1126/science.284.5416.970.
- Jules Dejerine. Contribution à l'étude anatomopathologique et clinique des différents variétés de cécité verbale. *Mémoires de la Société de Biologie*, 4:61–90, 1892.
- Robert Desimone, John Duncan, et al. Neural mechanisms of selective visual attention. *Annual review of neuroscience*, 18(1):193–222, 1995. doi:10.1146/annurev.ne.18.030195.001205.
- Grant Dick. Genetic programming, standardisation, and stochastic gradient descent revisited: Initial findings on srbench. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '22*, pages 2265–2273, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392686. doi:10.1145/3520304.3534040. URL <https://doi.org/10.1145/3520304.3534040>.
- Grant Dick, Caitlin A. Owen, and Peter A. Whigham. Feature standardisation and coefficient optimisation for effective symbolic regression. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 306–314. Association for Computing Machinery, 2020. ISBN 9781450371285. doi:10.1145/3377930.3390237.
- Wouter Dobbels, Maarten Baes, Sébastien Viaene, S Bianchi, JI Davies, V Casasola, CJR Clark, J Fritz, M Galametz, F Galliano, et al. Predicting the global far-infrared sed of galaxies via machine learning techniques. *Astronomy & Astrophysics*, 634:A57, 2020. doi:10.1051/0004-6361/201936695.
- Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah A. Smith. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *CoRR*, 2020. doi:10.48550/ARXIV.2002.06305.

- Richard Durbin and David E. Rumelhart. Product units: A computationally powerful and biologically plausible extension to backpropagation networks. *Neural Computation*, 1(1):133–142, 1989. doi:10.1162/neco.1989.1.1.133.
- Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990. doi:10.1207/s15516709cog1402.1.
- Jacques A. Esterhuizen, Bryan R. Goldsmith, and Suljo Linic. Theory-guided machine learning finds geometric structure-property relationships for chemisorption on subsurface alloys. *Chem*, 6(11):3100–3117, 2020. ISSN 2451-9294. doi:<https://doi.org/10.1016/j.chempr.2020.09.001>.
- Lukas Faber and Roger Wattenhofer. Neural status registers. *CoRR*, 2020. doi:10.48550/ARXIV.2004.07085.
- Fenglei Fan, Wenxiang Cong, and Ge Wang. Generalized backpropagation algorithm for training second-order neural networks. *International Journal for Numerical Methods in Biomedical Engineering*, 34(5):e2956, 2018. doi:<https://doi.org/10.1002/cnm.2956>.
- Samuel G Finlayson. *Learning Inductive Representations of Biomedical Data*. PhD thesis, Harvard University, 2020. URL <https://dash.harvard.edu/handle/1/37368883>.
- Leonardo Franco and Sergio A. Cannas. Solving arithmetic problems using feed-forward neural networks. *Neurocomputing*, 18(1):61–79, 1998. ISSN 0925-2312. doi:[https://doi.org/10.1016/S0925-2312\(97\)00069-6](https://doi.org/10.1016/S0925-2312(97)00069-6).
- Karlis Freivalds and Renars Liepins. Improving the neural GPU architecture for algorithm learning. *CoRR*, 2017. doi:10.48550/ARXIV.1702.08727.
- Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feed-forward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010. URL <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>.
- Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015. doi:10.48550/ARXIV.1412.6572.
- Diana F Gordon and Marie Desjardins. Evaluation and selection of biases in machine learning. *Machine learning*, 20(1):5–22, 1995. doi:10.1023/A:1022630017346.
- Anirudh Goyal and Yoshua Bengio. Inductive biases for deep learning of higher-level cognition. *Proceedings of the Royal Society A*, 2022. doi:10.1098/rspa.2021.0068.
- Anirudh Goyal, Aniket Rajiv Didolkar, Nan Rosemary Ke, Charles Blundell, Philippe Beaudoin, Nicolas Heess, Michael Curtis Mozer, and Yoshua Bengio. Neural



- production systems. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2021a. doi:10.48550/ARXIV.2103.01937.
- Anirudh Goyal, Alex Lamb, Phanideep Gampa, Philippe Beaudoin, Charles Blundell, Sergey Levine, Yoshua Bengio, and Michael Curtis Mozer. Factorizing declarative and procedural knowledge in structured, dynamical environments. In *International Conference on Learning Representations (ICLR)*, 2021b. URL <https://openreview.net/forum?id=VVdmjgu7pKM>.
- Anirudh Goyal, Alex Lamb, Jordan Hoffmann, Shagun Sodhani, Sergey Levine, Yoshua Bengio, and Bernhard Schölkopf. Recurrent independent mechanisms. In *International Conference on Learning Representations (ICLR)*, 2021c. URL <https://openreview.net/forum?id=mLcmd1EUxy->.
- Anirudh Goyal, Aniket Rajiv Didolkar, Alex Lamb, Kartikeya Badola, Nan Rosemary Ke, Nasim Rahaman, Jonathan Binas, Charles Blundell, Michael Curtis Mozer, and Yoshua Bengio. Coordination among neural modules through a shared global workspace. In *International Conference on Learning Representations (ICLR)*, 2022. doi:10.48550/ARXIV.2103.01197.
- Klaus Greff, Sjoerd van Steenkiste, and Jürgen Schmidhuber. On the binding problem in artificial neural networks. *CoRR*, 2020. doi:10.48550/ARXIV.2012.05208.
- IEEE 754 Working Group et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi:10.1109/IEEESTD.2019.8766229.
- Andras Hajnal, Wolfgang Maass, Pavel Pudlak, Mario Szegedy, and Gyorgy Turan. Threshold circuits of bounded depth. In *28th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 99–110, 1987. doi:10.1109/SFCS.1987.59.
- Serhii Havrylov and Ivan Titov. Emergence of language with multi-agent games: Learning to communicate with sequences of symbols. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, pages 2146–2156, 2017. ISBN 9781510860964. URL <https://proceedings.neurips.cc/paper/2017/file/70222949cc0db89ab32c9969754d4758-Paper.pdf>.
- Hussein Hazimeh, Zhe Zhao, Aakanksha Chowdhery, Maheswaran Sathiamoorthy, Yihua Chen, Rahul Mazumder, Lichan Hong, and Ed Chi. DSelect-k: Differentiable selection in the mixture of experts with applications to multi-task learning. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. doi:10.48550/ARXIV.2106.03760.

- Niklas Heim, Václav Šmídl, and Tomáš Pevný. Rodent: Relevance determination in differential equations. *CoRR*, 2019. doi:10.48550/ARXIV.1912.00656.
- Niklas Heim, Tomas Pevny, and Vasek Smidl. Neural power units. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 6573–6583, 2020. doi:10.48550/ARXIV.2006.01681.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Advances in Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*, 2021. URL <https://openreview.net/forum?id=7Bywt2mQsCe>.
- Theodore P. Hill. A Statistical Derivation of the Significant-Digit Law. *Statistical Science*, 10(4):354 – 363, 1995. doi:10.1214/ss/1177009869.
- M. Hittmair-Delazer, C. Semenza, and G. Denes. Concepts and facts in calculation. *Brain*, 117(4):715–728, 08 1994. ISSN 0006-8950. doi:10.1093/brain/117.4.715.
- Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. doi:10.1162/neco.1997.9.8.1735.
- Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080. doi:10.1016/0893-6080(89)90020-8.
- Yedid Hoshen and Shmuel Peleg. Visual learning of arithmetic operations. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 3733–3739, 2016. doi:10.1609/aaai.v30i1.9882.
- Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 29, pages 4107–4115, 2016. URL <http://papers.nips.cc/paper/6573-binarized-neural-networks>.
- HuggingFace. Hugging face: The ai community building the future. <https://huggingface.co>, 2022. Accessed: 2022-10-26.
- I. M. Huijben, W. Kool, M. B. Paulus, and R. G. van Sloun. A review of the gumbel-max trick and its extensions for discrete stochasticity in machine learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(02):1353–1371, 2023. ISSN 1939-3539. doi:10.1109/TPAMI.2022.3157042.
- Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, volume 37

- of PMLR, pages 448–456. PMLR, 2015. URL <https://proceedings.mlr.press/v37/ioffe15.html>.
- Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991. doi:10.1162/neco.1991.3.1.79.
- Alon Jacovi, Guy Hadash, Einat Kermany, Boaz Carmeli, Ofer Lavi, George Kour, and Jonathan Berant. Neural network gradient-based learning of black-box function interfaces. In *International Conference on Learning Representations (ICLR)*, 2019. doi:10.48550/ARXIV.1901.03995.
- Max Jaderberg, Karen Simonyan, Andrew Zisserman, and koray kavukcuoglu. Spatial transformer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 28, 2015. URL <https://proceedings.neurips.cc/paper/2015/file/33ceb07bf4eeb3da587e268d663aba1a-Paper.pdf>.
- William James. *The principles of psychology, Vol I*. Henry Holt and Co, 1890. doi:10.1037/10538-000.
- Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations (ICLR)*, 2017. doi:10.48550/ARXIV.1611.01144.
- T. Jia, Y. Ju, R. Joseph, and J. Gu. Ncpu: An embedded neural cpu architecture on resource-constrained low power devices for real-time end-to-end performance. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1097–1109, 2020. doi:10.1109/MICRO50266.2020.00091.
- Hongbo Jiang, Mengyuan Wang, Ping Zhao, Zhu Xiao, and Schahram Dustdar. A utility-aware general framework with quantifiable privacy preservation for destination prediction in lbss. *IEEE/ACM Transactions on Networking*, 29(5):2228–2241, 2021. doi:10.1109/TNET.2021.3084251.
- Michael I Jordan. Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier, 1997. doi:10.1016/S0166-4115(97)80111-2.
- M. Joseph-Rivlin, A. Zvirin, and R. Kimmel. Momenêt: Flavor the moments in learning to classify shapes. In *IEEE/CVF Workshop on International Conference on Computer Vision (ICCVW)*, pages 4085–4094, 2019. doi:10.1109/ICCVW.2019.00503.
- Alex Kacelnik and Fausto Brito e Abreu. Risky choice and weber’s law. *Journal of Theoretical Biology*, 194(2):289–298, 1998. ISSN 0022-5193. doi:<https://doi.org/10.1006/jtbi.1998.0763>.

- Daniel Kahneman. *Thinking, fast and slow*. Macmillan, 2011.
- Lukasz Kaiser and Ilya Sutskever. Neural gpu learn algorithms. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations (ICLR)*, 2016. doi:10.48550/ARXIV.1511.08228.
- Pierre-Alexandre Kamienny, Stéphane d’Ascoli, Guillaume Lample, and Francois Charton. End-to-end symbolic regression with transformers. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL [https://openreview.net/forum?id=GoOuIrDHG\\_Y](https://openreview.net/forum?id=GoOuIrDHG_Y).
- E.R. Kandel, J.H. Schwartz, T.M. Jessell, S.A. Siegelbaum, and A.J. Hudspeth. *Principles of Neural Science, Fifth Edition*. McGraw-Hill’s AccessMedicine. McGraw-Hill Education, 2013. ISBN 9780071390118.
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, 2015. doi:10.48550/ARXIV.1412.6980.
- Michael Kommenda, Bogdan Burlacu, Gabriel Kronberger, and Michael Affenzeller. Parameter identification for symbolic regression using nonlinear least squares. *Genetic Programming and Evolvable Machines*, 21(3):471–501, 2020. doi:10.1007/s10710-019-09371-3.
- Neehar Kondapaneni and Pietro Perona. A number sense as an emergent property of the manipulating brain. *CoRR*, 2020. doi:10.48550/ARXIV.2012.04132.
- John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994. doi:<https://doi.org/10.1007/BF00175355>.
- Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *International Conference on Learning Representations (ICLR)*, 2020. URL <https://openreview.net/forum?id=S1eZYeHFDS>.
- Mikel Landajuela, Chak Lee, Jiachen Yang, Ruben Glatt, Claudio P. Santiago, Ignacio Aravena, Terrell N. Mundhenk, Garrett Mulcahy, and Brenden K. Petersen. A unified framework for deep symbolic regression. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL <https://openreview.net/forum?id=2FNnBhwJsHK>.
- Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi:10.1109/5.726791.

- Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-35289-8\_3.
- Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multi-layer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993. ISSN 0893-6080. doi:[https://doi.org/10.1016/S0893-6080\(05\)80131-5](https://doi.org/10.1016/S0893-6080(05)80131-5).
- Aitor Lewkowycz, Anders Johan Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Venkatesh Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL <https://openreview.net/forum?id=IFXTZERXdm7>.
- Cathy Li, Jana Sotáková, Emily Wenger, Mohamed Malhou, Evrard Garcelon, Francois Charton, and Kristin Lauter. Salsa picante: a machine learning attack on lwe with binary secrets. *CoRR*, 2023. URL <https://arxiv.org/abs/2303.04178>.
- Haoran Li, Yang Weng, and Hanghang Tong. CoNSole: Convex neural symbolic learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. doi:10.48550/ARXIV.2206.00257.
- Li Li, Minjie Fan, Rishabh Singh, and Patrick Riley. Neural-guided symbolic regression with asymptotic constraints. *CoRR*, 2019. doi:10.48550/ARXIV.1901.07714.
- Qing Li, Siyuan Huang, Yining Hong, Yixin Zhu, Ying Nian Wu, and Song-Chun Zhu. A hint from arithmetic: On systematic generalization of perception, syntax, and semantics. In *ICLR Workshop on the Role of Mathematical Reasoning in General Artificial Intelligence (MATHAI)*, 2021. URL [https://mathai-iclr.github.io/papers/papers/MATHAI\\_1\\_paper.pdf](https://mathai-iclr.github.io/papers/papers/MATHAI_1_paper.pdf).
- Zachary Chase Lipton. The mythos of model interpretability. *CoRR*, 2016. doi:10.48550/ARXIV.1606.03490.
- Dianbo Liu, Alex M Lamb, Kenji Kawaguchi, Anirudh Goyal ALIAS PARTH GOYAL, Chen Sun, Michael C Mozer, and Yoshua Bengio. Discrete-valued neural communication. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 34, pages 2109–2121, 2021. URL <https://proceedings.neurips.cc/paper/2021/file/10907813b97e249163587e6246612e21-Paper.pdf>.

- Dianbo Liu, Alex Lamb, Xu Ji, Pascal Notsawo, Mike Mozer, Yoshua Bengio, and Kenji Kawaguchi. Adaptive discrete communication bottlenecks with dynamic vector quantization. *CoRR*, 2022. doi:10.48550/ARXIV.2202.01334.
- Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through  $l_0$  regularization. In *International Conference on Learning Representations (ICLR)*, 2018. doi:10.48550/ARXIV.1712.01312.
- Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *International Conference on Learning Representations (ICLR)*, 2017. URL <https://openreview.net/forum?id=S1jE5L5g1>.
- Andreas Madsen and Alexander Rosenberg Johansen. Measuring arithmetic extrapolation performance. In *NeurIPS Workshop on Science meets Engineering of Deep Learning*, October 2019. doi:10.48550/ARXIV.1910.01888.
- Andreas Madsen and Alexander Rosenberg Johansen. Neural arithmetic units. In *International Conference on Learning Representations (ICLR)*, 2020. doi:10.48550/ARXIV.2001.05016.
- Ashok Makkuva, Sewoong Oh, Sreeram Kannan, and Pramod Viswanath. Learning in gated neural networks. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 108 of *Proceedings of Machine Learning Research*, pages 3338–3348. PMLR, 26–28 Aug 2020. doi:10.48550/ARXIV.1906.02777.
- Georg S Martius and Christoph Lampert. Extrapolation and learning equations. In *5th International Conference on Learning Representations (ICLR) Workshop Track Proceedings*, 2017. URL <https://openreview.net/pdf?id=BkgRp0FYe>.
- E.J. Martínez-Estudillo, C. Hervás-Martínez, P.A. Gutiérrez, and A.C. Martínez-Estudillo. Evolutionary product-unit neural networks classifiers. *Neurocomputing*, 72(1):548–561, 2008. ISSN 0925-2312. doi:10.1016/j.neucom.2007.11.019.
- Anna A. Matejko and Daniel Ansari. Drawing connections between white matter and numerical and mathematical cognition: A literature review. *Neuroscience & Biobehavioral Reviews*, 48:35–52, 2015. ISSN 0149-7634. doi:10.1016/j.neubiorev.2014.11.006.
- R. Thomas McCoy, Robert Frank, and Tal Linzen. Does syntax need to grow on trees? sources of hierarchical inductive bias in sequence-to-sequence networks. *Transactions of the Association for Computational Linguistics (TACL)*, 8:125–140, 2020. doi:10.1162/tacl.a.00304.
- Ryszard S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20(2):111–161, 1983. ISSN 0004-3702. doi:[https://doi.org/10.1016/0004-3702\(83\)90016-4](https://doi.org/10.1016/0004-3702(83)90016-4).

- Matthias Michel, Stephen M Fleming, Hakwan Lau, Alan LF Lee, Susana Martinez-Conde, Richard E Passingham, Megan AK Peters, Dobromir Rahnev, Claire Sergent, and Kayuet Liu. An informal internet survey on the current state of consciousness science. *Frontiers in psychology*, 9:2134, 2018. doi:10.3389/fpsyg.2018.02134.
- Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. A primer for neural arithmetic logic modules. *Journal of Machine Learning Research (JMLR)*, 23(185):1–58, 2022a. doi:10.48550/ARXIV.2101.09530.
- Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. Exploring the learning mechanisms of neural division modules. *Transactions on Machine Learning Research (TMLR)*, 2022b. ISSN 2835-8856. URL <https://openreview.net/forum?id=HjelcW6wio>.
- Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. Improving the robustness of neural multiplication units with reversible stochasticity. *CoRR*, 2022c. doi:10.48550/ARXIV.2211.05624.
- Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency (FAccT)*, pages 220–229, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361255. doi:10.1145/3287560.3287596.
- Tom M Mitchell. The need for biases in learning generalizations. Technical report, Department of Computer Science, Laboratory for Computer Science Research, Rutgers University, 1980.
- Sarthak Mittal, Alex Lamb, Anirudh Goyal, Vikram Voleti, Murray Shanahan, Guillaume Lajoie, Michael Mozer, and Yoshua Bengio. Learning to combine top-down and bottom-up signals in recurrent neural networks with attention over modules. In *International Conference on Machine Learning (ICML)*, pages 6972–6986. PMLR, 2020. doi:10.48550/ARXIV.2006.16981.
- Sarthak Mittal, Yoshua Bengio, and Guillaume Lajoie. Is a modular architecture enough? In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2022a. URL <https://openreview.net/forum?id=3-3XMModtrx>.
- Sarthak Mittal, Sharath Chandra Raparthy, Irina Rish, Yoshua Bengio, and Guillaume Lajoie. Compositional attention: Disentangling search and retrieval. In *International Conference on Learning Representations (ICLR)*, 2022b. URL <https://openreview.net/forum?id=IwJPj2MBcIa>.
- Knut Mørken. Numerical algorithms and digital representation, 2013. URL <https://www.uio.no/studier/emner/matnat/math/MAT-INF1100/h13/kompendiet/matinf1100.pdf>.

- Aakanksha Naik, Abhilasha Ravichander, Carolyn Rose, and Eduard Hovy. Exploring numeracy in word embeddings. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 3374–3380. Association for Computational Linguistics, July 2019. doi:10.18653/v1/P19-1329.
- Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *CoRR*, 2015. doi:10.48550/ARXIV.1511.06807.
- Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 01 2011.
- Andreas Nieder. Neuroethology of number sense across the animal kingdom. *Journal of Experimental Biology*, 224(6), 03 2021. ISSN 0022-0949. doi:10.1242/jeb.218289.
- Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.
- Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of transformers with simple arithmetic tasks. In *ICLR Workshop on the Role of Mathematical Reasoning in General Artificial Intelligence (MATHAI)*, 2021. URL [https://mathai-iclr.github.io/papers/papers/MATHAI\\_11\\_paper.pdf](https://mathai-iclr.github.io/papers/papers/MATHAI_11_paper.pdf).
- Bastien Nollet, Mathieu Lefort, and Frédéric Armetta. Learning arithmetic operations with a multistep deep learning. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020. doi:10.1109/IJCNN48605.2020.9206963.
- Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *CoRR*, 2021. doi:10.48550/ARXIV.2112.00114.
- Randall C O'Reilly and Jerry W Rudy. Conjunctive representations in learning and memory: principles of cortical and hippocampal function. *Psychological review*, 108(2):311, 2001. doi:10.1037/0033-295x.108.2.311.
- Felipe Oviedo, Juan Lavista Ferres, Tonio Buonassisi, and Keith T. Butler. Interpretable and explainable machine learning for materials science and chemistry. *Accounts of Materials Research*, 3(6):597–607, 2022. doi:10.1021/accountsmr.1c00244.
- Caitlin A. Owen, Grant Dick, and Peter A. Whigham. Feature standardisation in symbolic regression. In Tanja Mitrovic, Bing Xue, and Xiaodong Li, editors, *AI 2018: Advances in Artificial Intelligence*, pages 565–576. Springer International Publishing, 2018. doi:10.1007/978-3-030-03991-2\_52.



- Sejun Park, Chulhee Yun, Jaeho Lee, and Jinwoo Shin. Minimum width for universal approximation. In *International Conference on Learning Representations (ICLR)*, 2021. URL <https://openreview.net/forum?id=0-XJwyoIF-k>.
- Max B Paulus, Chris J. Maddison, and Andreas Krause. Rao-blackwellizing the straight-through gumbel-softmax gradient estimator. In *International Conference on Learning Representations (ICLR)*, 2021. URL <https://openreview.net/forum?id=Mk6PZtgAgfq>.
- Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 690–702. Association for Computing Machinery, 2021. ISBN 9781450385626. doi:10.1145/3468264.3468607.
- Charles Sanders Peirce. *Reasoning and the logic of things: The Cambridge conferences lectures of 1898*. Harvard University Press, 1992.
- Pietro Perona. How did you learn the natural numbers?, Dec. 2020. URL <https://www.youtube.com/watch?v=TJRsqFovVmY>.
- Ellen Peters, Daniel Västfjäll, Paul Slovic, C.K. Mertz, Ketti Mazzocco, and Stephan Dickert. Numeracy and decision making. *Psychological Science*, 17(5):407–413, 2006. doi:10.1111/j.1467-9280.2006.01720.x.
- Michael I Posner and Charles R R Snyder. *Attention and Cognitive Control*. Key readings in cognition. Psychology Press, 2004. ISBN 1-84169-064-3.
- Ravi Prakash, Om Prakash, Shashi Prakash, Priyadarshi Abhishek, and Sachin Gandotra. Global workspace model of consciousness and its electromagnetic correlates. *Annals of Indian Academy of Neurology*, 11(3):146, 2008. doi:10.4103/0972-2327.42933.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. URL [https://d4mucfpksywv.cloudfront.net/better-language-models/language\\_models\\_are\\_unsupervised\\_multitask\\_learners.pdf](https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf).
- Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John

- Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Llorayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher. *CoRR*, 2021. doi:10.48550/ARXIV.2112.11446.
- Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research (JMLR)*, 21(140):1–67, 2020. URL <http://jmlr.org/papers/v21/20-074.html>.
- Nasim Rahaman, Muhammad Waleed Gondal, Shruti Joshi, Peter Vincent Gehler, Yoshua Bengio, Francesco Locatello, and Bernhard Schölkopf. Dynamic inference with neural interpreters. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2021a. doi:10.48550/ARXIV.2110.06399.
- Nasim Rahaman, Anirudh Goyal, Muhammad Waleed Gondal, Manuel Wuthrich, Stefan Bauer, Yash Sharma, Yoshua Bengio, and Bernhard Schölkopf. Spatially structured recurrent modules. In *International Conference on Learning Representations (ICLR)*, 2021b. URL <https://openreview.net/forum?id=519zj5G7vDY>.
- Aditya Raj, Pooja Consul, and Sakar K Pal. Fast neural accumulator (NAC) based badminton video action classification. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Proceedings of Intelligent Systems and Applications (ISWA)*, pages 452–467. Springer, 2020. doi:10.1007/978-3-030-55180-3\_34.
- Shangeth Rajaa and Jajati Keshari Sahoo. Convolutional feature extraction and neural arithmetic logic units for stock prediction. In Mayank Singh, P.K. Gupta, Vipin Tyagi, Jan Flusser, Tuncer Ören, and Rekha Kashyap, editors, *Advances in Computing and Data Sciences*, pages 349–359, Singapore, 2019. Springer Singapore. doi:10.1007/978-981-13-9939-8\_31.
- Ashish Rana, Avleen Malhi, and Kary Främling. Exploring numerical calculations with calcnet. In *31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1374–1379. IEEE, 2019. doi:10.1109/ICTAI.2019.00192. URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8995315>.

- Ashish Rana, Taranveer Singh, Harpreet Singh, Neeraj Kumar, and Prashant Singh Rana. Systematically designing better instance counting models on cell images with neural arithmetic logic units. *CoRR*, 2020. doi:10.48550/ARXIV.2004.06674.
- Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016. doi:10.1109/CVPR.2016.91.
- Jan Niclas Reimann and Andreas Schwung. Neural logic rule layers. *CoRR*, 2019. doi:10.13140/RG.2.2.10091.59687.
- Larry Rendell. Similarity-based learning and its extensions. *Computational Intelligence*, 3(1):241–266, 1987. doi:10.1111/j.1467-8640.1987.tb00213.x.
- Ribana Roscher, Bastian Bohn, Marco F. Duarte, and Jochen Garcke. Explainable machine learning for scientific insights and discoveries. *IEEE Access*, 8:42200–42216, 2020. doi:10.1109/ACCESS.2020.2976199.
- Clemens Rosenbaum, Ignacio Cases, Matthew Riemer, and Tim Klinger. Routing networks and the challenges of modular and compositional computation. *CoRR*, 2019. doi:10.48550/ARXIV.1904.12774.
- Rosa Rugani, Giorgio Vallortigara, Konstantinos Priftis, and Lucia Regolin. Number-space mapping in the newborn chick resembles humans’ mental number line. *Science*, 347(6221):534–536, 2015. doi:10.1126/science.aaa1379.
- David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. doi:10.1038/323533a0.
- Jacob Russin, Roland Fernandez, Hamid Palangi, Eric Rosen, Nebojsa Jojic, Paul Smolensky, and Jianfeng Gao. Compositional processing emerges in neural networks solving math problems. In *Proceedings of the Annual Meeting of the Cognitive Science Society (CogSci)*, volume 2021, pages 1767–1773. NIH Public Access, 2021.
- Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, pages 3859–3869, 2017. doi:10.48550/ARXIV.1710.09829.
- Subham Sahoo, Christoph Lampert, and Georg Martius. Learning equations for extrapolation and control. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning (ICML)*, volume 80 of *Proceedings of Machine Learning Research*, pages 4442–4450. PMLR, 10–15 Jul 2018. doi:10.48550/ARXIV.1806.07259.

- David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations (ICLR)*, 2019. doi:10.48550/ARXIV.1904.01557.
- Daniel Schlör, Markus Ring, Anna Krause, and Andreas Hotho. Financial fraud detection with improved neural arithmetic logic units. In *Mining Data for Financial Applications: 5th European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD) Workshop on Mining Data for financial applicationS (MIDAS), Revised Selected Papers*, pages 40–54. Springer-Verlag, 2020. ISBN 978-3-030-66980-5. doi:10.1007/978-3-030-66981-2\_4.
- Daniel Schlör, Markus Ring, and Andreas Hotho. iNALU: Improved neural arithmetic logic unit. *Frontiers in Artificial Intelligence*, 3:71, 2020. ISSN 2624-8212. doi:10.3389/frai.2020.00071.
- Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009. doi:10.1126/science.1165893.
- Bernhard Schölkopf, Dominik Janzing, Jonas Peters, Eleni Sgouritsa, Kun Zhang, and Joris Mooij. On causal and anticausal learning. In *Proceedings of the 29th International Conference on International Conference on Machine Learning (ICML)*, pages 459–466, 2012. ISBN 9781450312851.
- Bernhard Schölkopf, Francesco Locatello, Stefan Bauer, Nan Rosemary Ke, Nal Kalchbrenner, Anirudh Goyal, and Yoshua Bengio. Toward causal representation learning. *Proceedings of the IEEE*, 109(5):612–634, 2021. doi:10.1109/JPROC.2021.3058954.
- Carson D. Sestili, William S. Snavely, and Nathan M. VanHoudnos. Towards security defect prediction with AI. *CoRR*, 2018. doi:10.48550/ARXIV.1808.09897.
- Baoguang Shi, Xinggang Wang, Pengyuan Lyu, Cong Yao, and Xiang Bai. Robust scene text recognition with automatic rectification. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4168–4176. IEEE Computer Society, 2016. doi:10.1109/CVPR.2016.452.
- Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019. doi:10.1186/s40537-019-0197-0.
- K.-Y. Siu and J. Bruck. Neural computation of arithmetic functions. *Proceedings of the IEEE*, 78(10):1669–1675, 1990. doi:10.1109/5.58350.
- K.-Y. Siu, J. Bruck, T. Kailath, and T. Hofmeister. Depth efficient neural networks for division and related problems. *IEEE Transactions on Information Theory*, 39(3):946–956, 1993. doi:10.1109/18.256501.
- SRBench. SRBench: A living benchmark for symbolic regression. <https://cavalab.org/srbench/>, 2023. Accessed: 2023-01-19.

- Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, volume 28, pages 1139–1147. PMLR, 2013. URL <https://proceedings.mlr.press/v28/sutskever13.html>.
- Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI Conference on Artificial Intelligence*, 2017. doi:10.5555/3298023.3298188.
- Daniel Teitelman, Itay Naeh, and Shie Mannor. Stealing black-box functionality using the deep neural tree architecture. *CoRR*, 2020. doi:10.48550/ARXIV.2002.09864.
- Christine M Temple. Procedural dyscalculia and number fact dyscalculia: Double dissociation in developmental dyscalculia. *Cognitive neuropsychology*, 8(2):155–176, 1991. doi:10.1080/02643299108253370.
- Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Agueras-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. Lamda: Language models for dialog applications. *CoRR*, 2022. doi:10.48550/ARXIV.2201.08239.
- Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996. doi:10.1111/j.2517-6161.1996.tb02080.x.
- Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8035–8044, 2018. URL <https://openreview.net/pdf?id=H1gN0eHKPS>.
- Doris Y Tsao, Winrich A Freiwald, Roger BH Tootell, and Margaret S Livingstone. A cortical region consisting entirely of face-selective cells. *Science*, 311(5761):670–674, 2006. doi:10.1126/science.1119983.
- Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16), 2020. doi:10.1126/sciadv.aay2631.

- Paul E Utgoff. Shift of bias for inductive concept learning. *Machine learning: An artificial intelligence approach*, 2:107–148, 1986.
- Leen Van Beek, Pol Ghesquière, Lieven Lagae, and Bert De Smedt. Left fronto-parietal white matter correlates with individual differences in children’s ability to solve additions and multiplications: A tractography study. *NeuroImage*, 90:117–127, 2014. ISSN 1053-8119. doi:10.1016/j.neuroimage.2013.12.030.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, volume 30, pages 6000–6010, 2017. doi:10.48550/ARXIV.1706.03762.
- Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning (ICML)*, page 1096–1103. Association for Computing Machinery, 2008. ISBN 9781605582054. doi:10.1145/1390156.1390294.
- Marco Virgolin and Solon P Pissis. Symbolic regression is NP-hard. *Transactions on Machine Learning Research (TMLR)*, 2022. ISSN 2835-8856. URL <https://openreview.net/forum?id=LTiaPxqe2e>.
- Digvijay Wadekar, Leander Thiele, Francisco Villaescusa-Navarro, J. Colin Hill, Miles Cranmer, David N. Spergel, Nicholas Battaglia, Daniel Anglés-Alcázar, Lars Hernquist, and Shirley Ho. Augmenting astrophysical scaling relations with machine learning: Application to reducing the sunyaev–zeldovich flux–mass scatter. *Proceedings of the National Academy of Sciences*, 120(12), 2023. doi:10.1073/pnas.2202074120.
- Eric Wallace, Yizhong Wang, Sujian Li, Sameer Singh, and Matt Gardner. Do NLP models know numbers? probing numeracy in embeddings. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5307–5315. Association for Computational Linguistics, 2019. doi:10.18653/v1/D19-1534.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023. URL <https://openreview.net/forum?id=1PL1NIMMrw>.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL [https://openreview.net/forum?id=\\_VjQlMeSB\\_J](https://openreview.net/forum?id=_VjQlMeSB_J).

- Sean Welleck, Peter West, Jize Cao, and Yejin Choi. Symbolic brittleness in sequence models: on systematic generalization in symbolic mathematics. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8629–8637, 2022. doi:10.1609/aaai.v36i8.20841.
- Emily Wenger, Mingjie Chen, Francois Charton, and Kristin Lauter. SALSA: Attacking lattice cryptography with transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL <https://openreview.net/forum?id=p4xLHcTLRwh>.
- Matthias Werner, Andrej Junginger, Philipp Hennig, and Georg Martius. Informed equation learning. *CoRR*, 2021. doi:10.48550/ARXIV.2105.06331.
- Sam Wiseman and Alexander M. Rush. Sequence-to-sequence learning as beam-search optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1296–1306. Association for Computational Linguistics, November 2016. doi:10.18653/v1/D16-1137.
- Matthias Wittlinger, Rüdiger Wehner, and Harald Wolf. The ant odometer: stepping on stilts and stumps. *Science*, 312(5782):1965–1967, 2006. doi:10.1126/science.1126912.
- D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi:10.1109/4235.585893.
- Bo Wu, Haoyu Qin, Alireza Zareian, Carl Vondrick, and Shih-Fu Chang. Analogical reasoning for visually grounded language acquisition. *CoRR*, 2020. doi:10.48550/ARXIV.2007.11668.
- Zhu Xiao, Fancheng Li, Ronghui Wu, Hongbo Jiang, Yupeng Hu, Ju Ren, Chenglin Cai, and Arun Iyengar. Trajdata: On vehicle trajectory collection with commodity plug-and-play obu devices. *IEEE Internet of Things Journal*, 7(9):9066–9079, 2020. doi:10.1109/JIOT.2020.3001566.
- Zhu Xiao, Hui Fang, Hongbo Jiang, Jing Bai, Vincent Havyarimana, Hongyang Chen, and Licheng Jiao. Understanding private car aggregation effect via spatio-temporal analysis of trajectory data. *IEEE Transactions on Cybernetics*, pages 1–12, 2021. doi:10.1109/TCYB.2021.3117705.
- Fei Xu, Elizabeth S Spelke, and Sydney Goddard. Number sense in human infants. *Developmental science*, 8(1):88–101, 2005. doi:10.1111/j.1467-7687.2005.00395.x.
- Keyulu Xu, Mozhi Zhang, Jingling Li, Simon Shaolei Du, Ken-Ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *International Conference on Learning Representations (ICLR)*, 2021. URL <https://openreview.net/forum?id=UH-cmocLJC>.

- Yutaro Yamada, Ofir Lindenbaum, Sahand Negahban, and Yuval Kluger. Feature selection using stochastic gates. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pages 10648–10659. PMLR, 13–18 Jul 2020. URL <https://proceedings.mlr.press/v119/yamada20a.html>.
- Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural execution engines: Learning to execute subroutines. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, 2020. doi:10.48550/ARXIV.2006.08084.
- Dagmar Zeithamova, Margaret Schlichting, and Alison Preston. The hippocampus and inferential reasoning: building memories to navigate future decisions. *Frontiers in Human Neuroscience*, 6, 2012. ISSN 1662-5161. doi:10.3389/fnhum.2012.00070.
- Qibo Zhang, Fanzi Zeng, Zhu Xiao, Hongbo Jiang, Amelia C. Regan, Kehua Yang, and Yongdong Zhu. Toward predicting stay time for private car users: A rnn-nalu approach. *IEEE Transactions on Vehicular Technology*, 71(6):6007–6018, 2022. doi:10.1109/TVT.2022.3164978.
- Rui-Xiao Zhang, Tianchi Huang, Ming Ma, Haitian Pang, Xin Yao, Chenglei Wu, and Lifeng Sun. Enhancing the crowdsourced live streaming: A deep reinforcement learning approach. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV '19*, pages 55–60. Association for Computing Machinery, 2019a. ISBN 9781450362986. doi:10.1145/3304112.3325607.
- Rui-Xiao Zhang, Ming Ma, Tianchi Huang, Haitian Pang, Xin Yao, Chenglei Wu, Jiangchuan Liu, and Lifeng Sun. Livesmart: A qos-guaranteed cost-minimum framework of viewer scheduling for crowdsourced live streaming. In *Proceedings of the 27th ACM International Conference on Multimedia, MM '19*, pages 420–428. Association for Computing Machinery, 2019b. ISBN 9781450368896. doi:10.1145/3343031.3351013.
- Hattie Zhou, Azade Nova, Hugo Larochelle, Aaron Courville, Behnam Neyshabur, and Hanie Sedghi. Teaching algorithmic reasoning via in-context learning. *CoRR*, 2022. doi:10.48550/ARXIV.2211.09066.
- Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *International Conference on Learning Representations (ICLR)*, 2017. doi:10.48550/ARXIV.1612.01064.