Exploring the Learning Mechanisms of Neural Division Modules

Anonymous authors
Paper under double-blind review

Abstract

Of the four fundamental arithmetic operations (+, -, ×, ÷), division is considered the most difficult for both humans and computers. In this paper, we show that robustly learning division in a systematic manner remains a challenge even at the simplest level of dividing two numbers. We discover that robustness is greatly effected by the sign, magnitude and distribution of the input. We propose two novel approaches for division which we call the Neural Reciprocal Unit (NRU) and the Neural Multiplicative Reciprocal Unit (NMRU), and present improvements for an existing division module, the Real Neural Power Unit (Real NPU). Experiments in learning division with input redundancy on 225 different Uniform distributed training sets, find that our proposed modifications to the Real NPU obtains an average success of 85.3% improving over the original by 15.1%. In light of this modification to the Real NPU, our NMRU approach further improves the success rate to 91.6%.

1 Introduction

Division is one of the four fundamental arithmetic operations and is necessary for expressing real-world dynamical systems (Sahoo et al., 2018) or physics-based formulas (Udrescu & Tegmark, 2020a). However, the properties of division of values around zero leads to undesirable gradients for training Neural Networks through backpropogation making it the hardest operation to learn (Mistry et al., 2021). Networks try to learn division naively (such as MLPs) are unable to deal with the fluctuant gradients caused by the asymptotic nature and discontinuities in division (Trask et al., 2018). Furthermore, if the network lacks an appropriate bias for learning division, it can also lead to poor generalisation on out-of-distribution data. For example, a network could learn to achieve a reasonable loss on a training/validation range between $[1, 2]$ but be unable to maintain a reasonable loss when tested outside of $[1, 2]$ such as range $[2, 10]$.

In particular, 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 machine learning, 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. The main challenge of this task for a network comes from learning the selection and operation at the same time, which can lead to conflicting priorities when learning weights.

Selecting relevant inputs/features is a desirable property of neural networks useful for improved intepretability, reduced pre-processing costs and greater generalisation (Chandrashekar & Sahin, 2014). As differentable specialist modules (such as those for arithmetic operations) can be integrated with overparametrized networks as an intermediate module Mistry et al. (2021), being able to successfully select only the relevant inputs is important. Furthermore, as stated by Madsen & Johansen (2020), 'redundant units are very common in neural networks'. However, even recent models still struggle to learn division when there is input redundancy (Schlör et al., 2020).

Can we build models which can learn division in the presence of its undesirable, yet valid, properties? [1] We aim to address this question in this paper. Specifically, we contribute the following:[2]

[1] A desiderata for building a division module is provided in Appendix A.
[2] Code (MIT license) available at: `https://anonymous.4open.science/r/nalm-division-12EC`.

- We show how additional biases improves an existing division module, the Real NPU (Heim et al., 2020), by including: clipping, discretisation and constrained initialisation to improve performance in learning division on different training ranges.

- We develop two novel division modules, the NRU and the NMRU, by extending an existing multiplication unit. Through rigorous experiments we find both modules outperform the Real NPU for the no input redundancy tasks, with the NMRU also outperforming the Real NPU for the input redundancy tasks.

- We identify the types of data which hinders learning division for each module, including training on: mixed-sign inputs, negative ranges, extremely small values and different distributions. These difficulties can be sufficiently identified using synthetic division tasks.

2 Related Work

Symbolic regression searches the space of expressions to predict a mathematical expression from a given set of input-output observations. Compared to black box Neural Network (NN) functions, expressing a mathematical expression is significantly easier to interpret. Symbolic regression can be implemented using Genetic Programming using Evolutionary Algorithms (EA) which learn mathematical expressions. EAs maintain a population of expressions where individuals of the population get selected based-off a fitness function and modified via techniques such as crossover and mutation. This procedure is repeated until a stopping criterion is met. though EAs are interpretable they do not scale well due to the combinatorial nature of the method. Alternatively, a NN approach can be used by including biases to improve the interpretability of the network. Neural Arithmetic Logic Modules (NALMs) have specially designed architectures biased towards learning arithmetic operations (Mistry et al., 2021). (Sahoo et al., 2018) sets activation functions in a layer to different symbolic operations rather than using a traditional non-linear activation like ReLU. Udrescu & Tegmark (2020b) exploits patterns in the data by designing physics related biases such as translational symmetry or multiplicative separability into their architecture.

This work focuses on modelling division using NALMs; NN which learn arithmetic operations and input selection. The weights of NALMs are intepretable such that a discrete value represents a specific operation. For example, '-1' to represent division and '0' for no selection. Trask et al. (2018) developed the Neural Arithmetic Logic Unit (NALU), the first NALM, which can model all four arithmetic operations. However, studies show the NALU is unstable in learning division (Schlör et al., 2020; Heim et al., 2020). In particular, their gating method responsible for selecting an operation cannot learn consistently (Madsen & Johansen, 2020). To improve the NALU, Schlör et al. (2020) developed iNALU which applies weight and gradient clipping, sign retrieval, regularisation, reinitialisation and separating shared parameters. Even with these modifications, they find consistently learning division to a high precision to remain unattainable. Heim et al. (2020) develops a module which learns in the real and complex parameter space, with results showing their Real NPU to outperform the iNALU for division. Madsen & Johansen (2020) create the Neural Multiplication Unit (NMU) which only models multiplication and has significant performance gains compared to the NALU.

Of these NALMs, we focus on the Real NPU and the NMU. Until now, the Real NPU only has learned division on training ranges of either $\mathcal{U}[0.1,2]$ or Sobol(0,0.5) (Heim et al., 2020). It remains unclear if this module is robust to other training ranges even as a stand-alone unit. Robustness to training ranges is important as these module's applicational use comes from being part of larger end-to-end networks, where the input range into the module cannot be controlled. The NMU is a multiplication module which we extend to do division. The authors of the NMU hypothesised that such an extension incurs too many limitations for learning (Madsen & Johansen, 2020). We use this paper as an opportunity to explore this hypothesis.

3 Architectures

This section introduces the architectures for the (Real) NPU, NRU, and the NMRU. The (Real) NPU is an existing module, which we improve in Section 5. The NRU and NMRU are novel contributions which extend the existing NMU (see Appendix B) to do division.

3.1 Real Neural Power Unit

Heim et al. (2020) develop a module to multiply and divide using the intuition from Trask et al. (2018) that *multiplicative operations are additive operations in log space*. Their work extends this idea into complex space. The NPU can be used with its complex form (Equation 1) requiring both a complex and real weight matrix (i.e., $\boldsymbol{W}^{\mathrm{IM}}, \boldsymbol{W}^{\mathrm{RE}}$), or only its real form the Real NPU (Equation 2). The $\odot$ represents element-wise multiplication (Hadamard/Schur) product. For improved gradients, a relevance gate $\boldsymbol{r}$ (Equation 3) converts inputs close to 0 (i.e. irrelevant features) to 1 to avoid the resulting output evaluating to 0. A gating vector $\boldsymbol{g}$, learns to select relevant input elements, where gate values are clipped between [0,1] during training.

$$\mathrm{NPU} := \exp(\boldsymbol{W}^{\mathrm{RE}} \log(\boldsymbol{r}) - \boldsymbol{W}^{\mathrm{IM}} \boldsymbol{k}) \odot \cos(\boldsymbol{W}^{\mathrm{IM}} \log(\boldsymbol{r}) + \boldsymbol{W}^{\mathrm{RE}} \boldsymbol{k}), \tag{1}$$

$$\text{Real NPU} := \exp(\boldsymbol{W}^{\mathrm{RE}} \log(\boldsymbol{r})) \odot \cos(\boldsymbol{W}^{\mathrm{RE}} \boldsymbol{k}) \tag{2}$$

$$\text{where} \quad \boldsymbol{r} = \boldsymbol{g} \odot (|\boldsymbol{x}| + \epsilon) + (\boldsymbol{1} - \boldsymbol{g}) \quad \text{and} \quad k_i = \begin{cases} 0 & x_i \geq 0 \\ \pi \mathrm{g_i} & x_i < 0 \end{cases}. \tag{3}$$

A weighted L1 penalty is used when training. The weight value β grows between predefined values β_{start} to β_{end} and is increased every $\beta_{step} = 10,000$ iterations by a growth factor $\beta_{growth} = 10$. In our experiments, we focus on the Real NPU over the NPU as the solution of the tasks in this paper can be captured using only real values meaning that the complex form is not required.

3.2 Neural Reciprocal Unit

We propose the NRU, which can model multiplication and division. We extend the NMU motivated by the fact that *division is the multiplication of reciprocals*. The range which weight values can be 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

$$\mathrm{NRU} : z_o = \prod_{i=1}^{I} \left(\mathrm{sign}(\mathrm{x}_i) \cdot |\mathrm{x}_i|^{W_{i,o}} \cdot |W_{i,o}| + 1 - |W_{i,o}| \right), \tag{4}$$

where I is the number of inputs. Assuming weights are either 1 (multiply) or -1 (reciprocal), $|\mathrm{x}_i|^{W_{i,o}}$ 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 4 can contain points of discontinuity. To alleviate this issue, we approximate the absolute function using a scaled tanh (inspired by Faber & 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) controls how close to the absolute function the approximation is, where larger values give a more accurate approximation. For clipping and regularisation, the same scheme as the Neural Addition Unit (NAU) (see Appendix B) is used which forces weight elements to converge to -1, 0 or 1.

3.3 Neural Multiplicative Reciprocal Unit

An alternate extension of the NMU, also motivated by *division being multiplication of reciprocals* is the NMRU (Equation 5). We concatenate the reciprocal of the input (plus a small ϵ) to the input resulting in a module which only needs to learn selection. Hence, weights can be in the range [0,1].

$$\mathrm{NMRU} : z_o = \prod_{i=1}^{2I} (W_{i,o} \cdot |\mathrm{x}_i| + 1 - W_{i,o}) \cdot \cos(\sum_{i=1}^{2I} (W_{i,o} \cdot k_i)) \text{ , where } k_i = \begin{cases} 0 & x_i \geq 0 \\ \pi & x_i < 0 \end{cases}. \tag{5}$$

Table 1: Interpolation (train/validation) and extrapolation (test) ranges used. Data (as floats) is drawn from a Uniform distribution with the range values as the lower and upper bounds.

Interpolation	[-20, -10)	[-2, -1)	[-1.2, -1.1)	[-0.2, -0.1)	[-2, 2)
Extrapolation	[-40, -20)	[-6, -2)	[-6.1, -1.2)	[-2, -0.2)	[[-6, -2), [2, 6)]
Interpolation	[0.1, 0.2)	[1, 2)	[1.1, 1.2)	[10, 20)	
Extrapolation	[0.2, 2)	[2, 6)	[1.2, 6)	[20, 40)	

The iteration over $2I$ represents the going through all inputs and their reciprocals. We calculate the magnitude and sign separately, joining the result at the end. The magnitude is calculated passing absolute of the concatenated input through an NMU architecture and the sign by using a cosine mechanism similar to the Real NPU. However, unlike the Real NPU only the weight matrix is required. The norm of the weight's gradients are 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 Appendix B) is used.

4 Experiment Setup

We introduce the two main experiments used to evaluate modules, including: default parameters, train and test ranges, and evaluation metrics. The tasks evaluate the ability of a single module to divide two numbers from an input vector in two settings: **no redundancy** (2 inputs) and **with redundancy** (10 inputs).

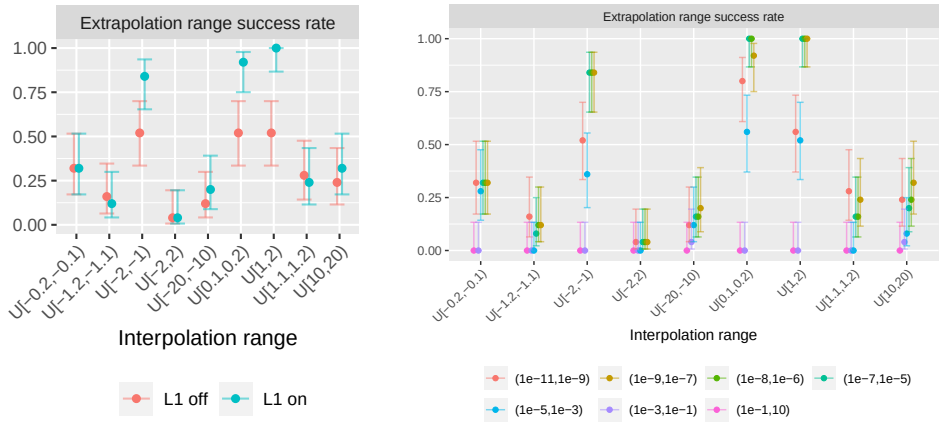
Default parameters: All experiments use a mean squared error (MSE) loss with an Adam optimiser (Kingma & Ba, 2015), with 10,000 samples for the validation and test sets. The best model for evaluation is taken using early stopping on the validation set. All runs are over 25 different seeds. All inputs are required in the *no redundancy* setting, i.e., input size of 2. Training takes 50,000 iterations where each iteration consists of a different batch of size 128. The Real NPU uses a learning rate of 5e-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 1e-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 with batch sizes of 128. The learning rates for the Real NPU, NRU and NMRU are 5e-3, 1e-3 and 1e-2 respectively. Sparsity regularisation scaling occurs during iteration 50,000 to 75,000 for all modules. A summary of all relevant parameters is found in Appendix C.

Ranges: The interpolation (train/validation) and extrapolation (test) ranges, are found in Table 1. The chosen ranges are influenced by Madsen & Johansen (2020). In particular, the interpolation ranges will not overlap with the extrapolation ranges.

Evaluation metrics: We use the Madsen & Johansen (2019)'s evaluation scheme, consisting of three evaluation metrics: the success on the extrapolation dataset against a near optimal solution (*success rate*), the first iteration which the task is considered solved (*speed of convergence*), and the extent of discretisation towards the weights' inductive biases (*sparsity error*). Sparsity error calculated by

$$\max_{i,o}(\min(|W_{i,o}|, 1 - |W_{i,o}|)) \ ,$$

measures the weight element which is the furthest away from the acceptable discrete weights for the module. A success means the MSE of the trained model is lower than a threshold value (i.e. the MSE of a near optimal solution). We differ from Madsen & Johansen (2019) by using a fixed threshold value 1e-5 rather than a simulated MSE. We choose this precision as it can be guaranteed when working with 32-bit PyTorch Tensors. 95% confidence intervals (over the 25 seeds) are calculated from a specific family of distributions dependant on the metric. The success rate uses 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



(a) L1 regularisation

(b) Sweep over L1 (start,end) beta parameters

Figure 1: Exploring the effect and sensitivity of L1 regularisation on the Real NPU

uses a Beta distribution. Both Beta and Gamma can easily approximate the normal distribution and support its corresponding metric.

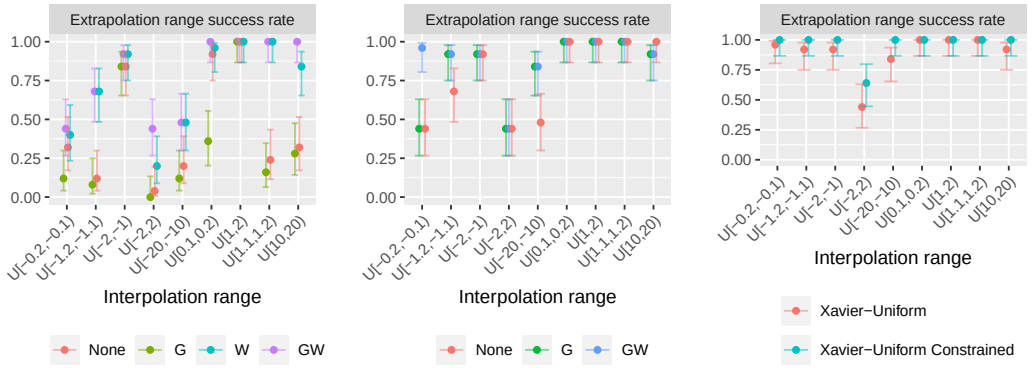
5 Improving the Real NPU's Robustness

We first improve the robustness of the Real NPU on different training ranges. We use the Single Module Task with no redundancy (see Section 4) to investigate the following questions:

1. Is L1 regularisation required, and if so, do the regularisation parameters require tuning?
2. Does clipping the weight matrix aid learning?
3. Does enforcing discretisation on parameters improve convergence?
4. Can the weight matrix initialisation be improved?

To address each question in order, we propose applying incremental modifications to the Real NPU. These modifications include: ablation study on the L1 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 suggested can also be generalised for the NPU architecture.

Is L1 regularisation required? (Yes.) L1 encourages sparsity (i.e., zero weights) in solutions. Zero-valued weights means not to select an input and return the identity value 1. For the task, the optimal weight values require selecting all inputs and therefore non-zero values, suggesting the application of L1 could be damaging. Therefore, we compare against a model which does not use L1 regularisation, shown in Figure 1a. Removing L1 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 L1 regularisation.[3] The L1 regularisation scaling (see Section 3.1), requires setting the hyperparameters for the start (β_{start}) and end (β_{end}) scaling values. We run a sweep over six different start and end values, denoted (<start>, <end>), displaying results in Figure 1b. We find the configuration (1e-9, 1e-7) is the most successful when considering performance on all the ranges, and larger scaling values perform worse.



(a) Clipping (b) Discretisation regularisation (c) W^{RE} initialisation schemes

Figure 2: Effect of clipping, discretisation, and the NAU initialisation scheme on the Real NPU.

Does clipping the learnable parameters help? (Yes.) Division and multiplication operations 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. The gate weights do get clipped and saved to another variable during the forward pass, meaning after an update step the gate values can also be out of the range [-1,1]. Hence, we investigate the effect of applying clipping directly to the weight and gate values after every optimisation step. Results, shown in Figure 2a, show clipping is beneficial, with clipping on both weight and gate (or just on the weights) to improve over the baseline on all ranges (excluding $\mathcal{U}$[1,2) where the baseline has already achieved full success).

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 & Johansen (2020)'s regularisation scaling scheme, we penalise weights for not being discrete. 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. Results, shown in Figure 2b, show discretising the gate improves over the baseline but also discretising the weights is additionally beneficial (especially for range $\mathcal{U}$[-0.2,-0.1)). $\mathcal{U}$[10,20) is the only range where the baseline outperforms using discretisation, succeeding on two additional seeds.

Does using a more constrained initialisation help? (Yes.) $\boldsymbol{W}^{\mathrm{RE}}$ uses a Xavier-Uniform initialisation (Glorot & Bengio, 2010). This can result in weights initialised out of the range [-1,1]. Therefore, we use the initialisation for the Neural Addition Unit 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] (Madsen & Johansen, 2020). Figure 2c shows using the constrained initialisation provides improvements over multiple ranges.

6 Results: Single Module Task

We analyse the results for the: Real NPU without using the modifications of Section 5, Real NPU with modifications, NRU, and NMRU on the 2 input and 10 input tasks.

6.1 No Redundancy

First, we investigate whether division without irrelevant features can be learnt, finding the Real NPU cannot while the NRU and NMRU can. Figure 3 shows the baseline Real NPU without modifications struggles with all ranges except $\mathcal{U}$[1,2), struggling with sparsity on the larger ranges. Applying the modifications deals with the sparsity issue and improves the robustness such that only range $\mathcal{U}$[-2,2) struggles (with a success rate of

[3]We also experimented with using L2 regularisation but found L1 to perform better. Results are found in Appendix G.

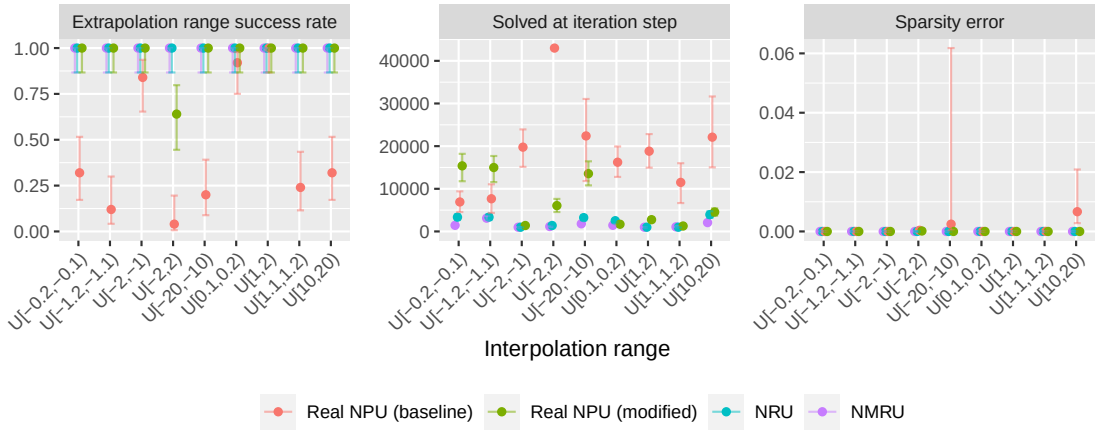


Figure 3: Division without redundancy (input size 2).

0.64). The NRU and NMRU achieve full success over all ranges while solving the problem consistently fast and with low sparsity error. The success of the NRU is correlated with the learning rate (see Appendix E).

6.1.1 Mixed-signed Inputs

The remaining failure range of the Real NPU (modified) is $\mathcal{U}$[-2,2) where inputs can consist of arbitrary signed values (e.g. all positives, all negatives, or a mixture of positive and negative values). *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).* We create five additional mixed-sign datasets, controlling the range for each element in the input. The interpolation and extrapolation ranges can be found in Appendix C. Datasets 1, 2, 4 and 5 sample 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.

As shown by Figure 4, the Real NPU struggles on all these ranges, implying that the core issue is not from different input samples having different signs or due to input values being close to 0. 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). When the signs for an input element are controlled, discretisation/sparsity is no problem, in contrast when the signs are arbitrary the sparsity error are slightly (though not significantly) higher.

6.1.2 Division by Small Numbers

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) diving the first input by the second given two inputs.

Figure 5 plots the test error for different modules 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 become increasingly deceptive with values close to zero. The Real NPU has larger test errors for all tasks and ranges, caused by adding ϵ to the input (see Equation 3). Setting $\epsilon = 0$ reduces the test error at the cost of the ability to deal with zero-valued inputs. Appendix F provides the corresponding experimental results finding that only modelling reciprocals can be learnt with extremely small values.

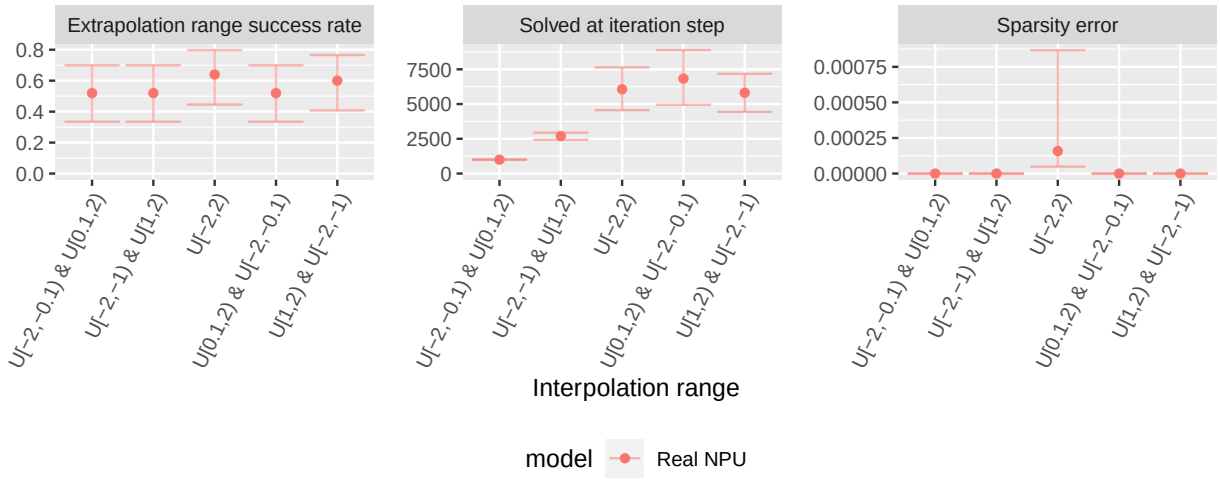


Figure 4: Extrapolation results on training the Real NPU using mixed-sign datasets that control the sign of the input elements. The ranges are in order of the datasets (i.e. dataset 1 to 5).

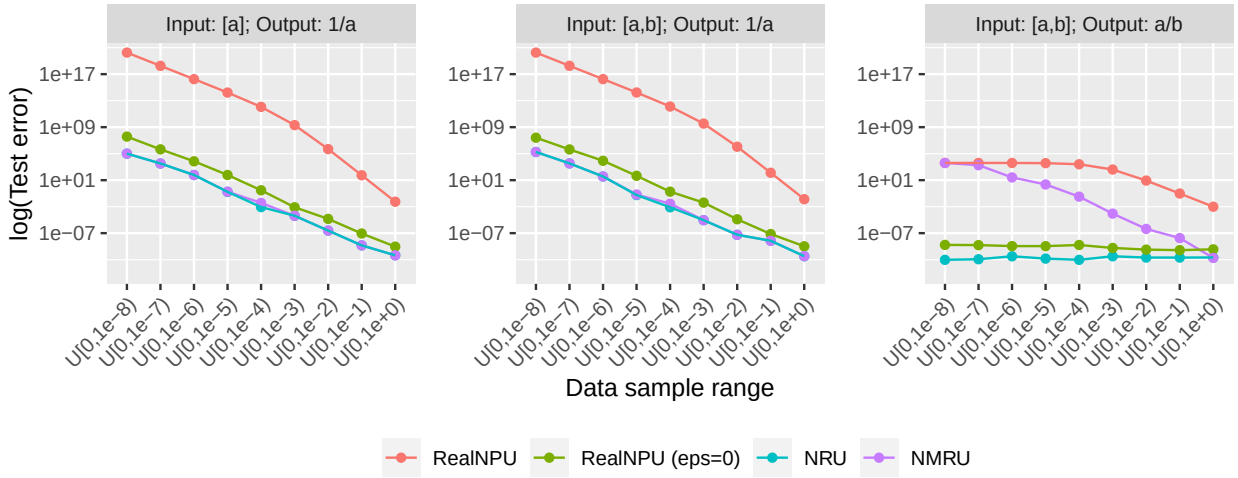


Figure 5: Effect of the singularity issue on the Real NPU, NRU and NMRU over increasing input ranges. Left: Reciprocal for an input size of 1 (no redundancy). Middle: Reciprocal for an input size of 2 (with redundancy). Right: Division for an input size of 2 (no redundancy).

6.2 With Redundancy

We now investigate the effect of irrelevant input features using the 10 input task. Introducing redundancy (Figure 6) causes multiple failure modes to arise on all modules. The baseline Real NPU produces high sparsity errors relative to the other modules suggesting struggle with discretisation. Using 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.[4] 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 (see Appendix H). Complex weights hinders success and convergence speeds of negative ranges. 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 L1 penalty. This did not result in any significant improvements against the Real NPU results. Input redundancy

[4]Except for the sparsity error for range $\mathcal{U}[10,20)$.

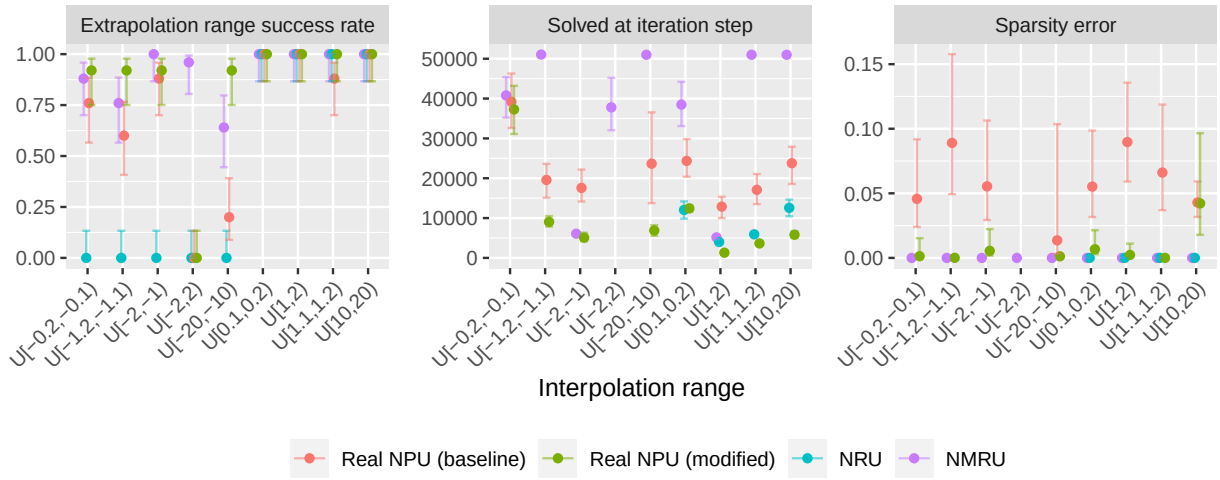


Figure 6: Division with redundancy (input size 10).

effects the NRU the most, resulting in full failures on all the negative ranges. The NMRU is the only module with success for the range $\mathcal{U}$[-2,2), which is a result of using the sign mechanism (see Appendix I). It performs well over all ranges but can be outperformed by the modified Real NPU for negative ranges. Multiple ranges for the NMRU are solved around 50,000 iterations correlating to the sparsity regularisation being turned on.

6.2.1 Gradient Difficulties with the NRU

For insight to why the NRU performs poorly, the partial derivative for the NRU weights are derived in Equation 6.

$$\begin{aligned}\frac{\partial \hat{y}}{\partial w_i} = \tanh(1000w_i)(\text{sign}(x_i)|x_i|(\tanh(1000w_i)\log(|x_i|)+ \\ 2000\,\text{sech}(1000w_i)^2) - 2000\,\text{sech}(1000w_i)^2) \times \text{NRU}_{\tilde{\mathbf{x}}\in\mathbf{x}\setminus\{\mathbf{x_i}\}}(\tilde{\mathbf{x}}).\end{aligned} \tag{6}$$

The $\text{NRU}_{\tilde{\mathbf{x}}\in\mathbf{x}\setminus\{\mathbf{x_i}\}}(\tilde{\mathbf{x}})$ 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. If $x_i \approx 0$ and $-1 \leq w_i < 0$ then as $w_i \to -1$ all gradients for x_i where $|x_i| >> 1$ become increasingly small. The gradients for $x_i = -1$ and $x_i = 1$ are 0 regardless 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. Even if the sign and magnitude are calculated separately and then combined (see Appendix J) 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.

6.2.2 The Real NPU's and NMRU's Exploitation of Multiplicative Rules

Modules with extrapolative solutions learn to exploit arithmetic 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 which contains 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, we can avoid this by allowing 0 to not be penalised during sparsity regularisation stage (see Appendix H); this alleviates the exploitation issue with no cost to performance.

Table 2: Interpolation (train/validation) and extrapolation (test) ranges used. Data (as floats) is drawn with the range values as the lower and upper bounds. TN = Truncated Normal distribution in the form TN(mean, sd)[lower bound, upper bound). B = Benford distribution. U = Uniform distribution.

Interpolation	TN(-1, 3)[-5, 10)	TN(0,1)[-5, 5)	TN(1, 3)[-10, 5)
Extrapolation	TN(-10, 3)[-15, -5)	TN(10,1)[5, 15)	TN(10, 3)[5, 15)
Interpolation	B[10, 100)	U[-100, 100)	U[-50, 50)
Extrapolation	B[100, 1000)	U[-200, -100) ∪ [100, 200)]	U[[-100, -50) ∪ [50, 100)]

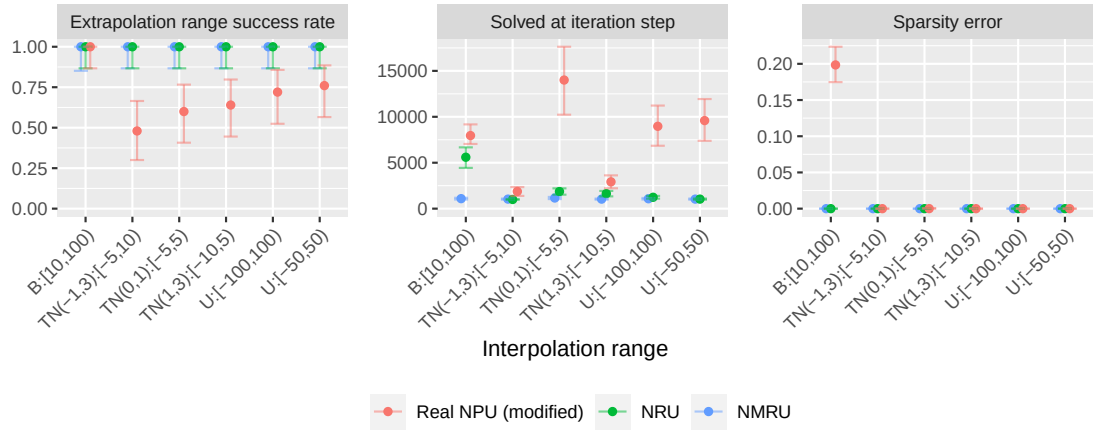


Figure 7: Division without redundancy (input size 2) on the Benford, Truncated Normal and Uniform distribution.

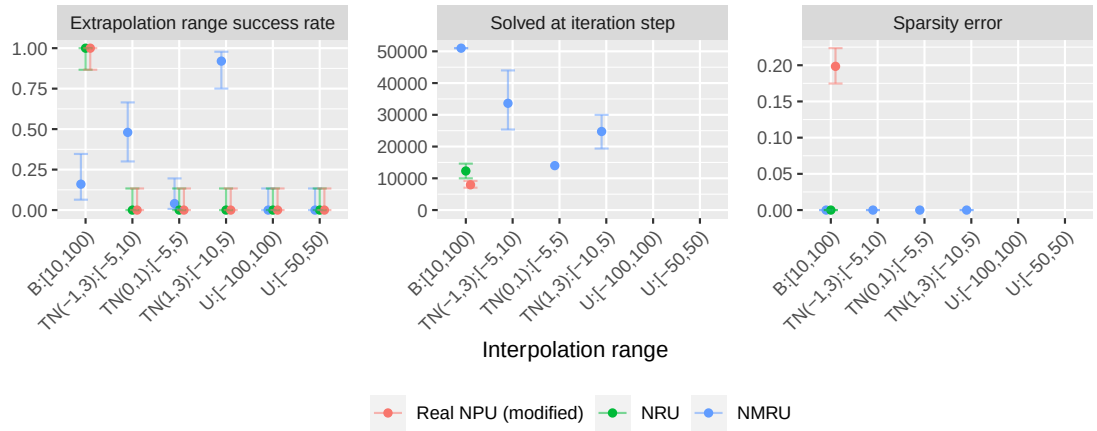


Figure 8: Division with redundancy (input size 10) on the Benford, Truncated Normal and Uniform distribution.

7 More Challenging Distributions

To discover more failure cases, we explore additional distributions (i.e., Benford and Truncated Normal) and larger ranges around zero (Uniform distribution) defined in Table 2. Each distribution is tested on both the 2 input no redundancy (Figure 7) and 10 input redundancy (Figure 8) setting.

Table 3: Summary of the types division tasks the models can solve. Redundancy means there are irrelevant inputs. A tick means the model can learn the task configuration and with a success rate is over 0.75 (if there are multiple ranges then the average success rate is used).

Redundancy?	Input type	Distribution	Real NPU (modified)	NRU	NMRU	Figure
No	Mixed-signs	Uniform		✓	✓	3; see U[-2,2)
	Mixed-signs	Truncated Normal		✓	✓	7
	Negative	Uniform	✓	✓	✓	3
	Negative	Truncated Normal		✓	✓	7
	Large magnitude	Uniform		✓	✓	7
	Large magnitude	Benford	✓	✓	✓	7
	Close to 0	Uniform				13
	Close to 0	Truncated Normal		✓	✓	8; see TN(0,1)[-5, 5)
Yes	Mixed-signs	Uniform			✓	6; see U[-2,2]
	Mixed-signs	Truncated Normal				8
	Negative	Uniform	✓		✓	6
	Negative	Truncated Normal				8
	Large magnitude	Uniform				8
	Large magnitude	Benford	✓	✓		8
	Close to 0	Truncated Normal				8; see (TN(0,1)[-5, 5))

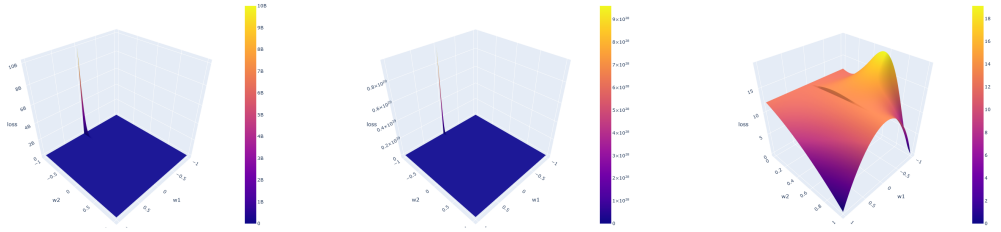
Uniform distributions: All modules find the larger ranges to be challenging when redundant inputs exist. On the 2-input setup, both NRU and NMRU have full success, while the Real NPU (modified) has failure cases for both Uniform distributions (with success rates of 0.72 on $\mathcal{U}$[-100,100) and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

Benford distribution: This follows a more natural distribution compared to the Uniform distribution, known to underlie real world data such as accounting data (Hill, 1995). A large range is sampled, showing full success for the NRU and modified Real NPU on the 10 input setting. This implies the underlying issue from the failures of the Uniform distributions are attributed to the used of mixed signed inputs rather than the large ranges. However, the NMRU shows majority failures (failure rate 0.84) suggesting that large ranges are also an area of struggle for the module.

Truncated Normal distribution: We discover further failure cases (especially the Real NPU (modified)). When trained using the 2-input setup, both NRU and SignNMRU have full success but the Real NPU (modified) has failure cases for all three distributions (with success rates 0.48, 0.6, 0.64 respectively). When trained using the 10-input setup, both the NRU and Real NPU (modified) have no success in any range. 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 that the NMRU works better when a majority of the inputs are likely to have the same sign and struggles with values around zero.

8 Discussion

In this paper, we demonstrate the limitations of intepretable neural networks in learning to divide, summarising the key challenges in Table 3. Using the no redundancy setting (size 2), we find that the Real NPU is challenged when 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) magnifies this issue, but also introduces failure modes for the NRU and NMRU for negative ranges. The NRU is unable to handle any negative ranges, in which we conclude it is not wise to use with MSE. Alternate losses can improve certain failure cases though sometimes at the cost of performance on other ranges. For further



(a) Real NPU (b) NRU (c) NMRU

Figure 9: Root Mean Squared loss curvature for the NAU stacked with either a RealNPU, NRU, or NMRU. "The weight matrices are constrained to $\mathbf{W}_1 = \left[\begin{smallmatrix} w_1 & w_1 & 0 & 0 \\ w_1 & w_1 & w_1 & w_1 \end{smallmatrix}\right]$, $\mathbf{W}_2 = [\, w_2 \; w_2 \,]$. 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 & 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 $\mathbf{g} = [\, 1 \; 1 \,]$.)

details see Appendix K which displays results on a correlation and scale-invariant based loss. Learning to divide values around zero remains challenging and from training on different distributions we find all modules struggle with the large Uniform distribution and Truncated Normal distributions in the redundancy setting.

Our NMRU is the only module with reasonable success over all tested Uniform distributions in Section 6, but is challenged by extremely large ranges (see Figure 8). The NMRU requires only $2I \times O$ learnable parameters, however this 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. 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. Previously, Madsen & 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. In Figure 9, we recreate their setup to produce the loss surfaces for the NAU-Real NPU[5], NAU-NRU and NAU-NMRU respectively.[6] We find a similar issue with the Real-NPU and NRU, as both these units use a weight range of [-1,1]. In contrast, the NMRU, whose weight's range is limited to [0,1] does not have exploding losses.

In conclusion, division remains a challenge to learn using intepretable 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.

References

Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014. ISSN 0045-7906. doi: https://doi.org/10.1016/j.compeleceng.2013.11.024. URL `https://www.sciencedirect.com/science/article/pii/S0045790613003066`. 40th-year commemorative issue.

Lukas Faber and Roger Wattenhofer. Neural status registers. *CoRR*, abs/2004.07085, 2020. URL `https://arxiv.org/abs/2004.07085`.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010. URL `http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf`.

[5]The NAU is a summative module (Madsen & Johansen, 2020).
[6]Appendix L displays larger versions of these plots.

Niklas Heim, Tomáš Pevnỳ, and Václav Šmídl. Neural power units. *Advances in Neural Information Processing Systems*, 33, 2020. URL https://papers.nips.cc/paper/2020/file/48e59000d7dfcf6c1d96ce4a603ed738-Paper.pdf.

Theodore P Hill. A statistical derivation of the significant-digit law. *Statistical science*, pp. 354–363, 1995.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015. URL https://arxiv.org/pdf/1412.6980.pdf.

Andreas Madsen and Alexander Rosenberg Johansen. Measuring arithmetic extrapolation performance. In *Science meets Engineering of Deep Learning at 33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, volume abs/1910.01888, Vancouver, Canada, October 2019. URL https://arxiv.org/pdf/1910.01888.pdf.

Andreas Madsen and Alexander Rosenberg Johansen. Neural arithmetic units. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1gNOeHKPS.

Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. A primer for neural arithmetic logic modules, 2021. URL https://arxiv.org/pdf/2101.09530.pdf.

Subham Sahoo, Christoph Lampert, and Georg Martius. Learning equations for extrapolation and control. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4442–4450. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/sahoo18a.html.

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. URL https://www.frontiersin.org/article/10.3389/frai.2020.00071.

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*, pp. 8035–8044, 2018. URL https://openreview.net/pdf?id=H1gNOeHKPS.

Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16), 2020a. doi: 10.1126/sciadv.aay2631. URL https://advances.sciencemag.org/content/6/16/eaay2631.

Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020b. URL https://www.science.org/doi/full/10.1126/sciadv.aay2631.

A 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.

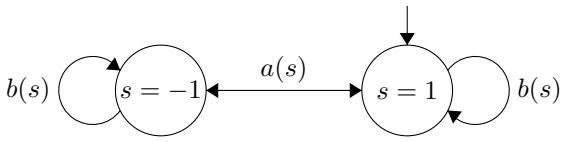
Intepretable weights: A good division module should produce generalisable solutions to out-of-bounds data. Using interpretable weights to represent exact operations is one way of doing so, e.g., -1 to divide, 1 to multiply, 0 to not select. For the scope of this paper we focus on discrete weights, however 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 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 for 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 17, 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 ≥ 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 ≥ 0 or the weight value to not select the input.

Selection: Not all inputs are relevant for 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 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 I).

B Neural Addition and Neural Multiplication Units' (NAU & NMU)

Madsen & Johansen (2020) develop two modules: one for dealing with addition and subtraction (the NAU) and the other for multiplication (the NMU). NAU output element a_o is defined as

$$\text{NAU} : a_o = \sum_{i=1}^{I} \left(W_{i,o} \cdot \mathrm{x}_i \right) \tag{7}$$

where I is the number of inputs. The NMU output element m_o is defined as

$$\text{NMU} : m_o = \prod_{i=1}^{I} \left(W_{i,o} \cdot \mathrm{x}_i + 1 - W_{i,o} \right). \tag{8}$$

Before passing a input through a module, the weight matrix is clamped to [-1,1] for the NAU or [0,1] for the NMU. Weights are ideally discrete values, where the NAU is 0, 1, or -1, representing no selection, addition and subtraction, and the NMU is 0 or 1, representing no selection and multiplication. To enforce discretisation of weights both units have a regularisation penalty for a given period of training. The penalty is

$$\lambda \cdot \frac{1}{I \cdot O} \sum_{o=1}^{O} \sum_{i=1}^{I} \min \left(|W_{i,o}|, 1 - |W_{i,o}| \right), \tag{9}$$

where O is the number of outputs and λ is defined as

$$\lambda = \hat{\lambda} \cdot \max \left(\min \left(\frac{iteration_i - \lambda_{start}}{\lambda_{end} - \lambda_{start}}, 1 \right), 0 \right). \tag{10}$$

Regularisation strength is scaled by a predefined $\hat{\lambda}$. The regularisation will grow from 0 to $\hat{\lambda}$ between iterations λ_{start} and λ_{end}, after which it plateaus and remains at $\hat{\lambda}$.

C Experiment Parameters

Tables 4 and 5 for the breakdown of parameters used in the Single Module Tasks. Table 6 gives the interpolation and extrapolation ranges used in the mixed-sign datasets tasks.

Table 4: 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
Layers	1	1
Input size	2	10
Total iterations	50,000	100,000
Train samples	128 per batch	128 per batch
Validation samples*	10000	10000
Test samples*	10000	10000
Seeds	25	25
Optimiser	Adam (with default parameters)	Adam (with default parameters)
$\hat{\lambda}^{\dagger}$	10	10

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

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

Table 6: 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.

	Interpolation		Extrapolation	
Dataset	**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)

C.1 Parameter Initialisation

We give the initialisations used on the different module parameters:

Real NPU: The real weight matrix uses the 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 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 & 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 & Johansen, 2020).)

D Hardware and Time to Run Experiments

All 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 7 displays 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 7: Timings of experiments.

Experiment	Model	Approximate time for completing 1 seed (mm:ss)
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 NRU on the Single Module Task (no redundancy): Effect of Learning Rate

Figure 10 displays the effect of different learning rates for the NRU. An learning rate of 1 gets full success on all ranges with performance deteriorating as the learning rate reduces.

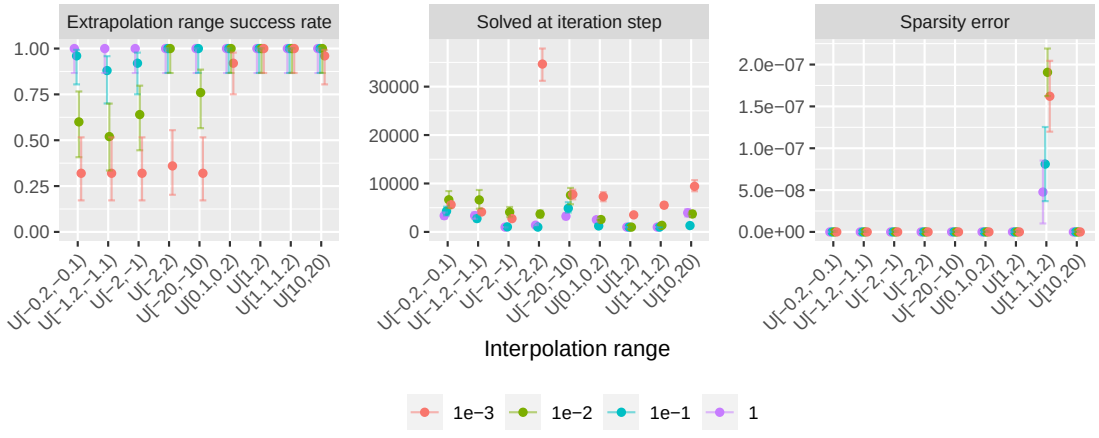


Figure 10: Different learning rates on the NRU for the Single Module Task (no redundancy)

F Division by small values: Experimental Results

This section shows the results on 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: 1e-4, 1e-3, 1e-2, 1e-1 and 1. Unless stated otherwise, the hyperparameters of a model are set to what is used for the Single Layer Task without redundancy. 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 which 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[7] 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 11) is achieved with ease for all modules, though for $\mathcal{U}$[0,1e-4), we find the NRU begins to start struggling.

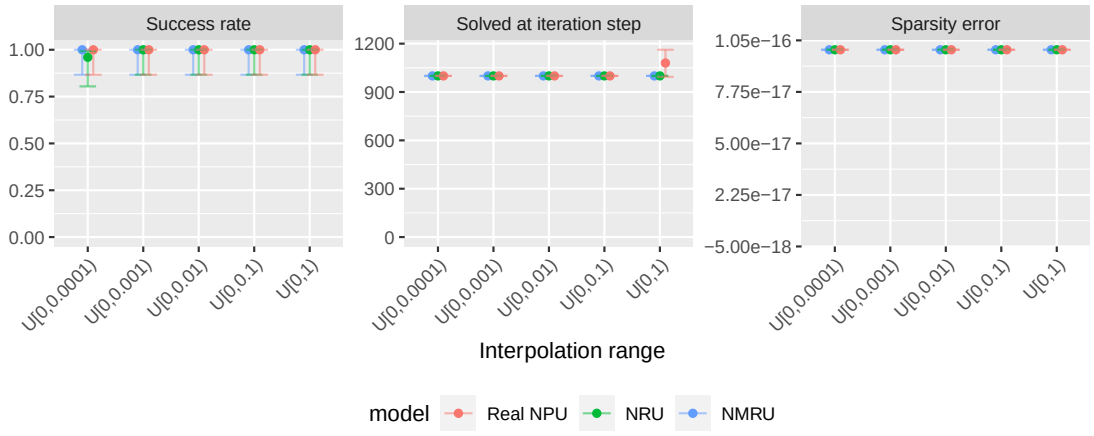


Figure 11: Input: [a], output $\frac{1}{a}$. Learns reciprocal when there is no input redundancy.

Introducing a redundant input (Figure 12) 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, 1e-4) are false positives caused by the ϵ 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 13), meaning the redundant input is now relevant, shows improvement for the NMRU and NRU for the larger ranges.

[7]The term is the pytorch default eps value, torch.finfo().eps

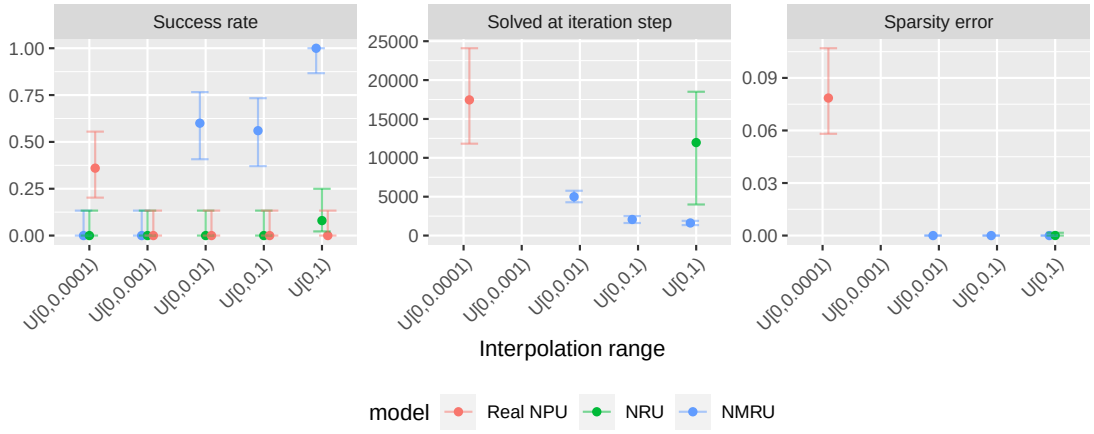


Figure 12: Input: [a,b], output $\frac{1}{a}$. Learns reciprocal of the first input when there is redundancy.

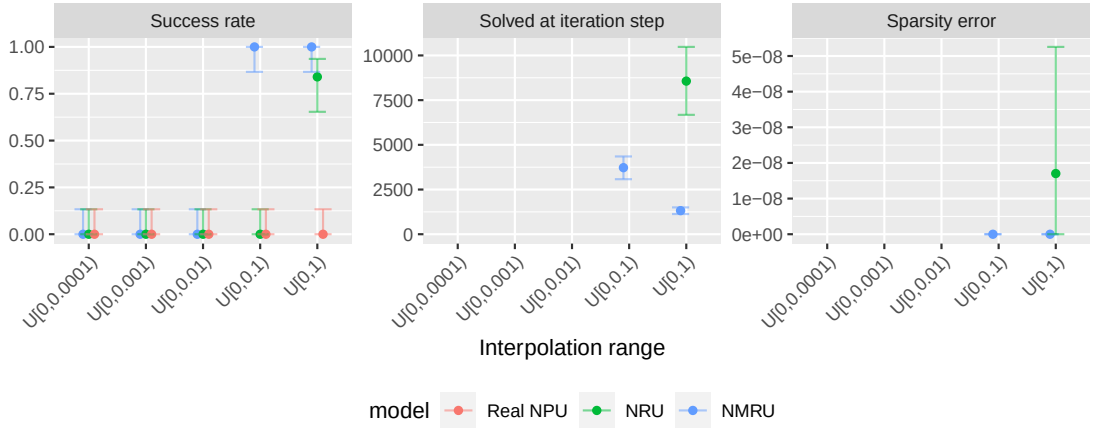


Figure 13: Input: [a,b], output $\frac{a}{b}$. Learns division of the first and second value when there is no redundancy.

G Real NPU; Single Module Task (without Redundancy): Additional Experiments

Figure 14 shows results of using the NPU for the 2-input task. Of the 9 tested ranges, L2 has a lower success rate than L1 for 5 ranges and has the same success rate for the remaining 4 ranges. If L2 regularisation is used instead of no regularisation, it performs worse in 3 (of the 9) ranges, better on 3 ranges and the same on the remaining 3 ranges.

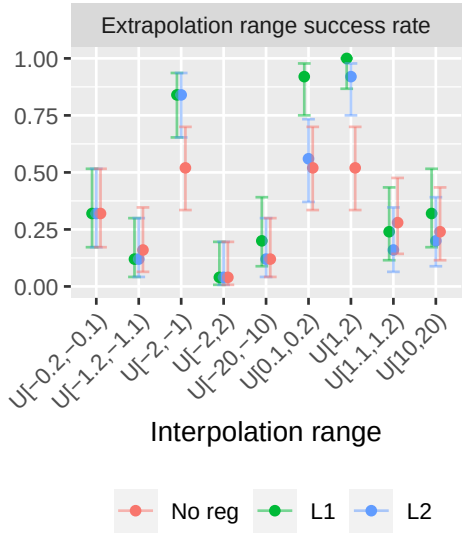


Figure 14: Applying no regularisation, L1 regularisation and L2 regularisation to enforce sparsity in weights.

H Real NPU; Single Module Task (with Redundancy): Additional Experiments

Figure 15 shows results of using the NPU for the task with redundancy.

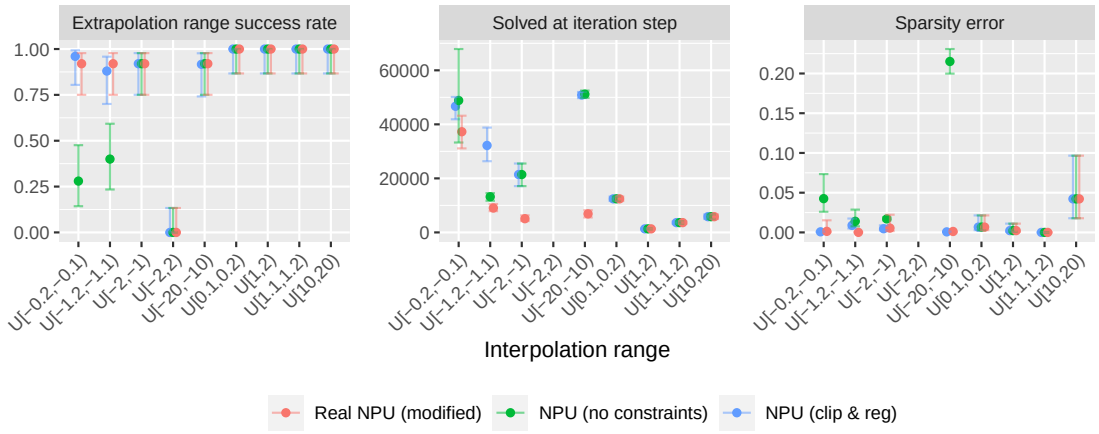


Figure 15: Adapting the Real NPU to use complex weights (NPU) on the Single Module Task with redundancy. Compares the NPU architecture with the Real NPU modifications (i.e. NPU (no constraints)) and the same model but with the imaginary weights clipped to [-1,1] and L1 sparsity regularisation on the complex weights (i.e. NPU (clip & reg)).

Figure 16 shows how modifying the weight discretisation to not penalise weights at 0 does not effect success.

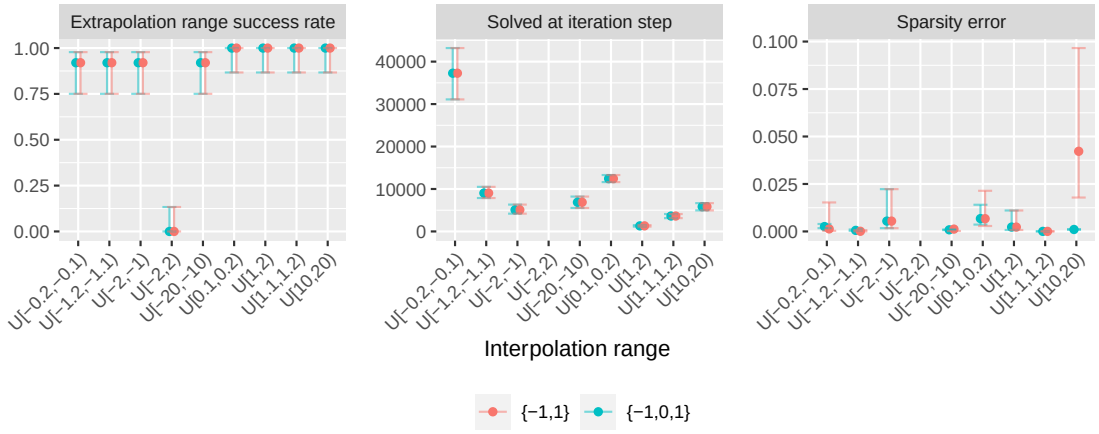


Figure 16: Comparing weight discretisation on the NPU weights which penalises not having weight of $\{-1, 1\}$ vs $\{-1, 0, 1\}$.

I NMRU; Single Module Task with Redundancy (Additional Experiments)

This section further explores the NMRU architecture.

Figure 17 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 in 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 effected. 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 discritisation. Even with the different ablations, the sparsity errors of the successful seeds remain extremely low (which is not always the case for the Real NPU (see Figure 6)).

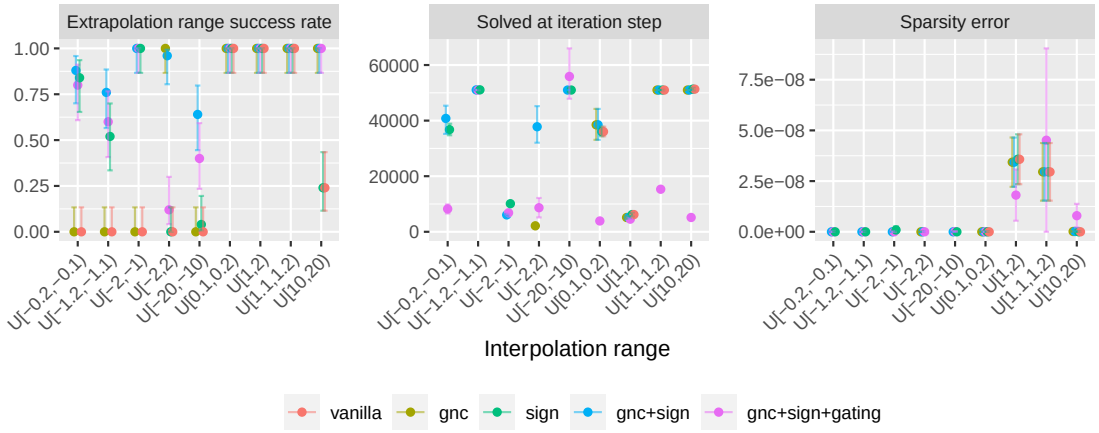


Figure 17: Ablation study for the NMRU.

Figure 18 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.

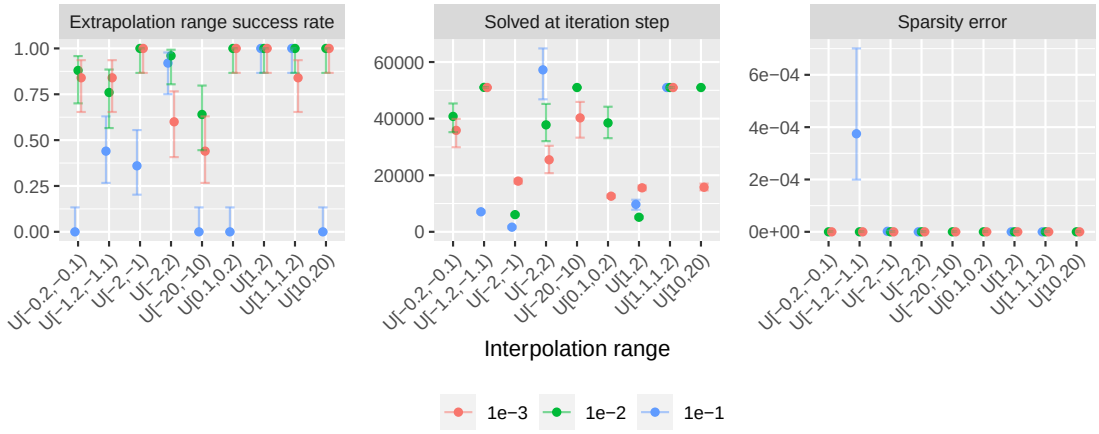


Figure 18: Effect of different learning rates on the NMRU

Figure 19 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 scale the step size of each weight, which can compliment the clipped gradient norm of the NMRU, in contrast to the SGD's global step size.

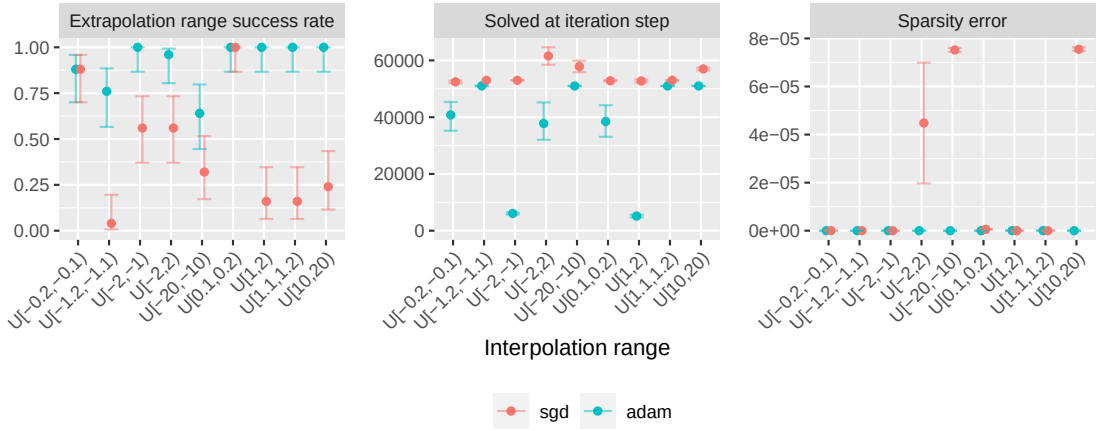


Figure 19: Effect of optimiser on the NMRU. SGD = Stochastic Gradient Descent.

J 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,

$$z_o = \prod_{i=1}^{I} \left(|\mathrm{x}_i|^{W_{i,o}} \cdot |W_{i,o}| + 1 - |W_{i,o}| \right) \cdot \prod_{i=1}^{I} \mathrm{sign}(\mathrm{x}_i)^{\mathrm{round}(W_{i,o})} \,. \tag{11}$$

Figure 20 shows results, where the separate sign method shows no difference in success to the original NRU architecture.

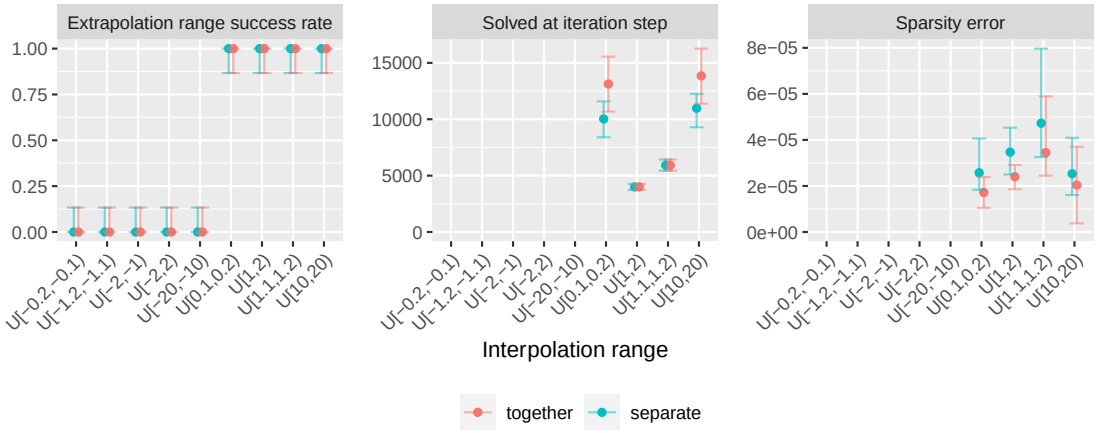


Figure 20: NRU on the redundancy experiment comparing a module which calculates the magnitude and sign together vs calculating the magnitude and sign separately and then combining them.

K Effect of Different Losses on the Single Module Task (with Redundancy)

Table 8: The properties of different loss functions.

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

Different losses induce different loss landscapes impacting the areas of success for a module. We explore the effects of three different losses including the MSE, Pearson's Correlation Coefficient (Equation 13), and the Mean Absolute Precision Error (Equation 14). We use the division task with 10 inputs. The properties of each loss is summarised in Table 8. All experiment parameters match the original MSE runs in the main experiments. The only difference is the loss used.

$$\begin{aligned} v_{x,i} &= (\hat{y}_i - \bar{\hat{y}}), \quad s_x = \sqrt{\text{clamp}(\frac{1}{N}\sum_i^N v_{x,i}^2,\, \epsilon)} \\ v_{y,i} &= (y_i - \bar{y}), \quad s_y = \sqrt{\text{clamp}(\frac{1}{N}\sum_i^N v_{y,i}^2,\, \epsilon)} \\ r &= \frac{1}{N}\sum_i^N (\frac{v_{x,i}}{s_x + \epsilon} \cdot \frac{v_{y,i}}{s_y + \epsilon}) \end{aligned} \tag{12}$$

$$\text{pcc loss} := 1 - r \tag{13}$$

where N is the batch size, and the means ($\bar{\hat{y}}$ and $\bar{y}$) are taken over the batch. ϵ is used to provide better numerical stability.

$$\text{mape loss} := \frac{1}{N}\sum_i^N (\frac{|y_i - \hat{y}_i|}{y_i}) \tag{14}$$

Real NPU (Figure 21) Both the Real NPU 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, in order to gain successes in the mixed-sign range, the other negative ranges have reduced in 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.

NRU (Figure 22) Different losses have little effect on the NRU. All three losses perform well on the positive ranges. Compared to the Real NPU, the PCC loss on the NRU takes longer to converge to a success for negative ranges.

NMRU (Figure 23) All three loses perform reasonably well, with the PCC struggling the most. Unlike the other units, $\mathcal{U}$[-20,-10) causes the most trouble, whereas $\mathcal{U}$[-2,2) gains near to full success on two of the three losses.

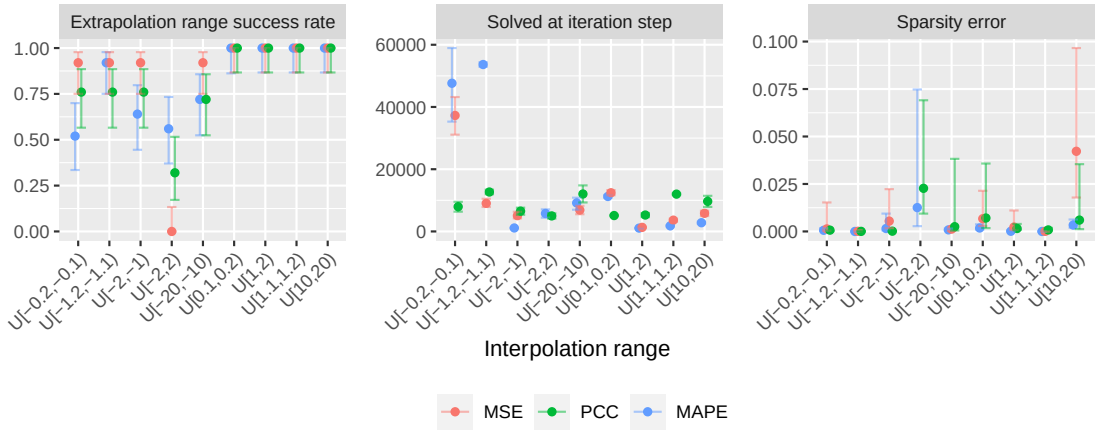


Figure 21: Single Module Task with redundancy on the Real NPU, comparing different loss functions.

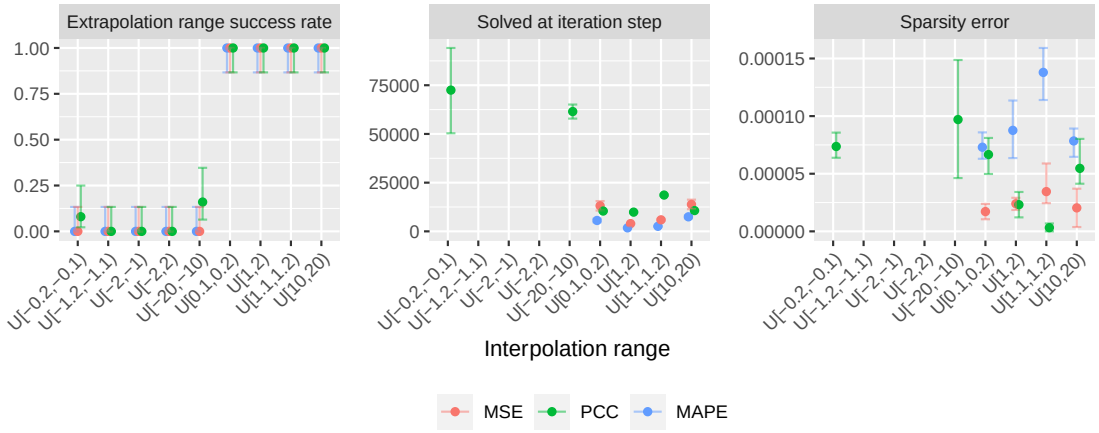


Figure 22: Single Module Task with redundancy on the NRU, comparing different loss functions.

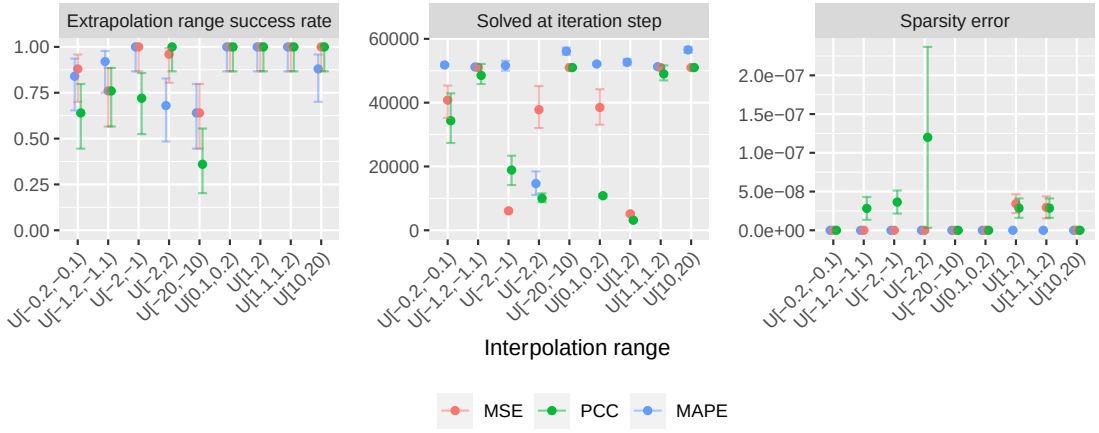
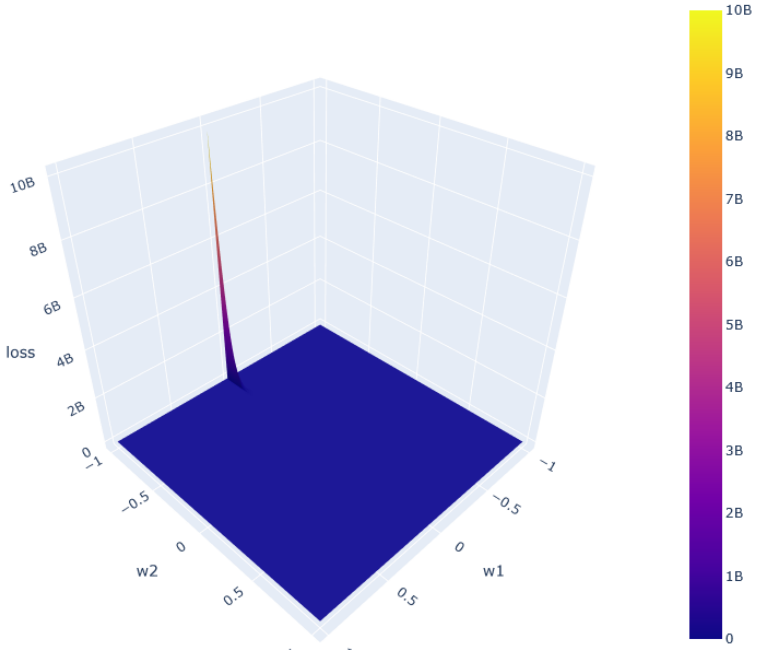


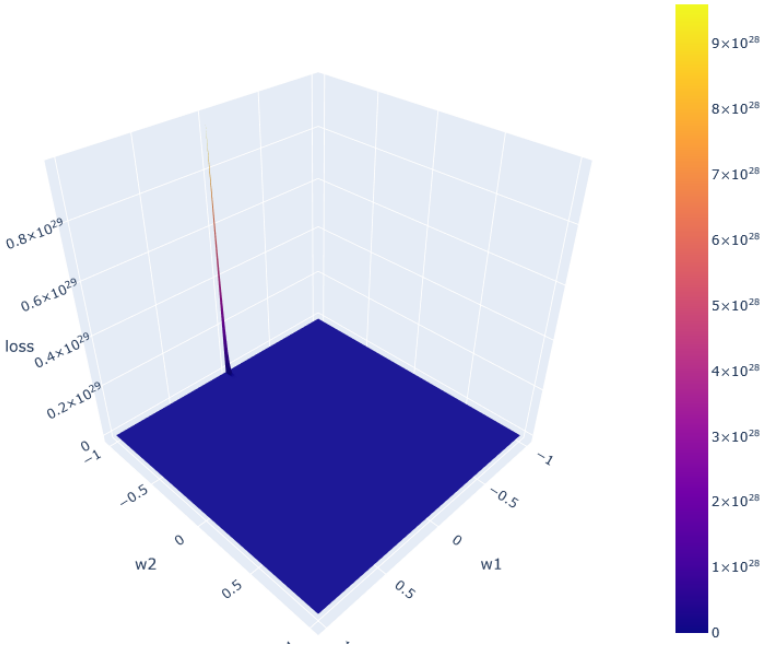
Figure 23: Single Module Task with redundancy on the NMRU, comparing different loss functions.

L RMSE Loss Landscapes

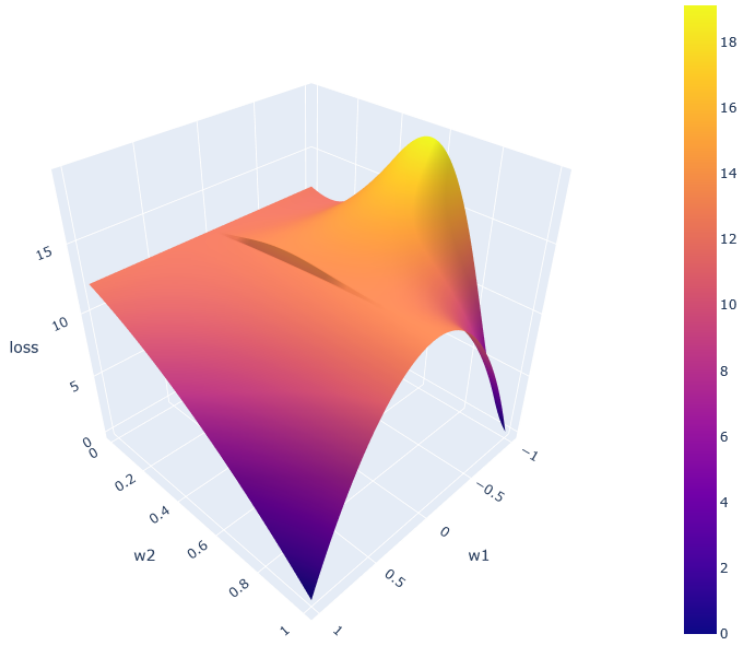
For clarity, we show bigger versions of each subplot from Figure 9.



(a) NAU-Real NPU (where $\epsilon = 1e - 5$)



(b) NAU-NRU



(c) NAU-NMRU

Figure 24: Enlarged loss landscapes of different stacked summative-multiplicative units.