Improving the Robustness of Neural Multiplication Units with Reversible Stochasticity

Bhumika Mistry & Katayoun Farrahi & Jonathon Hare
Department of Vision Learning, and Control
Electronics and Computer Science
University of Southampton
{bm4g15, k.farrahi, jsh2}@soton.ac.uk

Abstract

Multilayer Perceptrons struggle to learn certain simple arithmetic tasks. Specialist neural modules for arithmetic can outperform classical architectures with gains in extrapolation, interpretability and convergence speeds, but are highly sensitive to the training range. In this paper, we show that Neural Multiplication Units (NMUs) are unable to reliably learn tasks as simple as multiplying two inputs when given different training ranges. Causes of failure are linked to inductive and input biases which encourage convergence to solutions in undesirable optima. A solution, the stochastic NMU (sNMU), is proposed to apply reversible stochasticity, encouraging avoidance of such optima whilst converging to the true solution. Empirically, we show that stochasticity provides improved robustness with the potential to improve learned representations of upstream networks for numerical and image tasks.

1 Introduction

Machine Learning trains models to generalise to data under the assumption of the data being independent and identically distributed (i.i.d) rather than Out-of-Distribution (OOD) [Wang et al., 2021]. Generalisation to OOD data is considered to be extrapolation. Extrapolation is desirable as real-world training data can be limited to a small subset of the domain due to time and cost constraints. For example, a network to predict the kinematics for a robotic arm may be trained on one configuration, but be tested on an updated configuration outside the training one [Martius and Lampert, 2017]. However, networks still struggle to extrapolate on even simple mathematical tasks [Saxton et al., 2019]. One way to improve model compatibility with extrapolative data is via inductive biases [Mitchell, 1980], which help guide the optimisation of parameters using prior assumptions designed into the architecture. Structural biases which result in specialists can also help avoid memorisation, a prevalent issue in neural networks [Zhang et al., 2020].

Neural Arithmetic Logic Modules (NALMs), are specialist differentiable neural modules which learn systematic generalisations of arithmetic/logic operations [Mistry et al., 2022, Trask et al., 2018, Madsen and Johansen, 2020]. NALMs can extrapolate to OOD data while also having interpretable weights. The first NALM, introduced by Trask et al. [2018], focuses on learning arithmetic operations (i.e. $+$, $-$, $\times$, $\div$, square, and square-root). Their Neural Arithmetic Logic Unit (NALU) learns arithmetic operations and selection of relevant inputs in a differentiable manner where the weight values reflect the chosen operation and input. Real-world applications have included the NALU as a sub-component in larger end-to-end systems which learns more complex tasks such as analogy composition [Wu et al., 2020] or scheduling of content-delivery-networks [Zhang et al., 2019]. For the NALM to be a successful sub-component in such networks, three attributes are required. (1) Provide informative gradients for the upstream networks to learn. (2) Be a robust module which can learn regardless of the input as sub-components of such larger networks have limited (if any) control of the distribution/range of their inputs. (3) Learn implicit selection, as there is no guarantee

that all input values (i.e. features) are relevant. Though recent works improve on some of NALU's shortcomings such as convergence, initialisation, and interpretability [Madsen and Johansen, 2020, Schlör et al., 2020, Heim et al., 2020], none have been successful in achieving extrapolation over various input ranges.

Using stochasticity to improve learning is a practice that has been used by the community for many years. Noise can be injected into the input or weights to improve generalisation by implicitly inducing additional regularisation to the cost [An, 1996]. Or instead, noise can added to the gradients to encourage exploration giving the model the opportunity to escape local minimas. Such noise is *irreversible*, meaning that the convergence cannot reach zero, but if annealed throughout training then zero convergence can occur [Neelakantan et al., 2015a]. In contrast to these methods, our input noise 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 different inputs. For example in adversarial training, the noise is added to the input which the network must *learn* to denoise [Goodfellow et al., 2015]. 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 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.

In this paper, we show that by introducing reversible stochasticity to a multiplication NALM (Section 6) it is possible to improve the robustness without compromising on performance even if the NALM is a component in a deep neural network. We focus on multiplication as the effect of the operation can be understood by humans, but the space to learn is difficult enough to pose challenges to the optimisation of neural models. Multiplication is chosen over 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. The contributions of this paper are as follows[1]:

- In Section 3 we motivate using specialist arithmetic modules over MLPs, showing gains in extrapolation, convergence speed and interpretability.

- In Section 5 we show how robustness against different training ranges of specialist modules greatly varies using the simplest task for modelling an operation (i.e., calculating the multiplication of two inputs). We relate the robustness issue to local optima inducing biases and in Section 6 present a novel approach, the sNMU to fix the issue.

- We assess the sNMU on arithmetic regression (Section 7) and image tasks (static MNIST and sequential MNIST in Section 8) showing favourable performance compared to networks with pre-solved multiplication.

2 Overview of Neural Arithmetic Logic Modules

This section details two NALMs used in our experiments, the Neural Addition Unit (NAU) and the Neural Multiplication Unit (NMU). Madsen and Johansen [2020] introduce two specialist modules, one for dealing with addition and subtraction (the NAU) and the other for multiplication (the NMU). The NAU and NMU have a separate weight matrix and can be stacked (i.e. the output of a NAU is the input for a NMU) to learn compositional arithmetic expressions (e.g. $(x_1 + x_2) \times (x_3 - x_4)$). An NAU output element a_o is defined as

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

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

$$\text{NMU} : m_o = \prod_{i=1}^{I} (W_{i,o} \cdot \mathrm{x}_i + 1 - W_{i,o}). \tag{2}$$

---

[1]Code (MIT license) is available at: `https://anonymous.4open.science/r/nalm-robust-nmu-7577`.

Table 1: Comparison of learning multiplication using a MLP or NMU for multiplying two inputs. The extrapolation and interpolation error correspond to the iteration with the lowest validation (interpolation) error. For the MSE columns, values in brackets represent the percentage of successful seeds which meet the minimum extrapolation criteria of 1e-5. Solved iteration represents the iteration which successful models get solved at. 95% confidence intervals are given.

Model	#Params	Interpolation MSE; (%)	Extrapolation MSE; (%)	Solved Iteration	Iterations
MLP (1)	5	2.3E-01 $\pm$ 7.7E-02 (0)	1.9E+02 $\pm$2.9E01 (0)	-	5E+04
MLP (100)	401	3.4E-07 $\pm$ 4.7E-08 (100)	3.7E+01 $\pm$ 9.5E-01 (0)	-	2E+06
NMU	**2**	**4.6E-14 $\pm$ 0 (100)**	**9.9E-13 $\pm$ 0 (100)**	**1.0E+04 $\pm$ 2.8E+02**	5E+04

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, regularisation is applied for a given period of training (see Appendix A) and before applying a sub-unit the weight matrix is clamped to [-1,1] for the NAU or [0,1] for the NMU.

3 Motivating Specialist Modules over MLPs

We first demonstrate the advantage of using a specialist module, the NMU, over an MLP for multiplication in terms of extrapolation, convergence speed and interpretability for the task of applying the operation to two inputs. A single hidden layer MLP network with a width of either 1 or 100 is learnt to check both extremes. A ReLU activation is used and we keep bias terms to avoid constraining expressiveness.

Setup: Given two inputs x_1 and x_2, output the value for $x_1 \times x_2$. Networks are trained on a range $\mathcal{U}[1,2]$ and tested on an extrapolation range $\mathcal{U}[2,6]$. Results show the average error and best iteration step with a 95% confidence interval. For experiment details see Single Module Task in Appendix B.

Results: Table 1 shows MLPs are unable to find any solutions for multiplication. Comparing a width of 1 and 100: (1) Increasing width reduces both interpolation and extrapolation mean square error (MSE) at the cost of increased training iterations and parameters, but extrapolative solutions are not learnt. (2) Extremely wide networks encourage memorisation over the interpolation range which is unable to hold over the extrapolation range. (3) Robustness over different initialisations improves with increased width. In contrast, the NMU, which requires learning only two parameters (which is three less parameters than the MLP of width one), has success on all initialisations, is robust, and requires the least iterations to be solved. Comparing the learnt 2D surface plots (Figure 1) of the MLPs and the NMU better visualises points (1) and (2) from above.
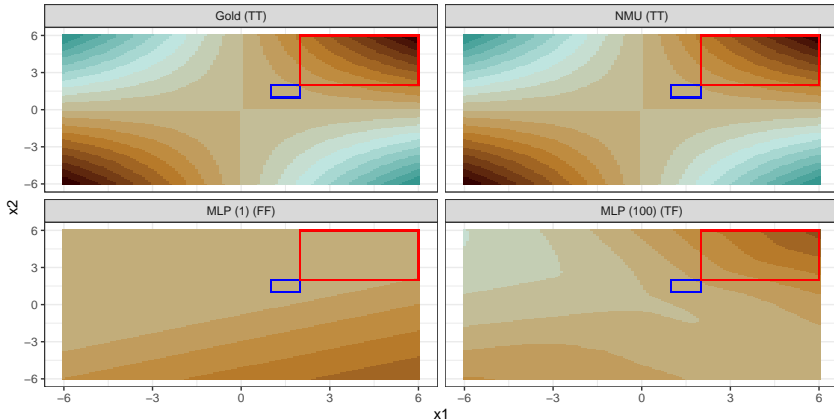

Figure 1: 20 bin surface plots for the multiplication operation, comparing the golden solution (top left) to a learnt NMU (top right) and MLPs of different widths (bottom row). The letters in the brackets are True (T)/False (F), representing if the minimum loss threshold (1e-5) for the interpolation and extrapolation range have been met respectively. The blue and red squares represent the interpolation (training) and extrapolation (test) ranges respectively.

3

Table 2: 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)	

4 Numerical Experiment Setups and Evaluation Metrics

Before learning larger systems, we must first verify if NALMs can work as an independent unit. As NALMs expect unnormalised inputs, we use two synthetic arithmetic regression tasks to test robustness against different training ranges. This section describes the experimental setup for our 'Single Module Task' and the 'Arithmetic Dataset Task' [Madsen and Johansen, 2020]. We also detail the three evaluation metrics: *success rate, speed of convergence* and *sparsity error* along with the interpolation and extrapolation ranges. Summaries of parameters and hardware/runtimes are found in Appendices B and C.

4.1 Single Module Task

Motivation. To determine the best module to further explore, we evaluate if the simplest possible multiplication task can be learnt over various training ranges.

Setup. Given two inputs x_1 and x_2 (as floats), use a single module to calculate the value for $x_1 \times x_2$. We test a variety of existing multiplication NALMs including the: NALU [Trask et al., 2018], NAC$_\bullet$ [Trask et al., 2018], NMU [Madsen and Johansen, 2020], iNALU [Schlör et al., 2020], G-NALU [Rana et al., 2019], NPU, and Real NPU [Heim et al., 2020]. A detailed explanation of each module can be found in Mistry et al. [2022] and therefore is omitted here.

4.2 Arithmetic Dataset Task

Motivation: This non-trivial synthetic task assesses if a multiplication module can pass useful gradients to learn the upstream layer. We first focus on learning linear units before moving onto more complex image based tasks where the solution of upstream layers is non-trivial.

Setup. This task requires learning a two-layer NALM for performing addition (in the first layer) followed by multiplication (in the second layer). The input vector, for a single unbatched data sample, consists of a 100 floating point numbers drawn from a uniform distribution with an upper and lower bound. The addition module must learn to select and sum two different overlapping subsets of this vector. Each subset is created from a continuous slice of the input between a lower and upper index. The proportion of a subset in comparison to the input and overlap ratio between the two subsets are set as 0.25 and 0.5 respectively. The subsets selected remain consistent for a batch but can vary for different seeds. Each seed uses a different model initialisation and a different training dataset for a range. The multiplication module must select both output values of the addition module and multiply them to give the predicted output value. We use Madsen and Johansen [2020]'s hyperparameters as these values are selected from parameter sweeps.

4.3 Evaluation Metrics

To quantitatively evaluate the modules, we use Madsen and Johansen [2019]'s evaluation scheme. We assess the NALMs over multiple ranges. Interpolation (training/validation) and extrapolation (test) ranges are presented in Table 2 and are chosen based on previous work [Madsen and Johansen, 2020] which requires the interpolation and extrapolation ranges not to overlap in order to test out-of-distribution performance. Early stopping is applied using a validation dataset sampled from the interpolation range.

The three evaluation metrics are: the success on the extrapolation dataset against a near optimal solution (*success rate*), the first iteration which the task is considered solved (*speed of conver-*
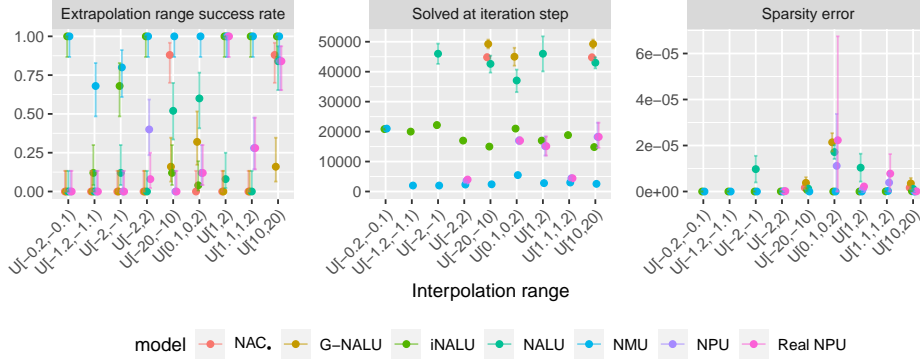
Figure 2: Single Module Task for multiplication.

gence), and the extent of discretisation of the weights (*sparsity error*). Sparsity error calculated by $\max_{i,o}(\min(|W_{i,o}|, 1 - |W_{i,o}|))$, measures the NALM weight element which is the furthest away from the acceptable discrete weights for a NALM. 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 an optimal weight value by $\epsilon = $1e-5. The Single Module Task also uses a simulated threshold value with an $\epsilon = $1e-5 (see Appendix D for details). 95% confidence intervals are calculated from a specific family of distributions dependant on the metric. The success rate uses a Binomial distribution because trials (i.e. run on a single seed) are pass/fail situations. The convergence metric uses a Gamma distribution and sparsity error uses a Beta distribution.

5 Robustness Issues with Multiplication Modules

Using the Single Module Task (Section 4), we highlight robustness issues when training on different inputs and relate the cause of the problem to local optimas. Results are displayed in Figure 2. 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 $\mathcal{U}[-2, -1]$. Though NPUs in theory can learn with negative inputs, empirical results suggest the module struggles. The $\text{NAC}_\bullet$ outperforms the NALU on all ranges except $\mathcal{U}[1,2]$, $\mathcal{U}[10,20]$, and $\mathcal{U}[-2,2]$ where both successes are equal. The failure of $\mathcal{U}[-2,2]$ is expected due to the $\text{NAC}_\bullet$'s inability to deal with mixed signed inputs, caused by the module using
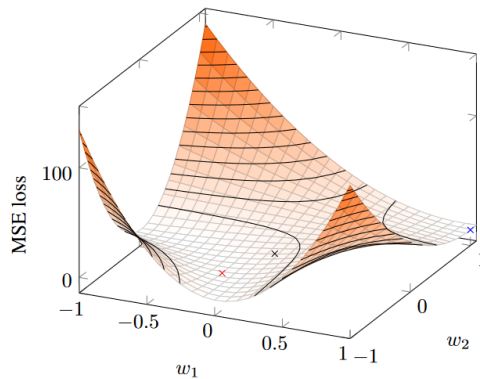

Figure 3: Static Loss Landscape with batch size of 1 for NMU weights in a Single Module Task for learning '-2×-1.8'. Ideally the weights should converge to the global minima $(1,1)$ (blue cross) which is the extrapolative solution. However, an alternate minima at $(-\frac{1}{6},-0.5)$ (red cross) exists which solves -2×-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.
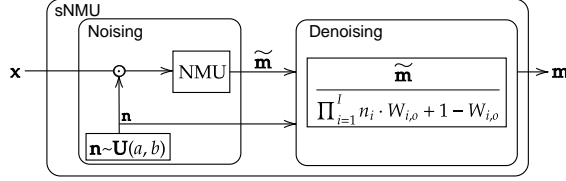
5

Figure 4: Stochastic NMU architecture

a log-exponential transformation in its architecture. Next, we reason as to why the NMU fails. We focus on the NMU because it has the best performance overall.

Problem: Inputs that Induce Local Optima.   The NMU fails to get 100% success for interpolation ranges $\mathcal{U}$[-1.2,-1.1) and $\mathcal{U}$[-2,-1). We find weights converge towards other local optima with values outside the applied clipping range of [0, 1]. Figure 3 illustrates this issue. 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). As the input range is an independent variable in our experiments, we conclude that the input data influences the weight learning towards local minimas which get wrongly enforced by the model's bias towards discrete weights (from regularisation and weight clipping).

6   Stochastic NMU (sNMU)

We propose the stochastic NMU (sNMU), Figure 4 to solve the above issue. This sNMU achieves full success on the Single Module Task on all ranges (Appendix E) without compromising the existing advantages of the NMU i.e., the low parameter count, fast solving speeds and low sparsity errors.

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 resulting output value would be as though the input is applied to the NMU without noise. As gradients are influenced by the inputs, by manipulating the input with noise, the resulting gradients are more likely to favour the correct optimum and has a loss landscape with fewer optima near the subspace of desired weight values which only solve the interpolation range. 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:

$$\text{NMU}_{\text{noisy}} : \tilde{m}_o = \prod_{i=1}^{I} (n_i x_i W_{i,o} + 1 - W_{i,o}) \ . \tag{3}$$

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:

$$\text{sNMU} : m_o = \frac{\tilde{m}_o}{\prod_{i=1}^{I} (n_i W_{i,o} + 1 - W_{i,o})} \ . \tag{4}$$

The noising the denoising is only used during training; during inference, the module will act exactly like a NMU. To the best of our knowledge, our paper is the first in using reversible noise in a NALM. The sNMU weight values remain interpretable as 0 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. For additional discussion, refer to Appendix F.
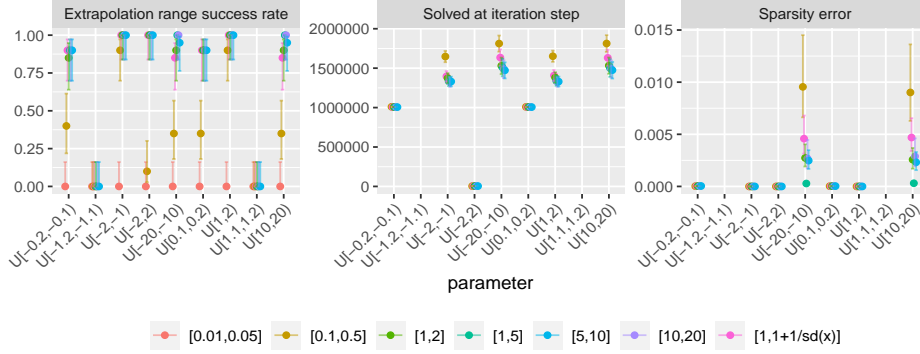
Figure 5: Arithmetic Dataset Task for multiplication over various noise ranges for the NAU-sNMU.

7 Arithmetic Dataset Task

We now evaluate against the more complex two-layer task (see Section 4.2), presenting results for the stacked NAU-NMU and NAU-sNMU. We first compare the effects of different noise ranges and take the best performing noise to compare against the NMU based model.

Effect of noise range. Figure 5 shows the effect of using different noise ranges for the sNMU. Smaller ranges under one perform worse for data generated with a Uniform distribution and too large a range also shows degradation on 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 $\mathcal{U}[1, 1 + \frac{1}{\sigma(\mathbf{x})}]$ (Figure 6), where $\sigma(x)$ is the standard deviation of the sNMU input $\mathbf{x}$, also achieves reasonable results, but the range $\mathcal{U}[1,5]$ is found to be the best for this task in regards to the three metrics.

NMU vs sNMU. Figure 7 shows the stacked NAU-NMU fails on multiple ranges. Similar to the Single Module Arithmetic Task, the range $\mathcal{U}[-2,2)$ is solved instantly. The sNMU shows improvement with faster solve speeds and lower sparsity errors compared to the NMU. Furthermore, 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 $\mathcal{U}[0.1,0.2)$ 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 remaining failures can be explained due to the use of an MSE loss (which is out of the scope here, but is covered in Appendices G and H).
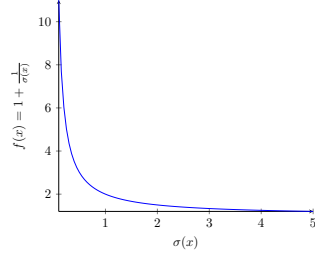

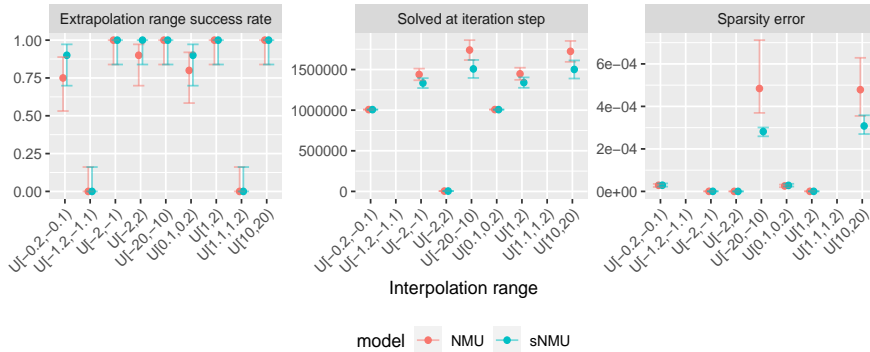Figure 6: Upper bounds for batch statistic noise

Figure 7: Arithmetic Dataset Task for multiplication. Models include: NMU (stacked NAU-NMU) and sNMU (stacked NAU-sNMU) with a noise range of [1,5].

(a) Isolated digits

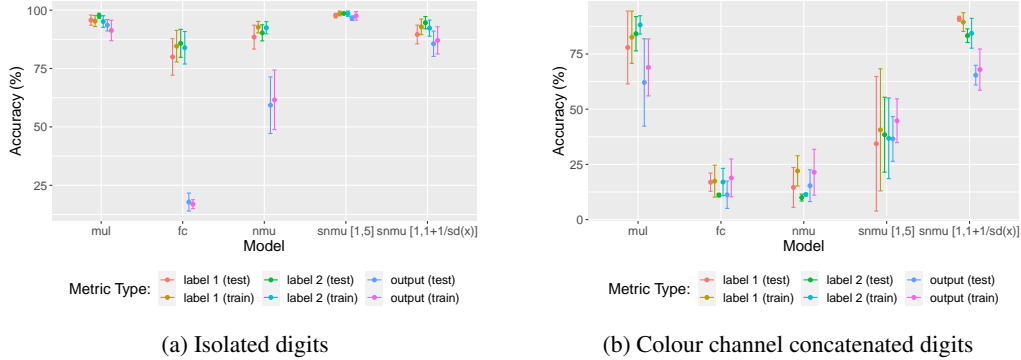(b) Colour channel concatenated digits

Figure 8: Isolated digit (left) and rounded colour channel concatenated (right) accuracies for classifying each of the two digits (label 1 and label 2) and the final product (output).

8 MNIST Arithmetic

This section explores the effect of including specialist multiplication modules as a downstream layer for two 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 parameters and hardware/runtimes can be found in Appendices B and C.

8.1 Isolated Digit Classification

Motivation. First, we determine if digit classification can be learnt 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. E.g., 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 to single digits, classify per digit and the recombine the two labels. The digit classifier is a convolutional network[2]. 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 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.

Metrics and results. 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. A strict criteria for measuring accuracy is used 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. Figure 8a shows the results. Both the sNMUs with 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. The FC model learns to classify each label to a reasonable accuracy but does not learn the multiplication weights robustly, whereas the specialised modules, the NMU and sNMU do (see Figure 15 in Appendix I). The $\mathcal{U}[1,5]$

---

[2]From the PyTorch MNIST example `https://github.com/pytorch/examples/blob/master/mnist/main.py`
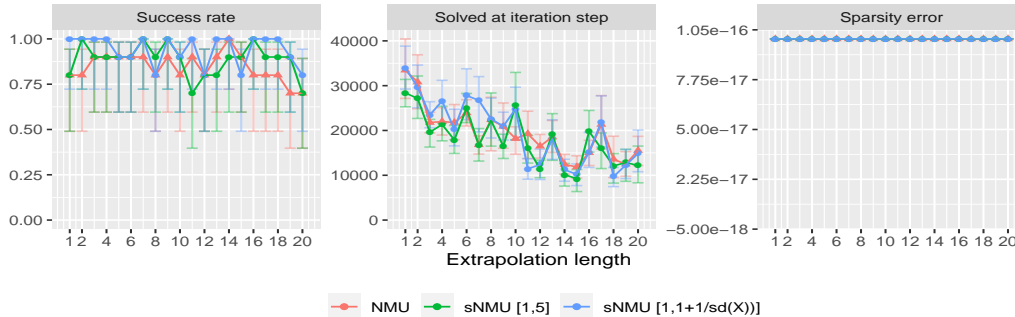
Figure 9: Performance on Product of Sequential MNIST. Model names represent the type of multiplication cell used. All models use the same CNN architecture to do digit classification.

sNMU outperforms both the NMU and MUL model, suggesting that the reversible stochasticity not only improves robustness but can aid with learning upstream layers.

8.2 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. The digit classifier uses a Spatial Transformer [Jaderberg et al., 2015] with a Thin Plate Spline transformation for digit localisation [Bookstein, 1989] which is bounded [Shi et al., 2016]. The multiplication layer uses the same options from Section 8.1 (i.e., MUL, FC, NMU and sNMU).

Metrics and results. The accuracies of each digit label and the final output value are taken. Due to the increased classification difficulty, accuracies are rounded. Figure 8b 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 the batch noise. The NMU fails to converge towards the correct multiplication weights for a fold (see Figure 16 in Appendix I), unlike its stochastic versions, achieving similar accuracies to the FC based network.

8.3 Sequential MNIST Product

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

Setup and network. Following Madsen and Johansen [2020], given a sequence of MNIST digits, process one image at a time using a classification network to convert an image to its label value, which gets passed into a recursive NALM cell to calculate the cumulative result. 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 digits. The multiplication layer is either solved (baseline), or requires learning a NALM (NMU or sNMU).

Metrics and results. The success threshold is the 1% one-sided upper confidence-interval using a student-t distribution over the MSE of solved NALM models. Training uses two-digit sequences while testing uses sequences up to 20 digits long. Figure 9 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.

9

9 Discussion

We provide an explanation for the lack of robustness to training ranges of a numerical unit, leading us to propose the stochastic NMU (sNMU) resulting in improved extrapolation. As the process is fully reversible the injected noise does not hinder the ability of the module to learn the exact operation. The stochastic nature of the module provides gradients to avoid local optimas which a regular NMU falls into, while still propagating a useful signal to upstream layers as shown by performance on the arithmetic image tasks. Though we focus on providing a proof of concept for using reversible stochasticity to improve learning, this idea can be further explored. For example, using other distributions of noise, learnable noise ranges or other forms of batch statistics. Future investigation could also be done to learn why certain noise ranges work better than others.

References

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.1 996.8.3.643. URL https://doi.org/10.1162/neco.1996.8.3.643.

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. URL https://link.springer.com/article/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. URL http://user.engineering.uiowa.edu/~aip/papers/bookstein-89.pdf.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, dec 2015. URL https://arxiv.org/pdf/1412.6572.pdf.

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/48e5900 0d7dfcf6c1d96ce4a603ed738-Paper.pdf.

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*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper/2015/file/33ceb07bf4eeb3da587e268d6 63aba1a-Paper.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 http://arxiv.org/abs/1910.01888.

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

Georg S Martius and Christoph Lampert. Extrapolation and learning equations. In *5th International Conference on Learning Representations, ICLR 2017-Workshop Track Proceedings*, 2017. URL https://openreview.net/pdf?id=ryUPiRvge.

Bhumika Mistry, Katayoun Farrahi, and Jonathon Hare. A primer for neural arithmetic logic modules. *Journal of Machine Learning Research*, 23(185):1–58, 2022. URL `http://jmlr.org/papers/v23/21-0211.html`.

Tom M Mitchell. *The need for biases in learning generalizations*. Department of Computer Science, Laboratory for Computer Science Research, 1980. URL `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.5466&rep=rep1&type=pdf`.

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, 2015a. URL `https://arxiv.org/abs/1511.06807`.

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. *arXiv preprint arXiv:1511.06807*, 2015b. URL `https://arxiv.org/abs/1511.06807`.

Ashish Rana, Avleen Malhi, and Kary Främling. Exploring numerical calculations with calcnet. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1374–1379. IEEE, 2019. URL `https://ieeexplore.ieee.org/document/8995315`.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=H1gR5iR5FX`.

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`.

Baoguang Shi, Xinggang Wang, Pengyuan Lyu, Cong Yao, and Xiang Bai. Robust scene text recognition with automatic rectification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4168–4176, 2016. URL `https://openaccess.thecvf.com/content_cvpr_2016/papers/Shi_Robust_Scene_Text_CVPR_2016_paper.pdf`.

Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019. URL `https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0197-0`.

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

Jindong Wang, Cuiling Lan, Chang Liu, Yidong Ouyang, and Tao Qin. Generalizing to unseen domains: A survey on domain generalization. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4627–4635. International Joint Conferences on Artificial Intelligence Organization, 8 2021. doi: 10.24963/ijcai.2021/628. URL `https://doi.org/10.24963/ijcai.2021/628`. Survey Track.

Bo Wu, Haoyu Qin, Alireza Zareian, Carl Vondrick, and Shih-Fu Chang. Analogical reasoning for visually grounded language acquisition. *arXiv preprint arXiv:2007.11668*, 2020. URL `https://arxiv.org/pdf/2007.11668.pdf`.

Yutaro Yamada, Ofir Lindenbaum, Sahand Negahban, and Yuval Kluger. Feature selection using stochastic gates. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 10648–10659. PMLR, 13–18 Jul 2020. URL `https://proceedings.mlr.press/v119/yamada20a.html`.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Michael C. Mozer, and Yoram Singer. Identity crisis: Memorization and generalization under extreme overparameterization. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=B1l6y0VFPr`.

Ruixiao Zhang, M. Ma, Tianchi Huang, Haitian Pang, X. Yao, Chenglei Wu, J. Liu, and L. Sun. Livesmart: A qos-guaranteed cost-minimum framework of viewer scheduling for crowdsourced live streaming. *Proceedings of the 27th ACM International Conference on Multimedia*, 2019. URL `https://dl.acm.org/doi/10.1145/3343031.3351013`.

A Regularisation used by the NAU and NMU

To enforce discretisation of weights both units have a regularisation penalty for a given period of training. The penalty is

$$\lambda \cdot \frac{1}{I \cdot O} \sum_{o=1}^{O} \sum_{i=1}^{I} \min\left(|W_{i,o}|, 1 - |W_{i,o}|\right), \tag{5}$$

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

$$\lambda = \hat{\lambda} \cdot \max\left(\min\left(\frac{iteration_i - \lambda_{start}}{\lambda_{end} - \lambda_{start}}, 1\right), 0\right). \tag{6}$$

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 $\hat{\lambda}$.

B Experiment Parameters

Refer to Tables 3, 4, 5 and 6 for the breakdown of parameters used in the Single Module Task and Arithmetic Dataset Task. Regularisation is only applied for a given period of time to avoid forcing weights to converge too early/late. For example, if started to early then the weights will not have a chance to converge to the correct value in time for discretisation. If started too late there is a danger the discretisation will not complete before the maximum number of training iterations is reached.

Refer to Table 7 for the breakdown of parameters used in the Static MNIST Product experiments.

Refer to Madsen and Johansen [2020] for the hyperparameters, setup, and regularisation used in the original experiment used in the Sequential MNIST Product experiments.

Table 3: 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.

Parameter	Arithmetic Dataset Task	Single Module Task
Layers	2	1
Input size	100	2
Subset ratio	0.25	0.5
Overlap ratio	0.5	0
Total iterations	5 million	50000
Train samples	128 per batch	128 per batch
Validation samples*	10000	10000
Test samples*	10000	10000
Seeds	20	25
Optimiser	Adam (with default parameters)	Adam (with default parameters)

Table 4: Additional parameters for the NMU (and NAU) for the Single Module and Arithmetic Dataset Task. The $\hat{\lambda}$, λ_{start}, λ_{end}.

Parameter	Arithmetic Dataset Task	Single Module Task
$\hat{\lambda}$	10	10
λ_{start}	1 million	20000
λ_{end}	2 million	35000
Learning rate	1.00E-03	1.00E-03

Table 5: Parameters specific to the NPU and RealNPU modules for the Single Module Task.

Parameter	Value
$(\beta_{start}, \beta_{end})$	(1e-7,1e-5)
β_{growth}	10
β_{step}	10000
Learning rate	5.00E-03

Table 6: Parameters specific to the iNALU for the Single Module Task.

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

Table 7: Static MNIST Product experiment parameters.

Parameter	Two digit independent	Localisation with Spatial Transformers
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 (with default parameters)	Adam (with default parameters)
Criterion	MSE	MSE
Learning rate	1e-3	1e-3
$\lambda_{start} - \lambda_{end}$ **epochs**	30-40	30-40
$\hat{\lambda}$	100	100

C Hardware and Time to Run Experiments

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 CPU server with 125 GB memory 1.2 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 a script execution because the queuing time from jobs when executing scripts is not relevant to the experiment timings.

All experiments for the Arithmetic Dataset 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 8.

Table 8: 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 NMU sNMU $\mathcal{U}[1,5]$ sNMU $\mathcal{U}[1,1+1/sd(x)]$	MSE	GPU	1000	02:00:00 02:55:00 03:00:00 03:10:00	679

D Single Module Task Evaluation:

We adopt the Madsen and Johansen [2019]'s evaluation scheme used for the Arithmetic Dataset Task but adapt the expression used to generate the predictions of an ϵ-perfect model (y_o^ϵ). For multiplication, the expression used is:

$$y_o^\epsilon = (x_1 x_2)(1 - \epsilon)^2 \times \prod_{i \in X_{irr}} (1 - |x_i|\epsilon) \tag{7}$$

Assume x_1 and x_2 are the operands to apply the operation to and any remaining features ($x_3, ..., x_n$) be irrelevant to the calculation and part of the set X_{irr}. We use I to denote the total number of input features. The ϵ for each feature will contribute some error towards the prediction. A simulated MSE is then generated with an $\epsilon = 1e - 5$ 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.

E Single Module Task for sNMU on Multiplication

Figure 10 shows the robustness of the sNMU on the Single Module Task. The sNMU achieves full success on all training ranges and solves the ranges within a reasonable number of iterations.

Figure 10: Single Module Task for multiplication comparing the NMU to a stochastic NMU (sNMU) with noise sampled from U[1,5].

F Stochastic NMU - Additional Thoughts

Why not just divide by the noise values? Denoising using only the product of the noise values ($\prod_{i=1}^{I} n_i$) is not valid for cases with redundant inputs (i.e. $w_i = 0$), where not all inputs are selected for multiplication. To alleviate the redundancy issue we multiply the noise with the weight values ($n_i W_{i,o}$). However, this causes a division by 0 if weight values are 0 or weight/noise values are close to 0 where numeric precision errors occur. Therefore, we include the $+1 - W_{i,o}$ term, resulting in

lowest possible value being the product of the noise. As the noise distribution is predefined, the lower bound of the noise can be controlled, avoiding issues with very small noise values.

What about other denoising factors? The linearity of the denominator in the chosen denoising factor is required for robust convergence. For example, an alternate mathematically valid denoising factor could be

$$\frac{m_o^{(n)}}{\prod_{i=1}^{I} n_i^{w_i}}, \tag{8}$$

where each partial calculation of the product in the denominator is a value between $[1, n_i]$. This form of denoising was found to encourage the model to converge to optima which could solve interpolation data but not the extrapolation data. Furthermore, issues regarding stability occur for noise values close to 0, and negative noise values are not allowed due to causing complex-valued solutions if weight values are between 0 and 1.

Are the inputs/outputs normalised? We do not normalise, rather part of our novel step is to completely reverse the effect of the stochasticity. Generic normalisation cannot work for such modules as multiplying normalised inputs gives the incorrect output and unnormalising to the correct output would require prior knowledge of the task.

Why apply stochasticity at the module level, and not the full network level? 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. At module level, this complexity is vastly reduced.

Motivations to real-world tasks: Our proposed module, the sNMU, are part of Neural Arithmetic Logic Modules (NALMs) which are independent reusable modules. Such modules can be integrated into networks for specific applications, such as Zhang et al. [2019]'s deep reinforcement network which schedules content-delivery-networks using a NALM's extrapolative ability to reduce failures outside the training range. Their work uses a module, whose multiplication unit could be replaced with the sNMU allowing for improved robustness.

How are gradients affected by the stochasticity? For the two layer arithmetic dataset task, compare the derivations for gradient updates for the NAU-NMU and NAU-sNMU models (see Appendix H), finding that the noise results in additional scaling of the gradients which does not exist in the NMU version.

Intuition behind the automatic batch statistic based noise range. The automatic noise range samples noise from a uniform distribution in which the lower bound is 1 and the upper bound is $1 + 1/\sigma(x)$. The lower bound is set to 1 so the noise cannot scale down the gradient magnitude. The upper bound function (see Figure 6 of the main body) 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.

G   Arithmetic Dataset Task Cause of Failure: Uninformative MSE loss

From Section 7 we saw certain training ranges resulted in no success for any model used. These failures are due to the input range causing difficulty for the NAU weights to select the relevant inputs. This is explained by the following scenario. Imagine the same task but with input size 4 and overlap and 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:

$$i_1 W^{(\text{NAU})} W^{(\text{NMU})} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 35$$

Table 9: Output values and absolute errors for simplified 2-layer task inputs i_1 [1, 2, 3, 4] and i_2 [1.11, 1.12, 1.13, 1.14]. Selection is if the NAU module selected the correct inputs. Weights is if the weights for the NAU are 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

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 ($\boldsymbol{W}^{(\mathrm{NAU})}$) is:

$$\text{Case 1: } \begin{bmatrix} 0 & 0 \\ 0.9 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \text{Case 2: } \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 0.9 & 1 \end{bmatrix}$$

Calculating the output and the absolute error (from the ideal solution) of these cases for the both inputs (Table 9) shows that [1.11, 1.12, 1.13, 1.14] has a much smaller difference in error than [1, 2, 3, 4]. The model will struggle to differentiate between correct and incorrect selection of weights for the input drawn from the distribution with a smaller range. This specific case also shows that the selection of an incorrect input and non-converged weight gives 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 invalid NAU weights (see Appendix H for derivations). Considering the stacked NAU-NMU, the gradients of both the NAU and NMU weight matrix is scaled by a residual factor ($\boldsymbol{y} - \hat{\boldsymbol{y}}$). We know that 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. When using the NAU-sNMU the gradients get scaled by a noise term (whose magnitude can be predefined to > 1). However, the residual term still remains, creating small gradients.

H   Arithmetic Dataset Task Partial Derivatives

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 ($\boldsymbol{W}^A$) = Weight matrix of a summative module (i.e. NAU)
- Superscript M ($\boldsymbol{W}^M$) = 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. $\boldsymbol{X}_n$ is the input vector for batch item n with target scalar y_n and predicted scalar $\hat{y}_n$. I is the number of
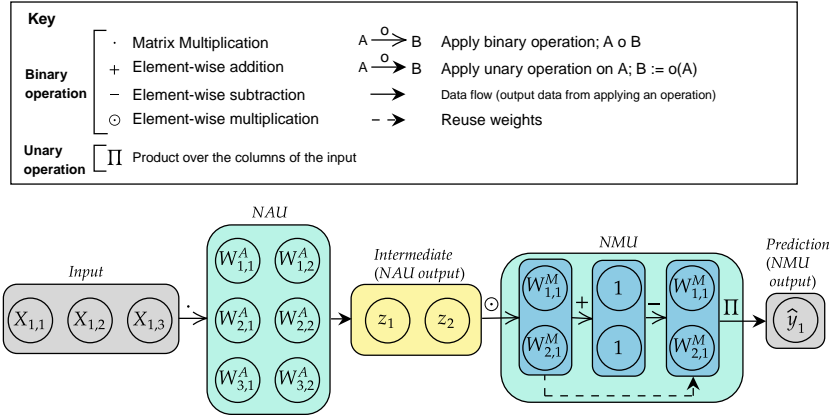
Figure 11: Illustration of the data flow of a NAU-NMU module. The annotation of weights will be consistent with the gradient calculations.

input elements (=100) in the input vector.

$$L = \frac{1}{N} \sum_{n}^{N} (y_n - \hat{y}_n)^2$$

$$L = \frac{1}{N} \sum_{n}^{N} (y_n - \text{NMU}(\text{NAU}(\boldsymbol{X_n})))^2$$

$$z_1 = \sum_{i}^{I} (X_{n,i} \cdot W_{i,1}^A)$$

$$z_2 = \sum_{i}^{I} (X_{n,i} \cdot W_{i,2}^A)$$

$$L = \frac{1}{N} \sum_{n}^{N} [y_n - ((1 + W_{1,1}^M \cdot z_1 - W_{1,1}^M)$$
$$(1 + W_{2,1}^M \cdot z_2 - W_{2,1}^M))]^2$$

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 $(\boldsymbol{X_1}, y_1)$. We annotate the weights and components explicitly in Figure 11. Matrix indexing follows the standard (row, column) convention, with indexing starting from 1.

H.2.1   MSE Loss Partial Derivatives:

Derivation 1 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 underline red and blue terms as the parts. To differentiate each part further, also requires another application of the product rule.

$$L = (y_1 - \hat{y_1})^2$$

$$\hat{y_1} = (W_{1,1}^M \cdot (X_{1,1} \cdot W_{1,1}^A + X_{1,2} \cdot W_{2,1}^A + X_{1,3} \cdot W_{3,1}^A) + 1 - W_{1,1}^M) \cdot$$
$$(W_{2,1}^M \cdot (X_{1,1} \cdot W_{1,2}^A + X_{1,2} \cdot W_{2,2}^A + X_{1,3} \cdot W_{3,2}^A) + 1 - W_{2,1}^M)$$

$$\frac{\partial L}{\partial W_{1,1}^A} = -2 \cdot \frac{\partial \hat{y_1}}{\partial W_{1,1}^A} \cdot (y_1 - \hat{y_1})$$

$$\frac{\partial \hat{y_1}}{\partial W_{1,1}^A} = W_{1,1}^M \cdot X_{1,1} \cdot (W_{2,1}^M \cdot (X_{1,1} \cdot W_{1,2}^A + X_{1,2} \cdot W_{2,2}^A + X_{1,3} \cdot W_{3,2}^A) + 1 - W_{2,1}^M)$$

$$= W_{1,1}^M \cdot X_{1,1} \cdot (W_{2,1}^M \cdot (\sum_i^I X_{1,i} \cdot W_{i,2}^A) + 1 - W_{2,1}^M)$$

$$\frac{\partial L}{\partial W_{1,2}^A} = -2 \cdot \frac{\partial \hat{y_1}}{\partial W_{1,2}^A} \cdot (y_1 - \hat{y_1})$$

$$\frac{\partial \hat{y_1}}{\partial W_{1,2}^A} = (W_{1,1}^M \cdot (X_{1,1} \cdot W_{1,1}^A + X_{1,2} \cdot W_{2,1}^A + X_{1,3} \cdot W_{3,1}^A) + 1 - W_{1,1}^M) \cdot W_{2,1}^M \cdot X_{1,2}$$

$$= (W_{1,1}^M \cdot (\sum_i^I X_{1,i} \cdot W_{i,1}^A) + 1 - W_{1,1}^M) \cdot W_{2,1}^M \cdot X_{1,2}$$

$$\frac{\partial L}{\partial W_{1,1}^M} = -2 \cdot \frac{\partial \hat{y_1}}{\partial W_{1,1}^M} \cdot (y_1 - \hat{y_1})$$

$$\frac{\partial \hat{y_1}}{\partial W_{1,1}^M} = ((X_{1,1} \cdot W_{1,1}^A + X_{1,2} \cdot W_{2,1}^A + X_{1,3} \cdot W_{3,1}^A) - 1) \cdot$$
$$(W_{2,1}^M \cdot (X_{1,1} \cdot W_{1,2}^A + X_{1,2} \cdot W_{2,2}^A + X_{1,3} \cdot W_{3,2}^A) + 1 - W_{2,1}^M)$$

$$= (\sum_i^I (X_{1,i} \cdot W_{i,1}^A) - 1) \cdot (W_{2,1}^M \cdot (\sum_i^I (X_{1,i} \cdot W_{i,2}^A) + 1 - W_{2,1}^M)$$

Derivation 1: 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 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 the 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 $(\boldsymbol{X_1}, y_1)$.

To reiterate, the NAU weight matrix is denoted as $W_{l,i}^A$ 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.

$$\frac{\partial L}{\partial W_{i,l}^A} = -2(y_1 - \hat{y_1}) \cdot W_{l,1}^M X_{1,i}$$

$$\cdot \prod_{j}^{\{O\backslash l\}} (W_{j,1}^M (\sum_{k=1}^{I} X_{1,k} W_{k,j}^A) + 1 - W_{j,1}^M)$$

$$\frac{\partial L}{\partial W_{l,1}^M} = -2(y_1 - \hat{y_1}) \cdot (\sum_{i=1}^{I} X_{1,i} W_{i,l}^A - 1)$$

$$\cdot \prod_{j}^{\{O\backslash l\}} (W_{j,1}^M (\sum_{k=1}^{I} X_{1,k} W_{k,j}^A) + 1 - W_{j,1}^M)$$

$\{O\backslash l\}$ 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

$$L = (y_1 - \hat{y_1})^2$$
$$= (y_1 - \text{sNMU}(\text{NAU}(\boldsymbol{X_1})))^2$$

H.4.2 Loss derivatives wrt NAU and sNMU weights

Let

$$A = \prod_{j}^{\{O\backslash l\}} N_{1,j} W_{j,1}^M \cdot (\sum_{k=1}^{I} X_{1,k} W_{k,j}^A) + 1 - W_{j,1}^M,$$

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,

$$\frac{\partial L}{\partial W_{i,l}^A} = -2(y_1 - \hat{y_1}) \cdot \frac{A}{D} \cdot W_{i,1}^M N_{1,i} X_{1,i}$$

$$\frac{\partial L}{\partial W_{i,1}^M} = -2(y_1 - \hat{y_1}) \cdot \frac{A}{D^2} \cdot [D(N_{1,i} z_i - 1)$$

$$- (W_{i,1}^M N_{1,i} z_i + 1 - W_{i,1}^M)(N_{1,i} - 1)$$

$$(\prod_{j}^{\{O\backslash l\}} (N_{1,j} W_{j,1}^M + 1 - W_{j,1}^M))]$$

Although the residual term $(y_1 - \hat{y_1})$ remains, the sNMU will scale the gradients by some noise factor $\geqslant 1$ which magnifies the magnitudes of the gradients. Hence, the sNMU's noise amplifies its gradients helping to alleviate the gradient dampening caused by the residual term.

I   Static MNIST Product

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

I.1   Architecture Details.

Isolated Digits The digit classification network can be found in Figure 12. The output of the digit classifier is passed to a multiplication module which returns the final output predictions.
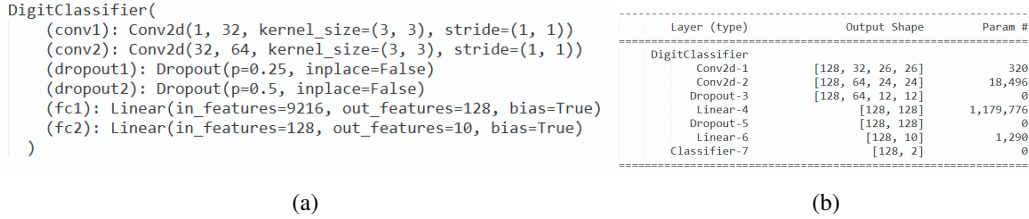
```
DigitClassifier(
    (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
    (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
    (dropout1): Dropout(p=0.25, inplace=False)
    (dropout2): Dropout(p=0.5, inplace=False)
    (fc1): Linear(in_features=9216, out_features=128, bias=True)
    (fc2): Linear(in_features=128, out_features=10, bias=True)
)
```

```
----------------------------------------------------------------
    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
================================================================
```

(a)	(b)

Figure 12: Digit classification network structure and summary used in the Isolated Digits MNIST task

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 ub this case is the digit. As there are two digits, two localisations are learnt. A localisation network consists of a convolutional network (see Figure 13) with a tanh transformation at the end for normalisation to [-1,1]. 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 14) to produce logits for digit classification. The classified digits are then passed to the relevant multiplication network.
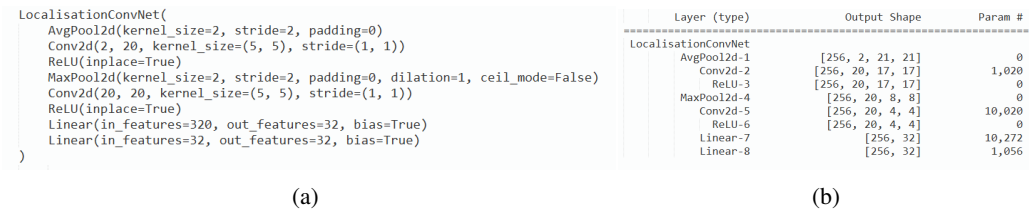
```
LocalisationConvNet(
    AvgPool2d(kernel_size=2, stride=2, padding=0)
    Conv2d(2, 20, kernel_size=(5, 5), stride=(1, 1))
    ReLU(inplace=True)
    MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    Conv2d(20, 20, kernel_size=(5, 5), stride=(1, 1))
    ReLU(inplace=True)
    Linear(in_features=320, out_features=32, bias=True)
    Linear(in_features=32, out_features=32, bias=True)
)
```

```
-------------------------------------------------------------
    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
=============================================================
```

(a)	(b)

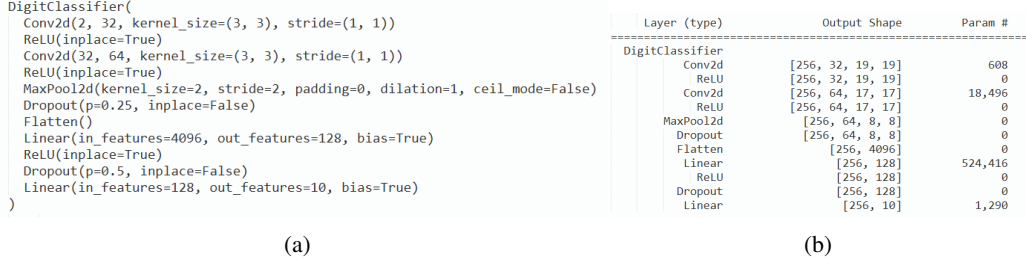Figure 13: Localisation network structure and summary used in the Colour Channel Concatenated Digits MNIST task

```
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)
)
```

```
     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
```

(a)   (b)

Figure 14: Digit classification network structure and summary used in the Colour Channel Concatenated Digits MNIST task

I.2   Multiplication Weight Trajectories

This section plots the trajectories for the two learnable weights used when calculating the multiplication operation for each fold. The baseline, which uses a solved multiplier, will not have any learnable weights for multiplication hence has no trajectory plot. The remaining models i.e., FC, NMU, sNMU do have multiplication weights to learn and for all cases the true solution of the weights is $[1,1]$.

Isolated digits. Figure 15. 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.

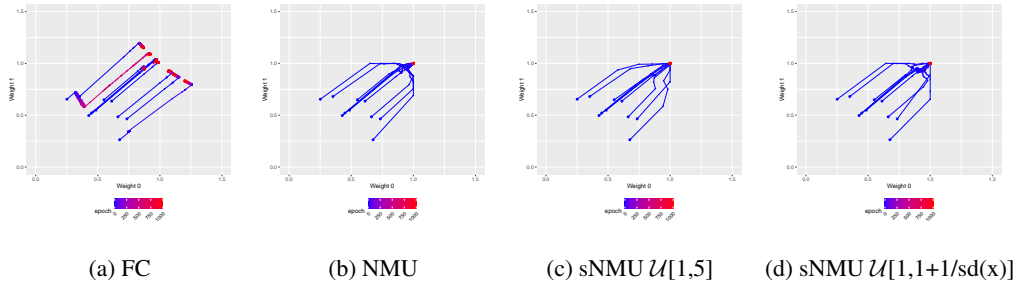(a) FC   (b) NMU   (c) sNMU $\mathcal{U}[1,5]$ (d) sNMU $\mathcal{U}[1,1+1/sd(x)]$

Figure 15: Path learnt by the weights for the multiplication network. Each path represents a different seed. Blue and red asterisks represent the starting and ending points respectively.

Colour Channel Concatenated Digits. Figure 16. When the task difficulty of classifying the digits is increased, the NMU is also unable to converge, while the sNMUs still can converge for all folds. The FC models are again unable to reliably converge to the true solution.
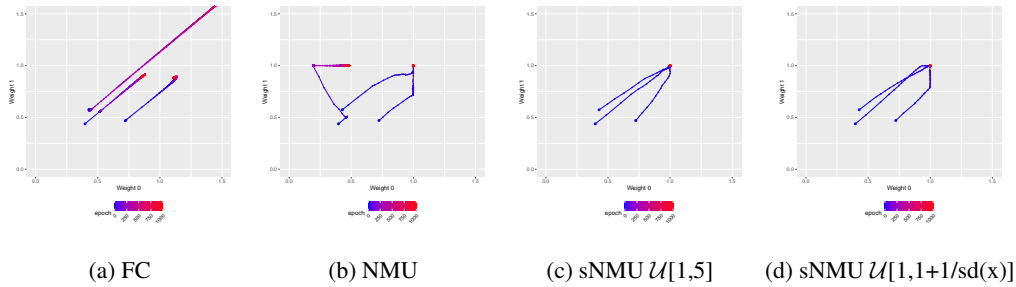

(a) FC   (b) NMU   (c) sNMU $\mathcal{U}[1,5]$ (d) sNMU $\mathcal{U}[1,1+1/sd(x)]$

Figure 16: Path learnt by the weights for the multiplication network. Each path represents a different seed. Blue and red asterisks represent the starting and ending points respectively.

I.3   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

21

each digit over the test set is plotted with a further breakdown of the decisions over each digit using a unnormalised confusion matrix. The distribution of digit labels will be non-uniform.

Isolated digits. Figures 17 and 18. 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.

Figure 17: 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) Full plot from success rate 0-1.

(a) Baseline (MUL)　　　　　(b) FC　　　　　(c) NMU

(d) sNMU $\mathcal{U}[1,5]$　　　　　(e) sNMU $\mathcal{U}[1,1+1/sd(x)]$

Figure 18: Confusion matrices for intermediate label classification

Colour Channel Concatenated Digits. Figures 19 and 20. Results are shown for a fold which models found especially challenging. Only the baseline and the sNMU using batch statistics are able to learn a 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.
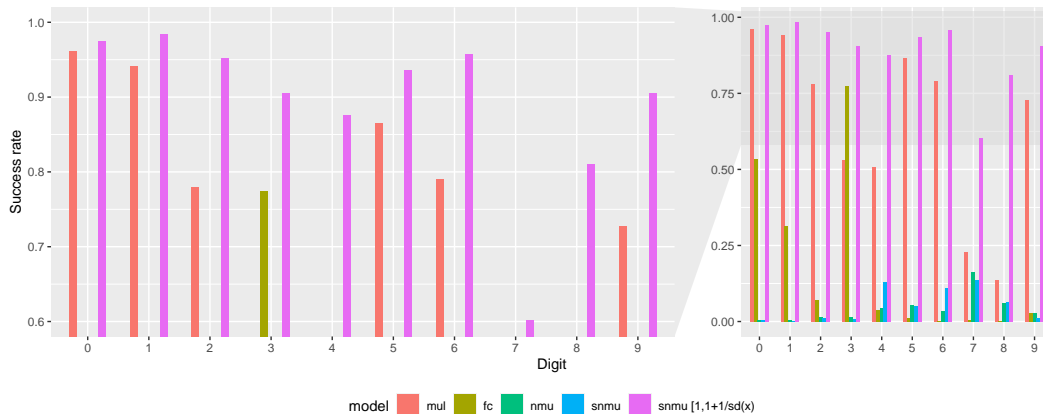
Figure 19: Success rates for classifying each digit (with rounding) in the test dataset for a single seed. (Left) Zoom-in for success in range 0.95-1. (Right) Full plot from success rate 0-1.
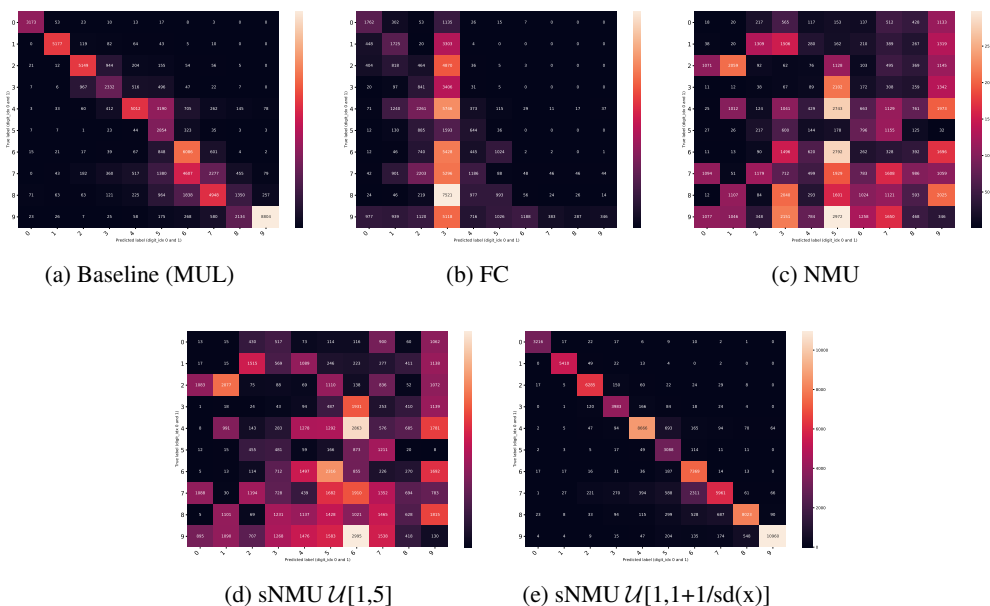

(a) Baseline (MUL)

(b) FC

(c) NMU

(d) sNMU $\mathcal{U}$[1,5]

(e) sNMU $\mathcal{U}$[1,1+1/sd(x)]

Figure 20: Confusion matrices for intermediate label classification.

I.4 Digit Classification Accuracy over Epochs

This section shows how accuracies for classifying each digits evolves over the epochs. The average values over all folds are shown (with 95% confidence intervals).

Isolated digits. Figure 21 shows similar leaning for both digits, with sNMU modules providing small confidence bounds. The sNMU with range $\mathcal{U}$[1,5] also shows better accuracy over the baseline after approximately 600 epochs.

Colour Channel Concatenated Digits. Figure 22 shows a greater variation of performance over the different models and the different folds in comparison the the isolated digits' results. The difference can be explained by the increased difficulty for 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).
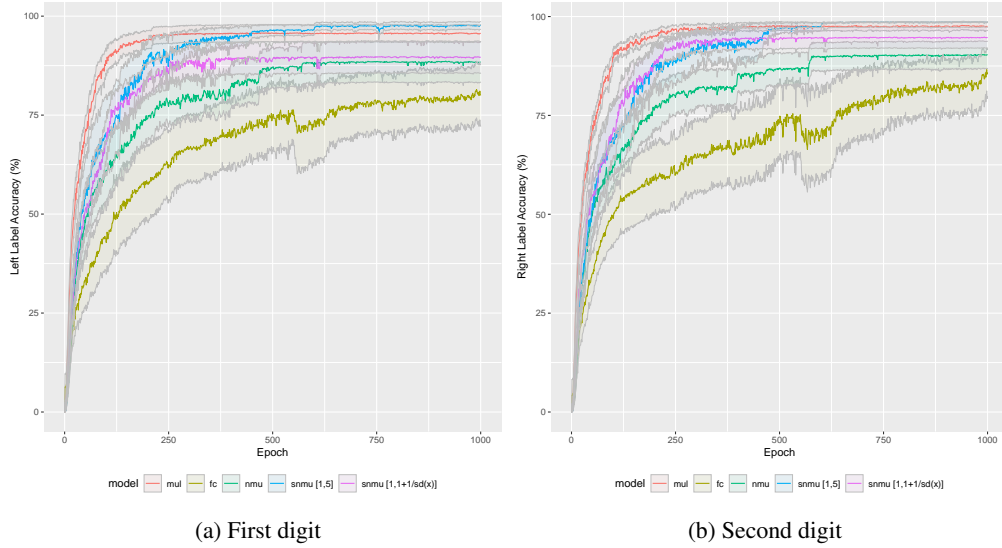
(a) First digit   (b) Second digit

Figure 21: Label accuracy vs epoch of the two digits for the Isolated digit variant of the Static MNIST Product task.

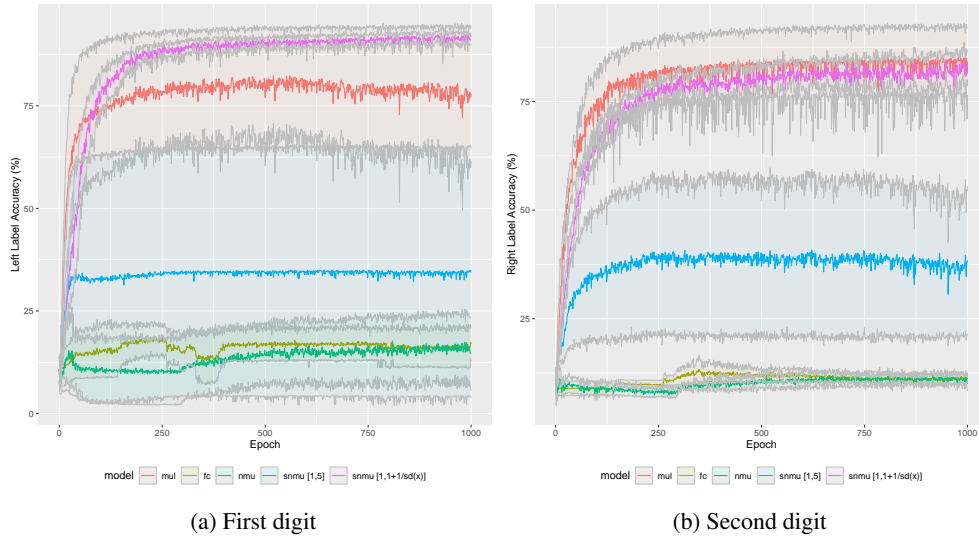(a) First digit   (b) Second digit

Figure 22: Label accuracy vs epoch of the two digits for the Colour channel concatenated digit variant of the Static MNIST Product task.

J   Alternate Stochastic Methods

We explore how other forms of stochasticity can effect learning of a NALM. The two methods are gradient noise and stochastic gating.

J.1   Gradient Noise

Rather than implicitly altering the gradients by using reversible stochasticity, we see if the altering the gradients of a NALM explicitly can improve learning. Following Neelakantan et al. [2015b], 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 η is a scaling factor.

24

J.2 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, we represent a NMU weight as a learnable stochastic gate Yamada et al. [2020]. 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, μ_i, is initialised to 0.5. 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 ϵ_i is noise sampled from $N(0, \sigma^2)$ and $\sigma = 0.5$. During training noise is used but during inference no noise is added. L0 regularisation is applied by taking the probability that the gates are active, which can be calculated using a standard Gaussian CDF i.e., $\Sigma_{i=1}^{I} \Phi(\frac{\mu_i}{\sigma})$. To balance the regularisation with the main loss objective, the regularisation is scaled by a pre-defined lambda.

J.3 Single Layer Task

Figure 23 shows that using stochastic gates for the NMU weighs can be beneficial if the lambda is sufficiently small (0.1 or under). When small lambdas are used, the stgNMU can improve on the remaining two ranges which the NMU struggles to obtain full success.
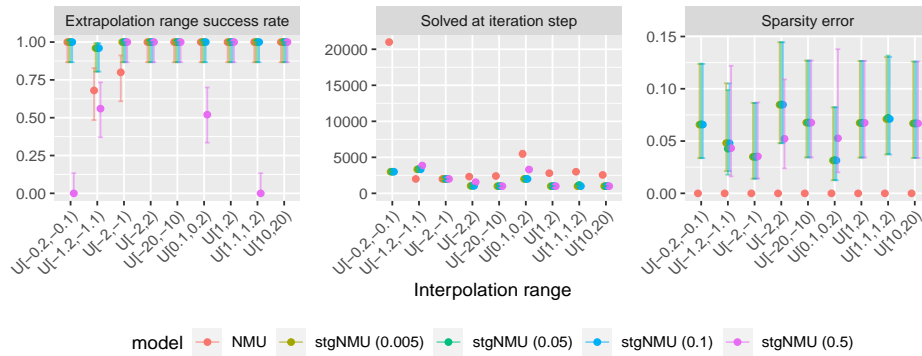

Figure 23: Single Layer Task results for using stochastic gating to learn the NMU weights over different λ.

Figure 24 shows that adding gradient noise does not effect the success rate in comparison to the NMU. If larger gradient noise is used (i.e. eta=10) then the success can degrade (see U[-0.2, -0.1]). Furthermore, the convergence speeds get slower with gradient noise.
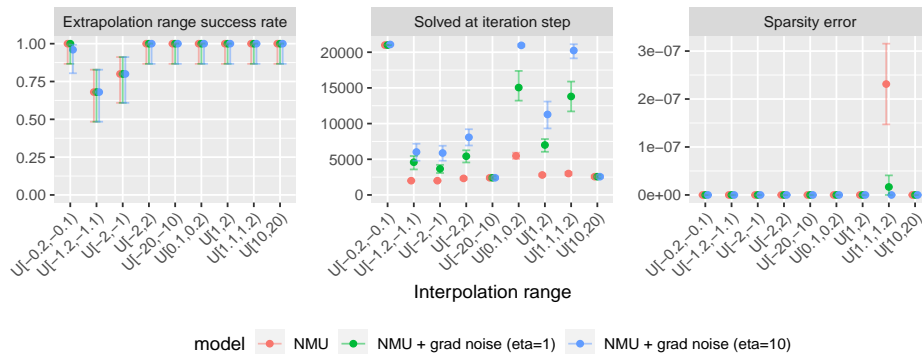

Figure 24: Single Layer Task results for using stochastic gating to learn the NMU weights over different η.

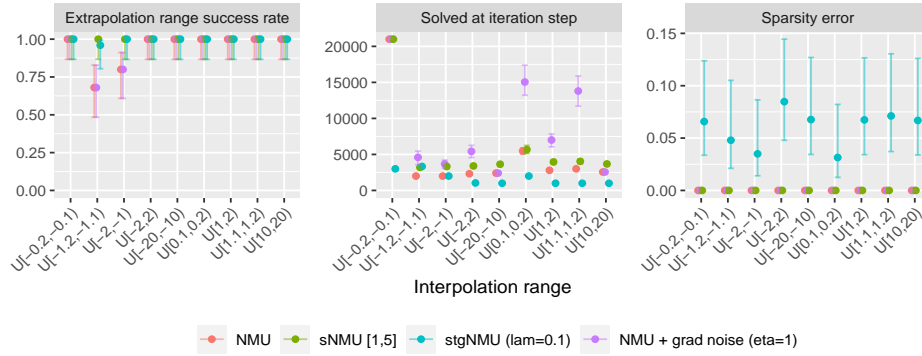Figure 25 shows the three different noise methods compared against the baseline NMU.

Figure 25: Single Layer Task results 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).

J.4 Arithmetic Dataset Task

Taking the best hyperparameters from the single layer task and training on the two layer Arithmetic dataset task results in Figure 26 which shows that both the stgNMU and gradient noise hurt performance.
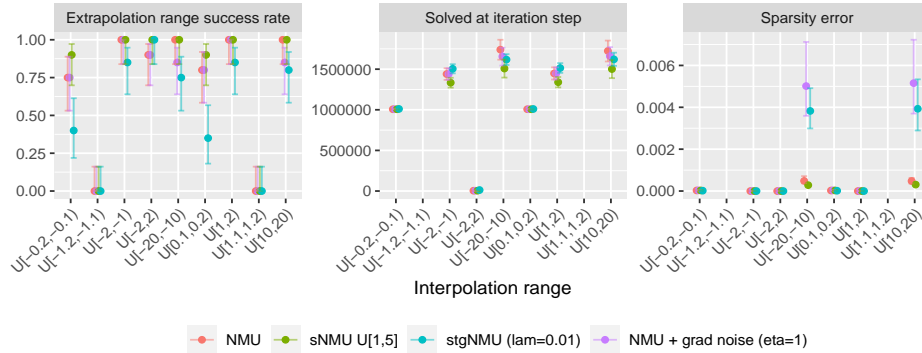

Figure 26: Arithmetic Dataset Task comparing the NMU to NMUs trained with the following types of stochastic methods: reversible stochasticity (sNMU), stochastic gate weights (stgNMU) and gradient noise (NMU + grad noise).