Neural Status Registers

Lukas Faber [1] **Roger Wattenhofer** [1]

Abstract

We study the problem of learning comparisons between numbers with neural networks. Despite comparisons being a seemingly simple problem, we find that both general-purpose models such as multilayer perceptrons (MLPs) as well as arithmetic architectures such as the Neural Arithmetic Logic Unit (NALU) struggle with learning comparisons. Neither architecture can extrapolate to much larger numbers than those seen in the training set. We propose a novel differentiable architecture, the Neural Status Register (NSR) to solve this problem. We experimentally validate the NSR in various settings. We can combine the NSR with other neural models to solve interesting problems such as piecewise-defined arithmetic, comparison of digit images, recurrent problems, or finding shortest paths in graphs. The NSR outperforms all baseline architectures, especially when it comes to extrapolating to larger numbers.

1. Introduction

Mathematical reasoning is a cornerstone for intelligence. In this paper, we study an innocent, yet critical part: comparisons. Nearly all species intuitively use comparisons, for example, to find the bigger of two food sources. Also, humans subconsciously compare all the time: Even at a glance, we can see whether two people have the same height or if one person is taller. On the other hand, we also consciously use comparisons in computer programs, for example in `if` and `while` statements. Given how ubiquitous comparisons are, we study how well neural networks can model them.

We find that common neural networks like multilayer perceptrons struggle with comparisons. While they can learn a comparison over the training set, these networks cannot *extrapolate*. A model can extrapolate if it can also solve comparisons far beyond the training set. A model failing to extrapolate suggests that the model did not fully understand how to compare the numbers but at least partially relied on artifacts in the training set. Furthermore, other popular arithmetic architectures such as the NALU (Trask et al., 2018) or NAU (Madsen & Johansen, 2020) cannot extrapolate either.

To address this problem, we propose a novel differentiable neural architecture, the Neural Status Register (NSR). The NSR mimics status registers (or condition code registers) that we can find in virtually all hardware processors. When a processor executes the code line `if x == y:`, it first subtracts y from x and sets certain condition code bits. We are interested in two bits for the NSR: the *sign bit* (if the difference is negative) and the *zero bit* (if the difference is 0). To evaluate `x == y`, for example, we can simply read the zero bit after the subtraction $x - y$. The NSR transfers this idea into a differentiable neural architecture that learns (i) to pick the right x, y from the input and (ii) which comparison to evaluate.

In isolation, we can use the NSR to learn comparisons robustly. However, it is more interesting to plug the NSR into a larger architecture to solve problems that require comparisons. For example, we combine the NSR with GNNs which benefit from a model that can find the minimum from a set of numbers to compute shortest paths. We summarize our contributions as follows:

- We introduce the NSR as novel neural architecture for comparisons. The NSR is a differentiable version of status registers found in processors that allows end-to-end training with gradient descent or in combination with other neural architectures. The NSR shows strong extrapolation, even to numbers many orders of magnitude larger than the training set.
- We analyze the gradient space of the NSR to propose optimizations to make the training more stable and efficient. We show how to choose the NSR components to obtain good gradient signals. We further propose adding some redundancy to make the NSR less initialization dependent. We experimentally validate the efficacy of these changes in an ablation study.
- We experimentally evaluate the NSR in a variety of tasks, mainly in conjunction with other architectures. This allows learning interesting tasks such as learning

[1]ETH Zurich, Switzerland. Correspondence to: Lukas Faber <lfaber@ethz.ch>, Roger Wattenhofer <wattenhofer@ethz.ch>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

simple comparisons, piecewise-defined functions, comparing images of digits, recurrent problems, shortest paths in graphs. The NSR outperforms baseline comparison architectures on all tasks and shows remarkable extrapolation.

2. Related Work

2.1. Mathematical Reasoning in Neural Networks

Our proposed NSR is an architecture that allows neural networks to do mathematical reasoning. Prior works exist which we roughly divide into two subgroups: neuro-symbolic versus quantitative models. Neuro-Symbolic models encode numbers into symbols (for example a sequence of digits) and reason over those symbols. Quantitative models instead have neurons that hold the number itself. The NSR belongs mostly to the latter group.

Neuro-Symbolic Models. One example approach is to encode numbers as a sequence of symbols in text and use text-based architectures such as transformers for reasoning. Saxton et al. (2019); Lample & Charton (2020); Lewkowycz et al. (2022) show that a tailored language model can achieve impressive results for a variety of tasks. Even general large language models such as GPT3 (Brown et al., 2020) can solve some mathematical problems. Another idea is to align symbols of numbers in grids and use convolutions over them. This approach is taken by NeuralGPUs (Kaiser & Sutskever, 2016) and their improvements by Freivalds & Liepins (2017). Kim et al. (2021) also learn to reorganize sequences (of symbol-encoded numbers) into grids to solve with convolutional architectures.

A set of related works in neuro-symbolic reasoning developed differentiable read-write memory modules (Graves et al., 2014; Weston et al., 2015; Zaremba & Sutskever, 2016; Zaremba et al., 2016; Graves et al., 2016; Le et al., 2020), or stacks (Grefenstette et al., 2015). We can then use this memory to learn algorithms similar to imperative programs. Some example algorithms we can learn are addition or sequence reversion. But other approaches for neuro-symbolic algorithm inference that follow different programming paradigms are possible. For example, some previous works use compositions of simple instructions Reed & de Freitas (2016); Li et al. (2017); Chen et al. (2018) or recursion Cai et al. (2017); Feser et al. (2017) which resembles functional programming. Evans & Grefenstette (2018); Dong et al. (2019) follow the ideas of declarative programming and learn logical reasoning.

Quantitative Models. Quantitative models have neurons holding the actual quantity, that may be encoded. Quantitative architectures then aim to learn mathematical operations on the quantities, for which conventional neural architectures often fail (Mistry et al., 2022). One seminal work are the Neural Arithmetic Logic Units (NALUs) by (Trask et al., 2018) that can compute addition and subtraction. NALUs further support multiplication and division by transforming the operands into logspace before adding and subtracting and back to the original space after. However, these operations are error-prone when facing potentially negative inputs. The authors further report unstable results for division. Schlör et al. (2020) propose improvements to the NALU, for example for robustness to negative inputs and divisions. Madsen & Johansen (2020) propose the Neural Arithmetic Unit (NAU), which comes with more robust initialization and training than the NALU. Heim et al. (2020) improve upon the NAU by allowing complex numbers and create the Neural Power Unit (NPU). The NPU can handle division and other power functions such as square roots. However, we found that none of these methods can reliably solve comparisons. The NSR is tailored to solve comparisons and can fill this gap.

2.2. The Importance of Extrapolation

Most of the above methods share their definition of success: A model is successful only if it can also solve out-of-distribution examples and/or extrapolate. Extrapolation is a special form of out-of-distribution sampling where the samples also become harder to solve. For example, we can create extrapolating test instances for neuro-symbolic sequences by increasing the sequence length, we can make arithmetic tasks harder by adding more digits to numbers (Kim et al., 2021). In quantitative architectures, we can sample larger numbers as inputs. Madsen & Johansen (2020) and Heim et al. (2020) train some problems on the range $[-2; 2]$ but evaluate on a larger range minus the training part, for example, $[-6; 6] \setminus [-2; 2]$.

Extrapolation is a desirable property since it suggests the model learned the real underlying problem. For example, if a model learned the true algorithm for digit-wise addition, it should not matter how many digits the inputs have. On the other hand, a model failing to extrapolate suggests that the model failed to learn the true solution. This idea of extrapolation exists in other fields of deep learning (Santoro et al., 2018) or reinforcement learning (Martius & Lampert, 2017; Sahoo et al., 2018). Extrapolation and out-of-distribution also exist in natural language processing (Lake & Baroni, 2018) or computer vision where they are often referred to as zero-shot learning(Xian et al., 2017).

Xu et al. (2021) investigate when neural architectures are likely to exhibit promising extrapolation. Neural architectures whose architecture algorithmically aligns with a given problem will generally be both more training data efficient and extrapolate better. For example, GNNs align well with the Bellman-Ford algorithm which makes them good at learning shortest paths (Xu et al., 2020).

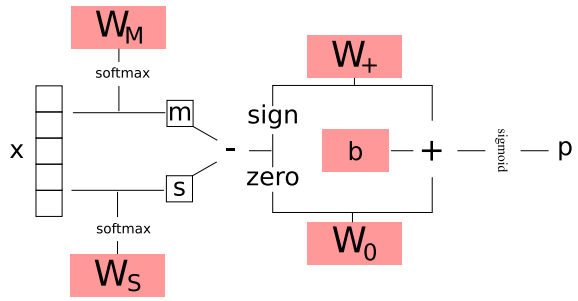


Figure 1. High-level architecture for the NSR. The NSR mimics physical status registers/condition code registers by subtracting the numbers to compare and analyzing if the subtraction returned a positive number or zero. The NSR has two sets of learnable parameters, highlighted in red. The first set W_M, W_S learns which parts of the inputs should be compared. The second set b, W_+, W_0 learns which comparisons to use.

We will show that quantitative architectures can learn comparisons but fail to extrapolate to much larger numbers. On the other hand, the NSR is well aligned with the comparison problem. Therefore the NSR can even extrapolate to numbers that are many orders of magnitude larger than those in the training set.

3. Neural Status Registers

3.1. The Basic NSR Model

In this section, we introduce the NSR model. The NSR learns to model functions of the form $f(\boldsymbol{x}) = p \in [0; 1]$ where p is the probability of some comparison (from $>, \geq, =, \neq, \leq, <$) being true. The NSR learns this comparison by following status registers/condition code registers: It learns a minuend m and a subtrahend s from $\boldsymbol{x}$, subtracts them, and applies a function approximating the sign and zero bits. We design the NSR to neither assume which numbers in $\boldsymbol{x}$ are important nor which comparison it needs. Instead, the NSR resolves both via learnable parameters. With two sets of parameters W_M and W_S, which we activate with softmax, the NSR learns the minuend and the subtrahend of the comparison. The second set of parameters b, W_+, and W_0 weigh the activation of the sign and zero-bit approximations and thus learn the comparison. The weighed activation of these bits is activated with σ to produce p. Figure 1 illustrates this flow, with learnable parameters highlighted in red. We can write the update mathematically as:

$$
\begin{aligned}
m &= \langle \boldsymbol{x}, softmax(W_M) \rangle \\
s &= \langle \boldsymbol{x}, softmax(W_S) \rangle \\
p &= \sigma(b + W_+ \hat{S}(m - s) + W_0 \hat{Z}(m - s))
\end{aligned}
$$

The learnable weights b, W_M, and W_0 and the intermediate values m, s receive the following gradients with respect to the output p:

$$
\begin{aligned}
\frac{\partial p}{\partial b} &= p(1-p) \\
\frac{\partial p}{\partial W_+} &= p(1-p)\hat{S}(d) \\
\frac{\partial p}{\partial W_0} &= p(1-p)\hat{Z}(d) \\
\frac{\partial p}{\partial m} &= p(1-p)(\hat{S}' W_+ + \hat{Z}' W_0) \\
\frac{\partial p}{\partial s} &= p(1-p)(\hat{S}' W_+ + \hat{Z}' W_0) \cdot (-1)
\end{aligned}
$$

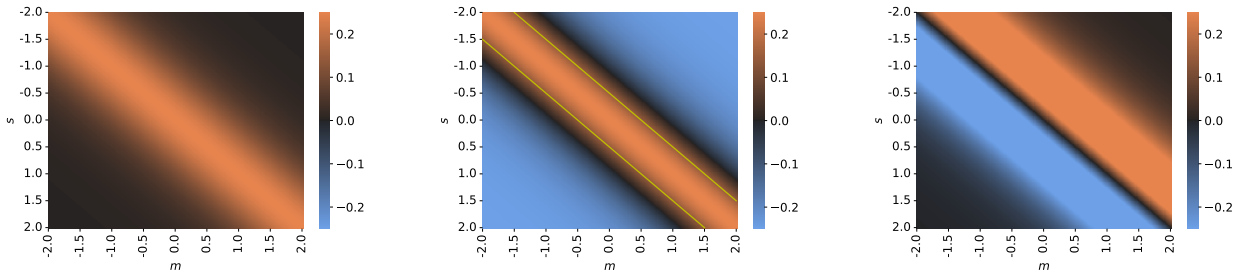
The last two equations require the derivative of the sign and zero functions. This means that we cannot use bits from physical status registers. We need to approximate the bits with continuous functions. We have to be thoughtful about which continuous approximations to use. Let us for example consider $\hat{Z} = 1 - \tanh(m - s)^2$ as an approximation for the zero bit. Figure 2a shows the gradient landscape for W_0 for different values of m and s. Since most numbers are not equal, $\hat{Z}$ approximates 0 in most of the space which causes a vanishing gradients signal for W_0.

To prevent this issue, we slightly alter the sign and zero-bit definitions to return 1 as the `true` value, but -1 as the `false` value. We show the gradient landscape for the modified function $\hat{Z} = 1 - (2\tanh(m - s)^2)$ in Figure 2b which now has a non-zero gradient almost everywhere.

With continuous $\hat{S}$ and $\hat{Z}$ we have one more issue to overcome. There will be differences $m - s$ where $\hat{S} \approx 0$ and neither $+1$ nor -1 (we will choose a function where this happens for $m \approx s$). As a consequence, the gradients for W_+ that have $\hat{S}$ as a factor will vanish. Equally, there must be some m, s, such that $m \neq s$ but $\hat{Z}(m, s) > 0$. In these cases, we say the difference is *below the NSR's resolution limit*. For example, the definition of $\hat{Z}$ in Figure 2b have a resolution limit of ≈ 0.88 (the smallest difference $m - s$ where $\hat{Z}$ is still positive). The two yellow highlight a difference of only 0.5. This difference falls in the orange region where the NSR considers the numbers equal. We say that differences of 0.5 are out of the resolution limit.

We can overcome these resolution issues by applying $\hat{S}$ and $\hat{Z}$ not directly on $m - s$, but scale this difference by a factor λ. If $\lambda > 1$, we can lower the resolution limit and handle very small differences between numbers. If we choose $\lambda < 1$, we can compensate for very large differences and prevent vanishing gradients. Ultimately, we propose the following approximations. Appendix B plots these functions for illustration.

$$
\begin{aligned}
\hat{S}(d) &= \tanh(\lambda(m - s)) \\
\hat{Z}(d) &= 1 - 2 * \tanh(\lambda(m - s))^2
\end{aligned}
$$



(a) Gradient landscape for W_0 for $\hat{Z} = 1 - \tanh(m-s)^2$. The gradients are mostly zero except for when $x_1 \approx x_2$

(b) Gradient landscape for W_0 for the improved $\hat{Z} = 1 - 2\tanh(m-s)^2$. Gradients are almost nonzero everywhere.

(c) Gradient landscape for m with $\hat{S} = \tanh(m-s), W_0 = 0, W_+ = 1$. Gradients tend towards 0 as $|m-s|$ grows.

Figure 2.

Another gradient problem might appear when the difference between two numbers $m-s$ becomes large. Figure 2c shows the derivative of m, assuming $\hat{S} = \tanh(m-s), W_0 = 0$, and$W_+ = 1$. The larger the differences $x_1 - x_2$ the more $\hat{S}$ and $\hat{Z}$ saturate. In turn, their derivatives approach 0 which might cause vanishing gradients for W_M and W_S. Using $\lambda < 1$ to downscale the difference can also combat this problem.

For simplicity, we will initially experiment with integers that require a resolution limit of at most 1. We later analyze the performance of different resolution limits δ versus different scale factors λ in Section 4.6. We also investigate the potential issue of vanishing gradients for W_M and W_S, however, we do see this problem materializing in practice.

3.2. The Redundant NSR Model

Next, we investigate how to initialize the NSR robustly. Bland (1998) did an initial study on which parts of the parameter space in an MLP allow for learning XOR (the bitwise version of $\neq$). Not the whole space leads to a solution. Frankle & Carbin (2019) obtain similar results experimentally. They further report that wider models with redundant neurons substantially increase the chance of good initial values (that the authors call lottery tickets) and therefore learning success. We also find this idea in Kaiser & Sutskever (2016). The authors report that creating redundant sets of parameters, which eventually are forced to converge, drastically helps the stability of their model. Based on these works, we measured the success probability of learning `x == y` with the NSR, searching over initial values for W_+ and W_0 while initializing W_M and W_S glorot uniform (Glorot & Bengio, 2010). While the NSR generally learns equality for most of the weight space, there are cases where a low value for W_0 lead to failure. This seems plausible since NSR requires positive weights in W_0 to express equality. Nevertheless, we can get unlucky initial weights for the NSR. Following previous work, we reduce the chance of such bad initialization by adding redundancy to the NSR.

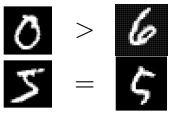
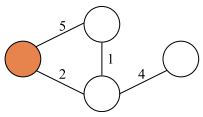
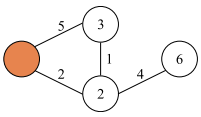
We can create redundancy by having multiple instances of W_M, W_S, W_+, and W_0. Practically, we can extend the learnable weights to matrices instead of vectors (for W_M, W_S) and vectors instead of scalars (for W_+, W_0). We additively combine all redundant parts with the bias term. Then, we activate the sum with sigmoid to produce p. We experimented with a regularizing loss similar Kaiser & Sutskever (2016) that forces the redundant structures eventually collapse in the end to one solution. We found no improvements with such a regularization and removed it again for simplicity.

4. Experiments

Code for all experiments is available[1]. We experimentally try the NSR in five different settings. First, we use a vanilla NSR to learn comparisons between numbers. Second, we learn piecewise-defined functions with two branches. We wire the NSR to two arithmetic units such as NALU or NAU that each learn a branch. The NSR learns the comparison directly on the input and gates which of the two branches to take. Third, we learn image comparisons. We use a standard convolutional neural network to predict the digit in the image. The NSR takes two such predictions and learns a comparison of the numbers. The entire architecture (CNN+NSR) is trained from scratch end-to-end. Fourth, we use the NSR in a recurrent architecture to find the minimum element or count the first element in a sequence of numbers. For finding the minimum, the NSR compares an internal state with the next input number and controls to what extent to keep the internal state or the new number. For counting, the NSR compares the next input number to the first number

[1] https://github.com/lukasjf/neural_status_register

Table 1. Overview of the experiments. *Comparisons*: A vanilla NSR learns to compare numbers. *Functions*: The NSR gates downstream arithmetic units (NAU, NALU) to compute piecewise-defined functions. *MNIST*: An upstream CNN feeds image predictions into an NSR to compare MNIST digits. *Recurrent* The NSR operates on a list of numbers to find the minimum or count a number's occurrences. *SSSP*: The NSR is used instead of an aggregation function inside a GNN to find shortest paths.

Task	Example input	Example output	Extrapolation
Comparisons	$10 > 5$ $1 > 5$	1 0	Larger numbers
Functions	$f(11, 10, 22, 44, 62)$ $f(10, 11, 22, 44, 62)$ $g(10, 10, 22, 44, 62)$ $g(11, 10, 22, 44, 62)$	66 22 66 22	Larger numbers for comparisons
MNIST		0 1	N/A
Recurrent	$\min([3, 5, 2, 7, 1])$ $count([3, 3, 5, 2, 3, 1])$	1 2	Longer sequences and/or larger numbers
SSSP			More nodes and/or larger edge weights

to control a "+1"-counter if it should increment or not. Last, we combine the NSR with graph neural networks (GNNs) to learn shortest paths. The NSR is replacing the aggregation function in the GNN, so it receives several input messages and has to reduce them to one embedding. In the case of finding shortest paths, the NSR should learn to find the minimum. Table 1 also shows an overview of the different tasks with examples.

In addition to the NSR and other arithmetic architectures, we attempt to solve these problems with a large generative language model for which we picked ChatGPT[2] with the default hyperparameter settings. We experimented with the following prompts

Comparison Which of the numbers x_1 and x_2 is larger

Functions Let $f(a, b, c, d, e)$ be a piecewise defined function that computes $e + 4$ if $a > b$ and $d - c$ otherwise. What is $f(x_1, x_2, x_3, x_4, x_5)$

Minimum What is the smallest number in $[x_1, x_2, \dots]$

Counting How many times does x_1 occur in $[x_2, x_3, \dots]$

Shortest Paths I am describing to you a graph in a set of triples (a,b,c) of undirected edges. a and b are node IDs and c is the distance between the two nodes. The edge list is $[e_1, e_2, \dots]$. Can you compute the lengths of shortest paths from v to all other nodes. You can skip the intermediate steps and only provide the final result

[2]https://openai.com/product/chatgpt

ChatGPT solved all instances of the first three problems The solutions also extrapolated to larger numbers. On the other hand, ChatGPT often failed on the counting task, especially as sequences became longer. ChatGPT also failed to compute shortest paths though most distances were in the right ballpark. We found the final instruction necessary to ensure the reply does not exceed the output size. We also experimented with encoding MNIST images in text but found no encoding that ChatGPT understood. This small study shows the impressive progress general-purpose symbolic architectures made on arithmetic tasks but it also demonstrates that quantitative architectures are still the better choice for some tasks.

4.1. Comparisons

Our first experiment is the seemingly simple task of learning comparisons between numbers. We sample integers from $[-10, 9]$ and compare them with $>, \geq, =, \neq, \leq, <$. The target variable is 1 if the comparison evaluates to `true` and 0 otherwise. For $>, \geq, \leq$, and $<$ we train with all pairs of digits. For $=$ and $\neq$ we create balanced training sets as follows: For every $i \in [-10, 9]$ we add (i, i) and (i, j) for one $j \neq i$ to the training set. After training, we test models in an extrapolation setting: We take pivot numbers $x = 10^i, i \in [1, 7]$ and compose the testing set of all pairs $(x, x + o), o \in [-5, 5]$ for $>, \geq, \leq$, and $<$. For $=$ and $\neq$

Table 2. Learning comparisons between numbers. For each column in the table, the column header is compared with every integer number with difference at most 5. Table entries denote the mean absolute error for model predictions (lower numbers are better).

Comparison	Model	10^1	10^2	10^3	10^4	10^5	10^6	10^7
$>$	MLP	0.00 ±0.00	0.00 ±0.00	0.01 ±0.01	0.23 ±0.16	0.49 ±0.12	0.52 ±0.01	0.52 ±0.01
	NPU	0.75 ±0.19	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02
	NALU	0.00 ±0.00	0.00 ±0.00	0.11 ±0.06	0.50 ±0.09	0.52 ±0.00	0.52 ±0.00	0.52 ±0.00
	NAU	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	NSR	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
$=$	MLP	0.18 ±0.16	0.28 ±0.17	0.47 ±0.14	0.52 ±0.05	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	NPU	0.40 ±0.15	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	NALU	0.45 ±0.10	0.45 ±0.10	0.48 ±0.08	0.52 ±0.07	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	NAU	0.33 ±0.11	0.33 ±0.11	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12
	NSR	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00

we again create a balanced testing set by using $o = 0$ and one other non-zero offset from $[-5, 5]$. The reason for these pivot plus offset sets is to test the models on the hardest predictions right at the decision boundary.

We train five different models on this problem: a normal two-layer MLP, Neural Power Units (NPU) (Heim et al., 2020), Neural Arithmetic Logic Units (NALU) (Trask et al., 2018), Neural Arithmetic Units (NAU) (Madsen & Johansen, 2020), and our Neural Status Register (NSR). For all models but the NSR, we add hidden units that we combine with a linear readout to create a fair setup in terms of available parameters. We supervise, in this and all subsequent experiments, with the mean absolute error and use the Adam (Kingma & Ba, 2015) optimizer with default settings. All results are averaged over 10 different seeds. We train for 50000 epochs and then test around each pivot element separately. Table 2 shows the results for $>$ and $=$, for space reasons we defer the full comparison table to Appendix C.

NPUs excel at power functions but are misaligned with comparing numbers which require subtractions. Therefore NPU struggles with the problem. While MLP and NALU can solve $>$ on the training range, they cannot extrapolate. NAU can solve $>$ and extrapolate but cannot solve $=$. The NSR outperforms all other methods and is the only method capable of learning and extrapolating in both problems.

4.2. Piecewise-defined Functions

In this experiment, we combine the NSR with arithmetic architectures to perform conditional calculations. We learn to compute the two piecewise-defined functions.

$$f(a, b, c, d, e) = \begin{cases} e + 4 & \text{if } a > b \\ d - c & \text{otherwise} \end{cases}$$

$$g(a, b, c, d, e) = \begin{cases} e + 4 & \text{if } a == b \\ d - c & \text{otherwise} \end{cases}$$

We sample values for a and b in the same way as for the previous experiments, for training and testing sets. The remaining numbers c, d, and e are randomly sampled from $[-100, 100]$. We wire the comparison architectures from the previous experiment to two arithmetic units. One unit is weighted by the comparison output score, the other with its inverse property. We experiment with NALU and NAU as arithmetic architectures. However, our experiments show using NALU as arithmetic architecture works better in all instances. We report the results with NAU to Appendix D. We train for 500000 epochs and report the results in Table 3.

We hypothesize that due to the clipping of weights and zeroing gradients, NAU makes it harder for upstream units to also receive meaningful gradients. The NSR proves to be the superior comparison unit for this task, providing among the best extrapolation for f. However, the NSR is the only architecture to solve g.

4.3. Digit Comparison with CNNs

In this task, we combine the NSR with convolutional neural networks (CNNs). The base architecture is the CNN sample implementation for MNIST from pytorch.[3] Instead of supervising the output with the target image, we feed two images through the CNN, interpret the model outputs as numbers and input these numbers into a NSR or comparable

[3] https://github.com/pytorch/examples/blob/234bcff4a2d8480f156799e6b9baae06f7ddc96a/mnist/main.py

Table 3. Learning piecewise-defined functions. The condition is between the first two parameters. One parameter is the column header the other parameter can be integers with difference at most 5. The remaining numbers are sampled from $[-100, 100]$. Table entries denote the mean absolute error for model predictions.

Function	Model	2^3	2^4	2^5	2^6	2^7	2^8	2^9
f	MLP	0.00 ±0.00	0.00 ±0.00	0.01 ±0.00	0.01 ±0.00	0.96 ±2.72	3.92 ±4.69	3.95 ±5.57
	NALU	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.01 ±0.00	0.02 ±0.03	0.30 ±0.81	2.77 ±4.77
	NAU	0.00 ±0.00	0.01 ±0.00	0.01 ±0.00	0.01 ±0.00	0.02 ±0.01	0.03 ±0.01	0.06 ±0.03
	NSR	0.00 ±0.00	0.00 ±0.00	0.01 ±0.00	0.01 ±0.00	0.01 ±0.01	0.02 ±0.01	0.05 ±0.06
g	MLP	4.78 ±8.78	5.81 ±9.18	4.71 ±8.52	5.42 ±8.73	4.26 ±6.55	12.72 ±13.34	25.93 ±16.09
	NALU	34.51 ±10.76	36.23 ±13.06	36.75 ±14.96	34.85 ±15.66	33.97 ±14.21	32.75 ±11.56	40.14 ±18.70
	NAU	33.86 ±11.40	37.59 ±11.18	35.92 ±13.19	37.52 ±13.16	40.80 ±16.17	42.15 ±24.79	57.77 ±37.23
	NSR	0.04 ±0.03	0.04 ±0.04	0.12 ±0.19	0.08 ±0.11	0.06 ±0.05	0.08 ±0.08	0.21 ±0.30

Table 4. Classification results on image comparison tasks. For Image$>$, models need to predict if the first image shows a larger digit than the second, for Image$=$ if they are equal. We train a CNN plus a comparison model together and only on the comparison signal. The table shows the test accuracy for different comparison architectures on their best epoch over 10 seeds.

Model	Image$>$	Image$=$
MLP	95.18±0.78	70.79±5.25
NALU	75.90±20.36	64.99±0.97
NAU	76.01±20.44	63.26±1.68
NSR	96.92±1.06	80.05±12.07

comparison architecture. The label is whether the number in the first image is larger/equal to that of the second. We supervise the entire architecture including the CNN only with this binary signal. Initially, the CNN will not be able to distinguish numbers, therefore we decrease the resolution limit of the NSR by setting $\lambda = 10$.

We create the data from normal MNIST batches: for $>$, we pair all images in a batch of 50 images. For $=$, we pair all images with the same digit within batches of 100 images. For every such pair we add an non-equal pair. Table 4 shows the test accuracy for the best epoch per model for both image comparison tasks.

Unsurprisingly, accuracy is lower than in a direct digit classification task (which reaches around 99%). To compare two digits, we need to identify the digits in the images (as in the classification task), but the supervision signal is much weaker. In vanilla MNIST, we supervise an image of a 4 with the label 4. In the image comparison task, we need to compare the image 4 with at least one image of almost every other digit to know if it is a 4. NALU und NAU produce mediocre results compared to MLP and NSR. The NSR produces better results than MLP, especially for the equality comparison.

4.4. Recurrent Computation and Control.

In this experiment, we use the NSR and comparable comparison architectures in recurrent problems. We look at two problems, where we input sequences of numbers. We process numbers from the input sequence recurrently one at a time. In the first problem *min*, comparison architectures compare the hidden state versus the new input and learn to identify the minimum element. In the second problem *count*, comparison architectures compare sequence elements versus the initial sequence element. On equality, the method learns to increment a counter to count the number of occurrences of that first element.

For training we create 400 sequences of length 5 (for min) or 6 (for count) with numbers sampled from $[1, 10]$. During testing, we create 50 test sequences that we independently extrapolate in two dimensions: We increase the upper limit of the sampling range by powers of two and also increase the sequence length up to 50 and 51, respectively. For min, we sample numbers from the entire interval. For count, we first sample a random pivot number to count and sample the remaining numbers such that they have a difference of at most 5 to this pivot. In addition to the previous models, we also run an LSTM (Hochreiter & Schmidhuber, 1997) baseline. Previous works have shown that LSTMs can also learn counting in some scenarios (Weiss et al., 2018; Suzgun et al., 2019a;b). We show the results in Figure 3.

The LSTM achieves close to 0 error on the counting task when there is little to no extrapolation. However, the LSTM fails to learn the minimum and can extrapolates neither to larger numbers nor to longer sequences in both tasks. MLP, NALU and NAU keep some error in both tasks. These models extrapolate to some extent to larger numbers but do not extrapolate to larger sequences. On the other hand, the NSR completely solves both problems and also perfectly extrapolates in both dimensions.

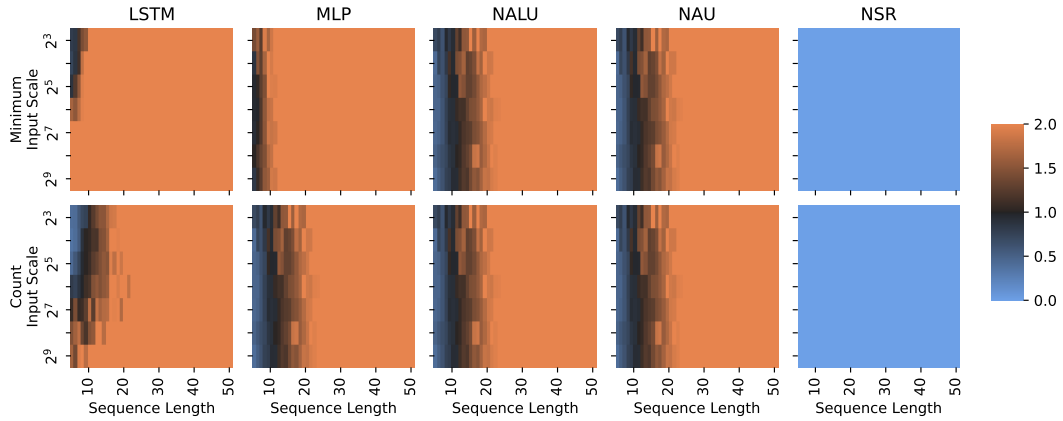


Figure 3. Mean absolute error of recurrent comparison architectures on finding the minimum element of a sequence (first row) and counting the occurrences of an element in the sequence (second row). We extrapolate training models to handle larger numbers than during training and larger sequences. The NSR outperforms all other architectures and is the only model solving and extrapolating both tasks.

4.5. Shortest Paths with Graph Neural Networks

In this experiment, we combine the NSR with the popular message passing structure (Gilmer et al., 2017; Battaglia et al., 2018) for graph neural networks (GNNs). Our problem is the algorithmic task of finding shortest paths from a source node (Velickovic et al., 2020; Tang et al., 2020; Xu et al., 2020). In contrast to existing work, the NSR will *learn* that the minimum is part of the solution, the model is not given this aggregation. We create a GNN variant—NSR-GNN— that uses an NSR in the aggregation step to aggregate the neighborhood messages in sequence form. Formally we can compare our NSR-GNN as follows against NEG (Velickovic et al., 2020) and IterGNN (Tang et al., 2020):

$$\text{NEG:} h^{l+1} = \texttt{UPDATE}(h^l, \min_{v \in N(v)} (\texttt{MESSAGE(v)}))$$

$$\text{IterGNN:} h^{l+1} = \texttt{UPDATE}(h^l, \min_{v \in N(v)} (\texttt{MESSAGE(v)}))$$

$$\text{NSR-GNN:} h^{l+1} = \texttt{NSR}(h^l, \|_{v \in N(v)} (\texttt{MESSAGE(v)}))$$

$N(v)$ contains the neighbors of v, `MESSAGE` and `UPDATE` are linear functions. Compared to NEG, IterGNN uses `MESSAGE` and `UPDATE` functions that must be homomorphic, NEG makes no restriction.

We train these models to learn to imitate the Bellman-Ford algorithm with an iteration-level supervision, following Velickovic et al. (2020). Nodes receive messages from all neighbors and their previous state (via a self loop edge). For training we create 10 graphs with 10 nodes each and edge weights uniformly sampled from the integers 1 to 10. Graphs are based on a random spanning tree, making all connected graphs equally likely. On average we add one random edge to every graph. We train for 1000 epochs.

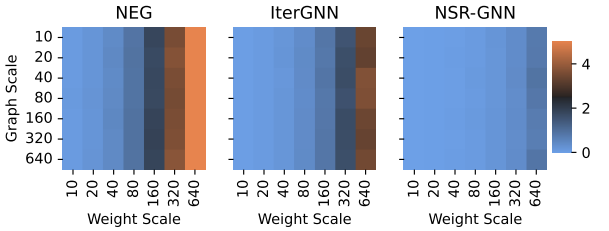
After training we expose the model to extrapolation in two dimensions. On the one hand we scale the maximum possible edge weight, on the other hand, we increase the graph size. Both extrapolations increase exponentially in powers of 2. In every extrapolation step, we compute the mean average error across 10 random test graphs. We repeat the entire setup for 10 different seeds. Figure 4a shows the results.

NEG struggles with extrapolating to larger weights while extrapolating well to larger graphs. IterGNN improves upon NEG: The homomorphic functions that allow IterGNN to better extrapolate across edge weights. The NSR outperforms both methods and achieves almost perfect extrapolation in both dimensions.

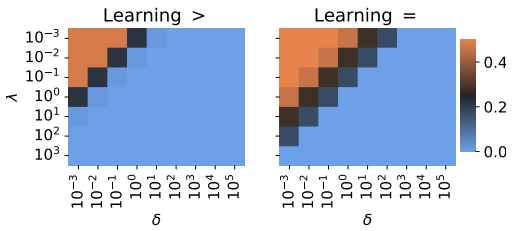
4.6. Ablation Study

Ablation on λ. We finish our analysis with a study on using the NSR for floats. We introduced the scaling constant λ to combat the vanishing gradients for W_M and W_S for very different inputs and to control the resolution limit of $\hat{Z}$. We repeat the experiment setup from Section 4.1. However, we scale the inputs so that their differences δ are different powers of 10 (with 10^0 being the integer setup as in the first experiment) and analyze the impact for different values of λ. Figure 4b shows the mean absolute error for the training set.

In both experiments, we can see that the NSR can even learn when the differences between numbers are large (errors in the bottom right are zero). This suggests vanishing gradients for W_M and W_S because of saturation in $\hat{S}$ and $\hat{Z}$ might not be as much of a practical concern.



(a) Learning and extrapolation shortest paths. Top left corner equals the training setting. Along the x-axis we increase the possible edge weights, along the y-axis we increase the graph size. All models can extrapolate to larger graphs. The homomorphic functions of IterGNN further allow it to extrapolate better to large weights than NEG. Yet, NSR outperforms both methods and is the only architecture extrapolating in both dimensions.



(b) Accuracy of the NSR for learning $>$ and $=$ when we vary the distance between different digits (δ, x-axis) and internal scaling parameter λ. If both δ and λ are small, we experience the first resolution error and the NSR fails to learn $>$. If both $\delta < \lambda^{-1}$, we experience the second resolution error and the NSR fails to learn $=$.

Figure 4.

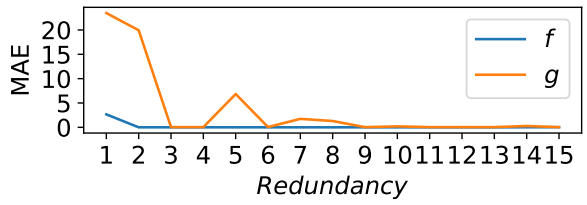


Figure 5. Mean absolute error on a testing set (with no extrapolation) for the piecewise-defined functions from Section 4.2. We experiment with different levels of redundancy from 1 to 15 as outlined in Section 3.2. Increased redundancy substantially reduces the error which reaches zero for already 3 redundant units. Afterwards there are diminishing benefits.

However, we can see errors in the top left corner in the left plot of Figure 4b. This is when differences between numbers become very small such that $\hat{S} \approx 0$. This causes a vanishing gradient on the important weight W_+ and prevents learning (the first resolution error).

In the right plot of Figure 4b, we can see the other resolution error where different numbers are close and the NSR mistakenly considers them equal. This error happens when $\lambda < \delta^{-1}$. Setting $\lambda = \delta^{-1}$ solves both resolution errors.

Therefore, we can control the NSR to support a resolution δ that we want the NSR to achieve by setting $\lambda = \delta^{-1}$. We already used λ in the image comparison task to adjust the NSR resolution to smaller differences while the CNN still needs training.

Ablation on redundancy. We experimented with different redundancy levels from Section 3.2 for learning f and g (the piecewise-defined functions from Experiment 4.2). We vary the redundancy in the NSR from 1 to 15 and repeat the experiment for 10 different seeds. Figure 5 shows the mean average error of the testing set without extrapolation for each redundancy level.

For both functions the error with using one NSR unit is high, suggesting that at least some seeds suffered from bad initialization. This effect is even more prominent in g which requires solving $=$ internally. We established in Section 3.2 that $=$ is already initialization dependent so functions building on top of this method surely are as well. However, the error rapidly declines with increasing redundancy and after 3 units, the error drops almost to zero (except one outlier). This confirms that redundancy is indeed an effective way to largely eliminate unlucky initial parameter values.

5. Conclusion

This paper introduced Neural Status Registers, a new architecture tailored for quantitative reasoning. Like other quantitative architectures such as the NAU or NALU, the NSR works directly on the input numbers and does not encode them like neuro-symbolic methods. The NSR fills the gap of learning comparisons between numbers that existing architectures struggle with.

We show that the NSR excels at learning comparisons. More importantly, we can plug the NSR into larger neural architectures as a neural module to solve comparisons. This allows us to solve interesting problems: For example we can combine the NSR with arithmetic models for piecewise-defined arithmetic, with CNNs for image digit comparisons, with recurrent architectures for numeric sequence problems, or with GNNs to compute shortest paths. The NSR clearly outperforms baseline comparison architectures and learns all these problems well. The NSR also extrapolate to more difficult problems, such as larger numbers, longer sequences, or larger graphs.

References

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Bland, R. *Learning XOR: exploring the space of a classic problem*. Department of Computing Science and Mathematics, University of Stirling, 1998.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Neural Information Processing Systems (NeurIPS), virtual*, 2020.

Cai, J., Shin, R., and Song, D. Making neural programming architectures generalize via recursion. In *International Conference on Learning Representations (ICLR), Toulon, France*, 2017.

Chen, K., Dong, Y., Qiu, X., and Chen, Z. Neural arithmetic expression calculator. *arXiv preprint arXiv:1809.08590*, 2018.

Dong, H., Mao, J., Lin, T., Wang, C., Li, L., and Zhou, D. Neural logic machines. In *International Conference on Learning Representations (ICLR), New Orleans, USA*, 2019.

Evans, R. and Grefenstette, E. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 2018.

Feser, J. K., Brockschmidt, M., Gaunt, A. L., and Tarlow, D. Neural functional programming. In *International Conference on Learning Representations (ICLR), Toulon, France*, 2017.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations (ICLR), New Orleans, USA*, 2019.

Freivalds, K. and Liepins, R. Improving the neural gpu architecture for algorithm learning. *arXiv preprint arXiv:1702.08727*, 2017.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *International Conference on Machine Learning (ICML), Sydney, Australia*, 2017.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS*. JMLR Workshop and Conference Proceedings, 2010.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

Grefenstette, E., Hermann, K. M., Suleyman, M., and Blunsom, P. Learning to transduce with unbounded memory. In *Neural Information Processing Systems (NeurIPS), Montreal, Canada*, 2015.

Heim, N., Pevnỳ, T., and Šmídl, V. Neural power units. *Neural Information Processing Systems (NeurIPS), remote*, 2020.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 1997.

Kaiser, L. and Sutskever, I. Neural gpus learn algorithms. In *International Conference on Learning Representations (ICLR), San Juan, Puerto Rico*, 2016.

Kim, S., Nam, H., Kim, J., and Jung, K. Neural sequence-to-grid module for learning symbolic rules. In *AAAI Conference on Artificial Intelligence (AAAI), remote*, 2021.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR), San Diego, USA*, 2015.

Lake, B. and Baroni, M. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International Conference on Machine Learning (ICML), Stockholm, Sweden*, 2018.

Lample, G. and Charton, F. Deep learning for symbolic mathematics. In *International Conference on Learning Representations (ICLR), Addis Ababa, Ethiopia*, 2020.

Le, H., Tran, T., and Venkatesh, S. Neural stored-program memory. In *International Conference on Learning Representations (ICLR), Addis Ababa, Ethiopia*, 2020.

Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., et al. Solving quantitative reasoning problems with language models. *Neural Information Processing Systems (NeurIPS), New Orleans, USA*, 2022.

Li, C., Tarlow, D., Gaunt, A. L., Brockschmidt, M., and Kushman, N. Neural program lattices. In *International Conference on Learning Representations (ICLR), Toulon, France*, 2017.

Madsen, A. and Johansen, A. R. Neural arithmetic units. In *International Conference on Learning Representations (ICLR), Addis Ababa, Ethiopia*, 2020.

Martius, G. S. and Lampert, C. Extrapolation and learning equations. In *International Conference on Learning Representations (ICLR), Workshops, Toulon, France*, 2017.

Mistry, B., Farrahi, K., and Hare, J. A primer for neural arithmetic logic modules. In *Journal of Machine Learning Research*, 2022.

Reed, S. E. and de Freitas, N. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR), San Juan, Puerto Rico*, 2016.

Sahoo, S., Lampert, C., and Martius, G. Learning equations for extrapolation and control. In *International Conference on Machine Learning (ICML), Stockholm, Sweden*, 2018.

Santoro, A., Hill, F., Barrett, D., Morcos, A., and Lillicrap, T. Measuring abstract reasoning in neural networks. In *International Conference on Machine Learning (ICML), Stockholm, Sweden*, 2018.

Saxton, D., Grefenstette, E., Hill, F., and Kohli, P. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations (ICLR), New Orleans, USA*, 2019.

Schlör, D., Ring, M., and Hotho, A. inalu: Improved neural arithmetic logic unit. *arXiv preprint arXiv:2003.07629*, 2020.

Suzgun, M., Belinkov, Y., and Shieber, S. M. On evaluating the generalization of lstm models in formal languages. *Society for Computation in Linguistics (SCiL), New York City, USA*, 2019a.

Suzgun, M., Gehrmann, S., Belinkov12, Y., and Shieber, S. M. Lstm networks can perform dynamic counting. In *Association for Computational Linguistics (ACL), Florence, Italy*, 2019b.

Tang, H., Huang, Z., Gu, J., Lu, B.-L., and Su, H. Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. *Neural Information Processing Systems (NeurIPS), remote*, 2020.

Trask, A., Hill, F., Reed, S. E., Rae, J., Dyer, C., and Blunsom, P. Neural arithmetic logic units. In *Neural Information Processing Systems (NeurIPS), Montreal, Canada*, 2018.

Velickovic, P., Ying, R., Padovano, M., Hadsell, R., and Blundell, C. Neural execution of graph algorithms. In *International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia*, 2020.

Weiss, G., Goldberg, Y., and Yahav, E. On the practical computational power of finite precision rnns for language recognition. In *Association for Computational Linguistics (ACL), Florence, Italy*, 2018.

Weston, J., Chopra, S., and Bordes, A. Memory networks. In *International Conference on Learning Representations (ICLR), San Diego, USA*, 2015.

Xian, Y., Schiele, B., and Akata, Z. Zero-shot learning-the good, the bad and the ugly. In *IEEE conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, USA*, 2017.

Xu, K., Li, J., Zhang, M., Du, S. S., Kawarabayashi, K., and Jegelka, S. What can neural networks reason about? In *International Conference on Learning Representations(ICLR), Addis Ababa, Ethiopia*, 2020.

Xu, K., Zhang, M., Li, J., Du, S. S., Kawarabayashi, K., and Jegelka, S. How neural networks extrapolate: From feedforward to graph neural networks. In *International Conference on Learning Representations (ICLR), remote*, 2021.

Zaremba, W. and Sutskever, I. Reinforcement learning neural turing machines-revised. *arXiv preprint arXiv:1505.00521*, 2016.

Zaremba, W., Mikolov, T., Joulin, A., and Fergus, R. Learning simple algorithms from examples. In *International Conference on Machine Learning (ICML), New York City, USA*, 2016.

A. Motivating Example for a Redundant NSR

We conducted a small experiment to test the NSR's sensitivity to weight initialization. The task was to learn equality between numbers. The first set of weights W_M and W_S were glorot-uniform initialized. The possible initialization values of W_+ and W_0 varied from -0.25 to 0.25 in 0.05 increments. We repeated every setup 10 times and measured how often the NSR successfully learned equality. Figure 6 shows the results. In all settings, the NSR learnt equality in at least half the runs. On the other hand, low values for W_0 resulted in lower chances at learning equality. A redundant NSR would have multiple chances to roll a high value for W_0 for one of the initializations.

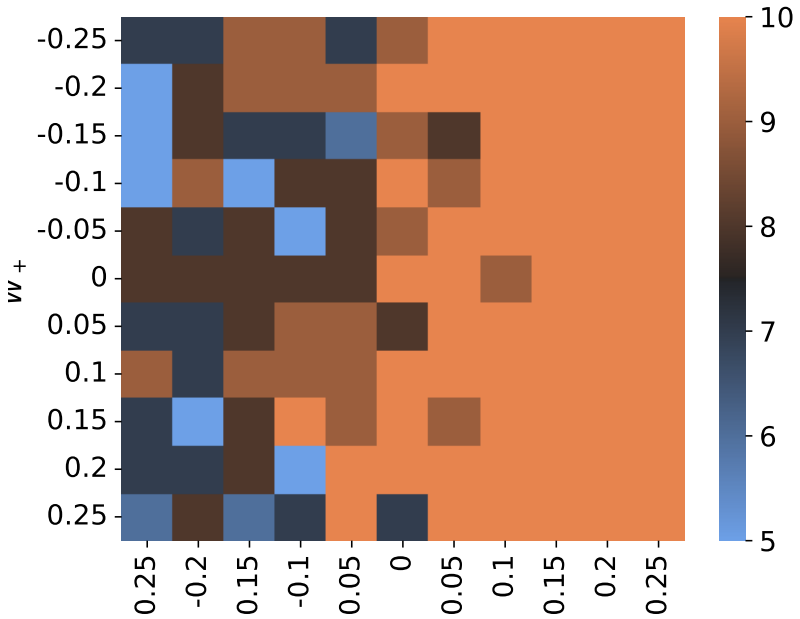


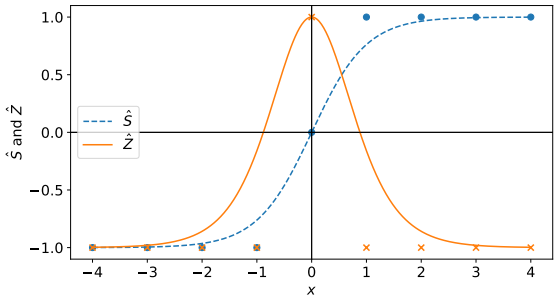
Figure 6. Number of succesful training trials (out of 10) for learning equality for given W_0 and W_+ while initializing W_M and W_S glorot uniform and $b = 0$. Not all parameter initializations for W_0 and W_+ learn equally well. This motivates to use redundant structure in the NSR to have multiple random initializations.

B. Continuous Approximations for sign and zero Bits

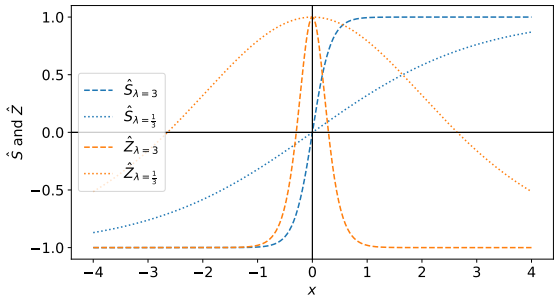
Recall that we use the following functions $\hat{S}$ and $\hat{Z}$ to approximate the sign bits:

$$\hat{S}(d) = \tanh\left(\lambda(m-s)\right)$$
$$\hat{Z}(d) = 1 - 2 * \tanh\left(\lambda(m-s)\right)^2$$

Figure 7a shows the alignment of $\hat{S}$ and $\hat{Z}$ with their respective bits when not using the rescaling factor ($\lambda = 1$). The right Figure 7b shows the bit approximations for $\lambda = 3$ and $\lambda = \frac{1}{3}$. Setting $\lambda > 1$ sharpens the bit approximation and the NSR can resolve smaller differences. On the other hand, we can soften the approximations by setting $\lambda < 1$ to keep unsaturated functions and large gradients for larger differences $m - s$.



(a) Discrete sign and zero bits for integers and the continuous approximations $\hat{S}$ and $\hat{Z}$. Their scaling parameter λ is set to 1.



(b) Two rescaled versions of $\hat{S}$ and $\hat{Z}$ with $\lambda = 3$ and $\lambda = \frac{1}{3}$.

Figure 7.

C. Full Experiment Results for Learning Comparisons

Table 5 shows the results for all comparisons $>, \geq, =, \neq, \leq, <$. For $>, \geq, \leq$, and $<$ from Section 4.1.

Table 5. Learning comparisons between numbers. For each column in the table, the column header is compared with every integer number with difference at most 5. Table entries denote the mean absolute error for model predictions (lower numbers are better).

Comparison	Model	10^1	10^2	10^3	10^4	10^5	10^6	10^7
$>$	MLP	0.00 ±0.00	0.00 ±0.00	0.01 ±0.01	0.23 ±0.16	0.49 ±0.12	0.52 ±0.01	0.52 ±0.01
	npu	0.75 ±0.19	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02
	nalu	0.00 ±0.00	0.00 ±0.00	0.11 ±0.06	0.50 ±0.09	0.52 ±0.00	0.52 ±0.00	0.52 ±0.00
	nau	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
$<$	MLP	0.00 ±0.00	0.00 ±0.01	0.09 ±0.14	0.33 ±0.18	0.52 ±0.01	0.52 ±0.01	0.52 ±0.01
	npu	0.83 ±0.14	0.52 ±0.02	0.52 ±0.01	0.52 ±0.01	0.52 ±0.01	0.52 ±0.01	0.52 ±0.01
	nalu	0.00 ±0.00	0.00 ±0.00	0.11 ±0.05	0.51 ±0.03	0.52 ±0.00	0.52 ±0.00	0.52 ±0.00
	nau	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
$=$	MLP	0.18 ±0.16	0.28 ±0.17	0.47 ±0.14	0.52 ±0.05	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	npu	0.40 ±0.15	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	nalu	0.45 ±0.10	0.45 ±0.10	0.48 ±0.08	0.52 ±0.07	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	nau	0.33 ±0.11	0.33 ±0.11	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
$\neq$	MLP	0.18 ±0.11	0.27 ±0.12	0.42 ±0.11	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	npu	0.43 ±0.15	0.52 ±0.05	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	nalu	0.38 ±0.12	0.38 ±0.12	0.41 ±0.13	0.50 ±0.10	0.52 ±0.07	0.50 ±0.00	0.50 ±0.00
	nau	0.33 ±0.11	0.33 ±0.11	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
$\geq$	MLP	0.00 ±0.00	0.00 ±0.00	0.01 ±0.01	0.23 ±0.16	0.49 ±0.12	0.52 ±0.01	0.52 ±0.01
	npu	0.75 ±0.19	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02
	nalu	0.00 ±0.00	0.00 ±0.00	0.11 ±0.06	0.50 ±0.09	0.52 ±0.00	0.52 ±0.00	0.52 ±0.00
	nau	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
$\leq$	MLP	0.00 ±0.00	0.00 ±0.00	0.02 ±0.04	0.25 ±0.19	0.52 ±0.01	0.52 ±0.00	0.52 ±0.00
	npu	0.74 ±0.20	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02
	nalu	0.00 ±0.00	0.00 ±0.00	0.11 ±0.06	0.50 ±0.08	0.52 ±0.00	0.52 ±0.00	0.52 ±0.00
	nau	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00

D. Complete Results for Piecewise Functions

Table 6 shows results for all comparison architectures trying both NALU and NAU as the arithmetic model. Across both learning tasks and all comparison architectures, NALU performs better as NAU as arithmetic unit. There are two exceptions for learning g where NAU as arithmetic unit is better. However, the error in those two cases is so large we would argue learning failed completely.

We hypothesize that NALU works better since it does not clip gradients—which NAU does. In case of gradient clipping, NAU propagates 0 gradients to the upstream comparison unit which makes learning that unit much more difficult. On the other hand, NALU always provides gradients to the comparison unit.

Table 6. Learning piecewise-defined functions. The condition is between the first two parameters. One parameter is the column header the other parameter can be integers with difference at most 5. The remaining numbers are sampled from $[-100, 100]$. Table entries denote the mean absolute error for model predictions.

Function	Comparison	2^1	2^2	2^3	2^4	2^5	2^6	2^7
f	MLP+NALU	0.00 ±0.00	0.00 ±0.00	0.01 ±0.00	0.01 ±0.00	0.96 ±2.72	3.92 ±4.69	3.95 ±5.57
	MLP+NAU	1.46 ±1.67	4.45 ±8.09	6.42 ±8.88	17.72 ±24.86	27.41 ±37.65	49.67 ±56.88	103.28 ±152.32
	NALU+NALU	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.01 ±0.00	0.02 ±0.03	0.30 ±0.81	2.77 ±4.77
	NALU+NAU	7.28 ±12.25	14.83 ±21.82	9.72 ±12.24	14.12 ±14.42	21.86 ±23.19	25.78 ±27.84	65.97 ±114.87
	NAU+NALU	0.00 ±0.00	0.01 ±0.00	0.01 ±0.00	0.01 ±0.00	0.02 ±0.01	0.03 ±0.01	0.06 ±0.03
	NAU+NAU	7.19 ±12.28	13.45 ±21.69	9.76 ±12.22	16.15 ±18.70	21.41 ±22.59	37.93 ±53.33	93.63 ±142.86
	NSR+NALU	0.00 ±0.00	0.00 ±0.00	0.01 ±0.00	0.01 ±0.00	0.01 ±0.01	0.02 ±0.01	0.05 ±0.06
	NSR+NAU	1.75 ±1.83	2.27 ±2.54	4.01 ±5.34	5.84 ±6.28	10.28 ±15.71	15.58 ±23.08	24.01 ±42.46
g	MLP+NALU	4.78 ±8.78	5.81 ±9.18	4.71 ±8.52	5.42 ±8.73	4.26 ±6.55	12.72 ±13.34	25.93 ±16.09
	MLP+NAU	8.28 ±9.89	8.64 ±10.52	9.30 ±9.71	15.96 ±13.28	18.37 ±13.11	33.30 ±9.29	37.38 ±10.04
	NALU+NALU	25.39 ±9.36	30.76 ±14.19	37.31 ±29.58	43.95 ±47.93	75.85 ±119.67	139.51 ±254.32	204.69 ±357.51
	NALU+NAU	34.51 ±10.76	36.23 ±13.06	36.75 ±14.96	34.85 ±15.66	33.97 ±14.21	32.75 ±11.56	40.14 ±18.70
	NAU+NALU	24.89 ±9.20	28.86 ±12.74	29.62 ±14.80	32.59 ±21.83	57.92 ±71.28	82.04 ±120.31	148.11 ±256.99
	NAU+NAU	33.86 ±11.40	37.59 ±11.18	35.92 ±13.19	37.52 ±13.16	40.80 ±16.17	42.15 ±24.79	57.77 ±37.23
	NSR+NALU	0.04 ±0.03	0.04 ±0.04	0.12 ±0.19	0.08 ±0.11	0.06 ±0.05	0.08 ±0.08	0.21 ±0.30
	NSR+NAU	5.66 ±8.20	6.48 ±8.34	7.24 ±6.48	12.72 ±11.35	17.48 ±12.49	23.94 ±24.45	41.20 ±50.47