Machine Learning Minor Project

Counting/Manipulating numbers and, to Extrapolate the Numeric Relationship using Neural Arithmetic Logic Units

---

IIT INDORE

Submitted to :
Dr.Aruna Tiwari
Associate Professor
Department of
Computer Science
IIT Indore

Submitted By:
Bharath kollanur
(MS2104101004)
MS Research Student
Department of
Computer Science
IIT Indore

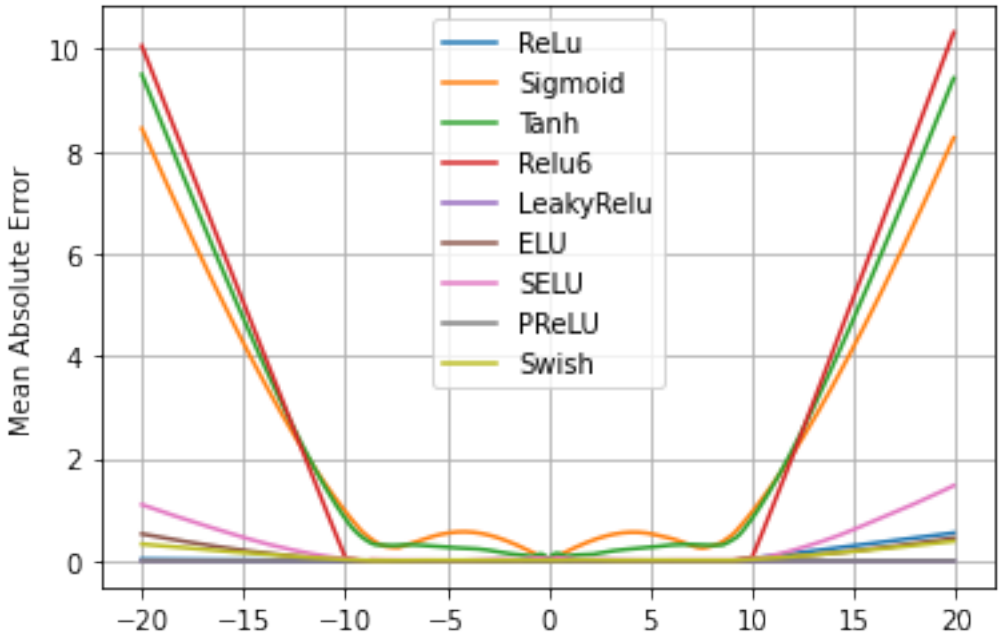
Contents

1 INTRODUCTION

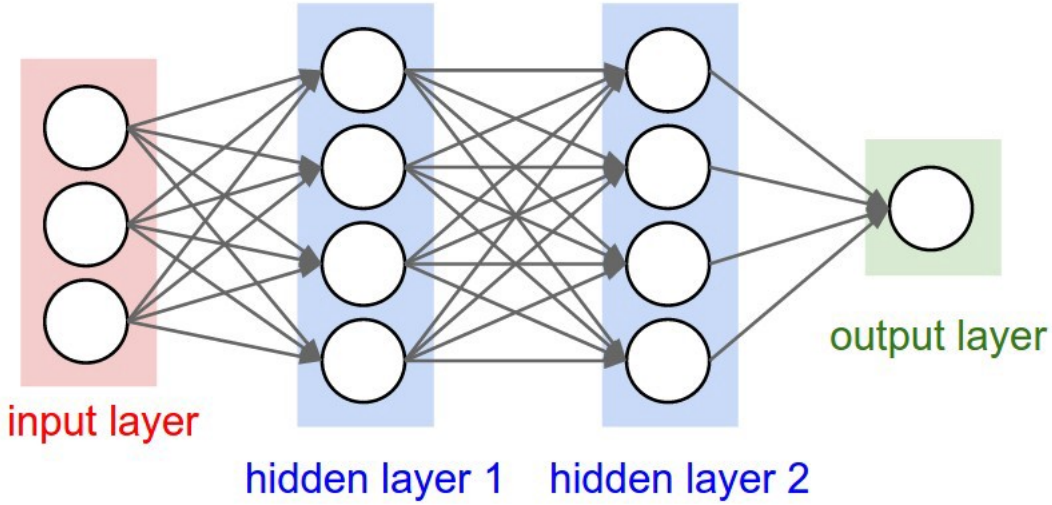
The ability to represent and manipulate numerical quantities is apparent in the behavior of many species, from insects to mammals to humans, suggesting that basic quantitative reasoning is a general component of intelligence[1,2].
while neural networks can successfully represent and manipulate numerical quantities given an appropriate learning signal, the behavior that they learn does not generally exhibit systematic generalisation [3,4]. Specifically, one frequently observes failures when quantities that lie outside the numerical range used during training are encountered at test time, even when the target function is simple. This failure pattern indicates that the learned behavior is better characterized by memorization than by systematic abstraction.

2 NUMERICAL EXTRAPOLATION FAILURES IN NEURAL NETWORKS

To illustrate the failure of systematicity in standard networks, the behavior of various MLPs trained to learn the scalar identity function, which is the most straight forward systematic relationship possible. The notion that neural networks struggle to learn identity relations is not new [5]. eventhough many of the architectures evaluated below could theoretically represent the identity fucntion, they typically fail to acquire it.



The Neural Network architecture used for the training purpose is :



3 layers (2 - hidden layers, 1 - output layer)
Each Hidden layer contains 8 neurons
L2 Loss function and Stochastic Gradient Descent Optimiser is used.
The following are the activation functions are used to demonstrate the failure of the neural networks.
(Relu, Sigmoid, Tanh, Relu6, LeakyRelu, ELU, SELU, PRelu, Swish).
Training Data range (-10, 10).
Testing Data range (-10, 20).

To illustrate the failure of systematicity in standard networks, the behaviour of various MLP's trained to learn the scalar identity function.
The notion of that neural networks struggle to learn the identity function is not new[5]. Even though many of the architectures evaluated below could theoretically represent the identity function, they typically fail to acquire it.
We see that even over a basic task using a simple architecture, all nonlinear functions fail to learn to represent numbers outside of the range seen during training. The severity of this failure directly corresponds to the degree of non-linearity within the chosen activation function. Some activations learn to be highly linear (such as PReLU) which reduces error somewhat, but sharply non-linear functions such as sigmoid and tanh fail consistently. Thus, despite the fact that neural networks are capable of representing functions that extrapolate, in practice we find that they fail to learn to do so.

- Loss function used in the following project is smooth L1 Loss.

$$l_n = \begin{cases} 0.5(x_n - y_n)^2/beta, & if|x_n - y_n| < beta. \\ |x_n - yn| - 0.5 * beta, & otherwise \end{cases} \quad (1)$$

$Optimiserusedinthefollowingprojectis RMSprop.$

$\mathrm{E}[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)(\frac{\delta C}{\delta w})^2$

$w_t = w_{t-1} - \frac{\eta}{\sqrt{E[g^2]_t}} \frac{\delta C}{\delta w}$

$\frac{\delta C}{\delta w}$ gradient of the cost function with respect to the weight.

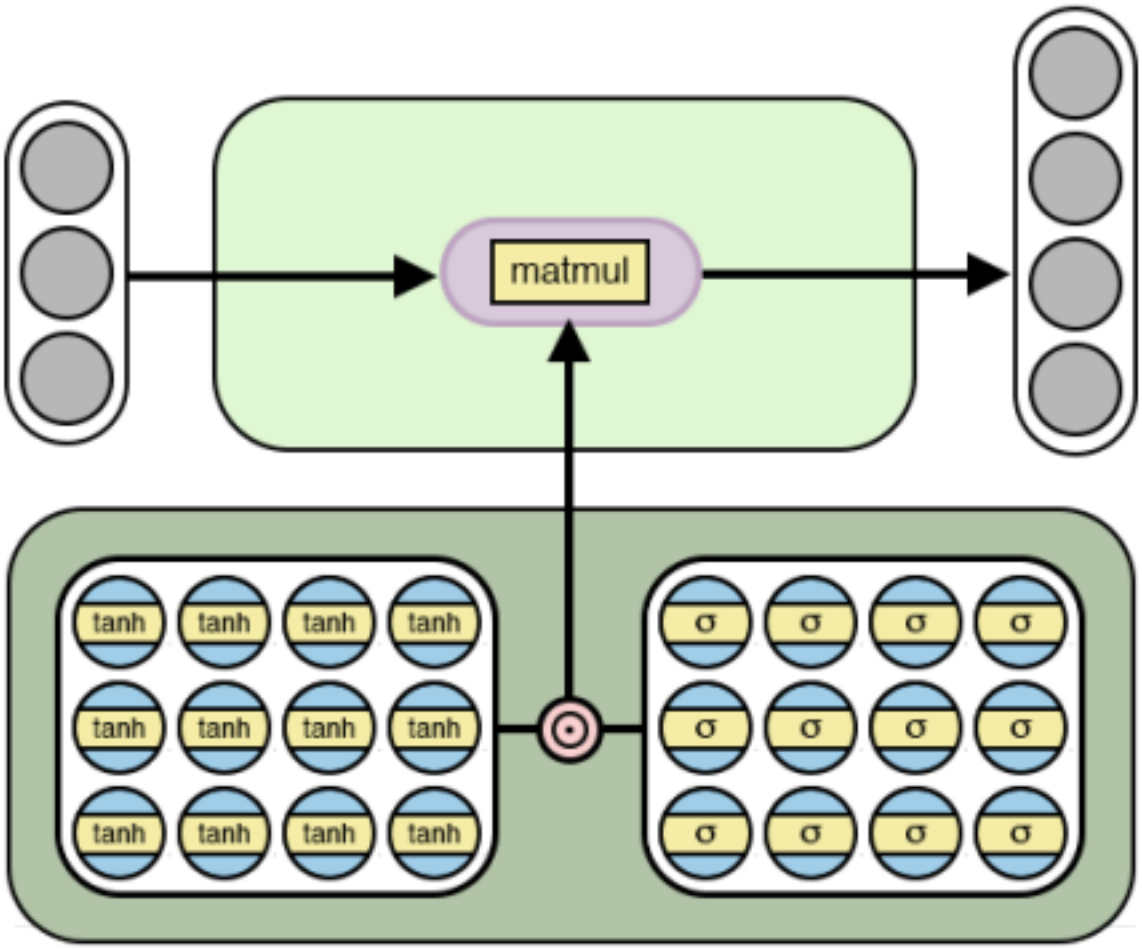
$E[g^2]_t$ moving average of squared gradients.

η learning rate, β moving average parameter (default value 0.9)

3 Neural Accumulator (NAC)

The neural accumulator (NAC), which is a special case of a linear (affine) layer whose transformation matrix W consists just of 1's, 0's, and 1's; that is, its outputs are additions or subtractions (rather than arbitrary rescalings) of rows in the input vector. This prevents the layer from changing the scale of the representations of the numbers when mapping the input to the output, meaning that they are consistent throughout the model, no matter how many operations are chained together. We improve the inductive bias of a simple linear layer by encouraging 0's, 1's, and 1's within W in the following way.

Since a hard constraint enforcing that every element of W be one of 1, 0, 1 would make learning hard, we propose a continuous and differentiable parameterization of W in terms of unconstrained parameters: $W = \tanh(\hat{W}) * \sigma(\hat{M})$. This form is convenient for learning with gradient descent and produces matrices whose elements are guaranteed to be in [1, 1] and biased to be close to 1, 0, or 1.1 The model contains no bias vector, and no squashing nonlinearity is applied to the output.



3.1 Addition using NAC

Training data range(20, 40)
Validating data range (20, 40)
Testing data range (0, 60)

Test Accuracy for [+] with 1 operands : 100.00%
Test Accuracy for [+] with 2 operands : 100.00%
Test Accuracy for [+] with 3 operands : 100.00%
Test Accuracy for [+] with 4 operands : 100.00%
Test Accuracy for [+] with 5 operands : 100.00%
Test Accuracy for [+] with 6 operands : 100.00%
Test Accuracy for [+] with 7 operands : 100.00%
Test Accuracy for [+] with 8 operands : 100.00%
Test Accuracy for [+] with 9 operands : 100.00%

Test Accuracy for [+] with 10 operands : 100.00%

3.2 Subtraction using NAC

Training data range(20, 40)
Validating data range (20, 40)
Testing data range (0, 60)

Test Accuracy for [-] with 1 operands : 100.00%
Test Accuracy for [-] with 2 operands : 100.00%
Test Accuracy for [-] with 3 operands : 100.00%
Test Accuracy for [-] with 4 operands : 100.00%
Test Accuracy for [-] with 5 operands : 100.00%
Test Accuracy for [-] with 6 operands : 100.00%
Test Accuracy for [-] with 7 operands : 100.00%
Test Accuracy for [-] with 8 operands : 100.00%
Test Accuracy for [-] with 9 operands : 100.00%
Test Accuracy for [-] with 10 operands : 100.00%

3.3 Random Function chosen and learning using NAC

Y = x1 + x2 - x3 + X4 - x5 + x6 - x7 - x8

$W^{|}$ = [8.6499, 8.4963, -8.4905, 8.6622, -8.4617, 8.5317, -8.4859, -8.5097]

$M^{|}$ = [14.0710, 14.0996, 14.1673, 14.0797, 14.1597, 14.0898, 14.1709, 14.1493]

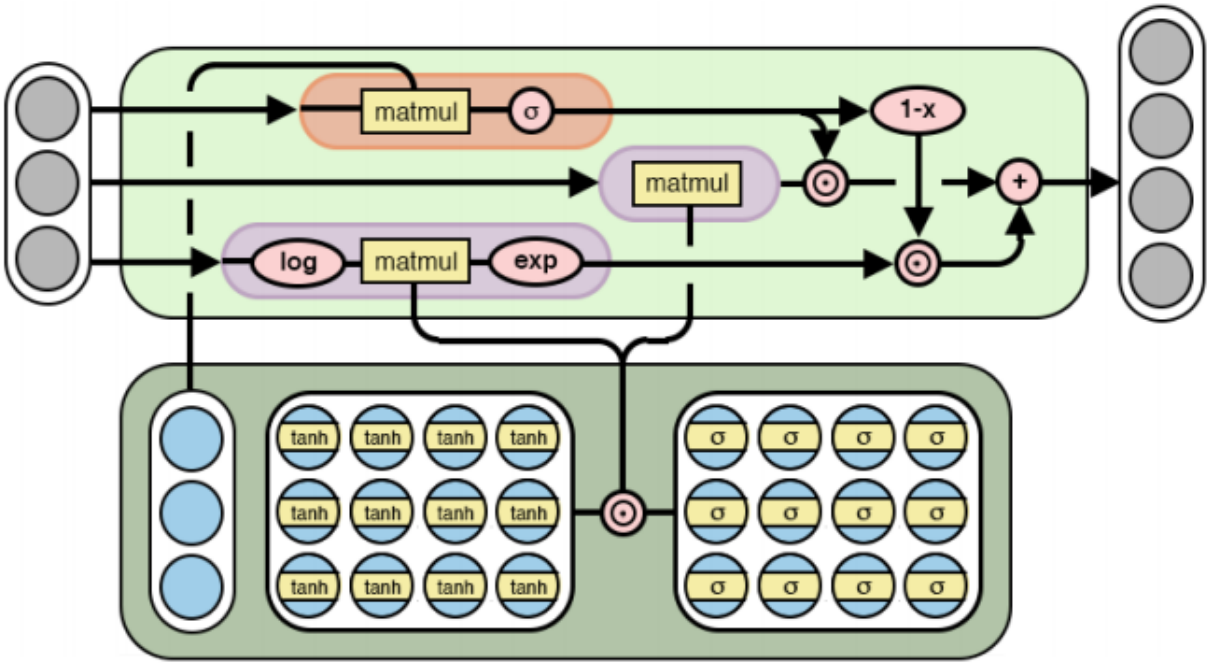
$W = \tanh(W^{|}).\sigma(M^{|})$

[1.0000, 1.0000, -1.0000, 1.0000, -1.0000, 1.0000, -1.0000, -1.0000]

4 Neural Arithmetic Logic Unit (NALU)

While addition and subtraction enable many useful systematic generalizations, a similarly robust ability to learn more complex mathematical functions, such as multiplication, may be be desirable. Figure 2 describes such a cell, the neural arithmetic logic unit (NALU), which learns a weighted sum between two subcells, one capable of addition and subtraction and the other capable of multiplication, division, and power functions such as $\sqrt{x}$. importantly, the NALU demonstates how the NAC can be extended with gate-controlled sub-operations, facilitating end-to-end learning of new classes of numerical functions. As with the NAC, there is the same bias against learning to rescale during the mapping from input to output.

The NALU consists of two NAC cells (the purple cells) interpolated by a learned sigmoidal gate g (the orange cell), such that if the add/subtract subcell's output value is applied with a weight of 1 (on), the multiply/divide subcell's is 0 (off) and vice versa. The first NAC (the smaller purple subcell) computes the accumulation vector a, which stores results of the NALU's addition/subtraction operations; it is computed identically to the original NAC, (i.e., a = Wx). The second NAC (the larger purple subcell) operates in log space and is therefore capable of learning to multiply and divide, storing its results in m:



NAC: a = Wx W = tanh(Wˆ) (Mˆ)

NALU: y=ga+(1g)m m=expW(log(—x—+)), g=(Gx)

where prevents log 0. Altogether, this cell can learn arithmetic functions consisting of multiplication, addition, subtraction, division, and power functions in a way that extrapolates to numbers outside of the range observed during training.

4.1 Arithmetic operations using NALU (2 operands)

Accuracy with two different operands on different operators.

Test Accuracy for [+] with 2 operands : 99.80%
Test Accuracy for [-] with 2 operands : 87.30%
Test Accuracy for [*] with 2 operands : 92.30%
Test Accuracy for [/] with 2 operands : 98.60%
Test Accuracy for [$\sqrt{}$] with 2 operands : 99.10%
Test Accuracy for [power of 2] with 2 operands : 91.60%

4.2 Arithmetic operations using NALU (5 operands)

Accuracy with two different operands on different operators.

Test Accuracy for [+] with 5 operands : 100.00%
Test Accuracy for [-] with 5 operands : 98.90%
Test Accuracy for [*] with 5 operands : 93.90%
Test Accuracy for [/] with 5 operands : 18.70%

5 Improvements

The following improvement is made to the original algorithm.

5.1 Reinitialization

Since NALU doesn't recover well from local optima by its own [6], we suggest a reinitialization strategy. This strategy evaluates the loss for each m-th epoch and randomly reinitializes all weights if the loss did not improve for the last n steps and if the loss is greater than a predefined threshold.

6 Challenges

6.1 Exploding Intermediate Results

From the observation, the training often fails because of exploding intermediate results especially when stacking NALUs to deeper networks and having many input and output variables. For example, consider a model con- sisting of four NALU layers with four inputs, four outputs neurons each and a simple summation task. Assuming the same magnitude for all input dimensions the first layer could (depending on the initialization) calculate x4 for each output dimension whereas the following layer could calculate (x4)4 ultimately leading to x4l for layer l. Therefore, the calculation can exceed the valid numeric range already in the forward pass ultimately causing the training to fail. For example in a network with three NALU layers in a MNIST classification downstream task, the NALU models failed after the first training steps (resulting in NaNs).

6.2 Multiplication / Division with Negative Result

The NALU by design isn't capable of multiplying or dividing values with a negative result. In the multiplicative path, the input values are represented by their absolute value to guarantee a real-valued calculation in log-space. There- fore, learning multiplication for mixed signed data with a result y ¡ 0 fails. Since the NALU is expected to learn either multiplication/division or sum- mation/subtraction in each layer, sign(y) is in the multiplicative case clearly determined by the number of negative multiplicands being even or odd. Since input dimensions can be deactivated for Wi,j = 0, the sign can't be inferred counting negative input variables. In the next section, we propose a method taking deactivated input dimensions into account to correct the sign of the mul- tiplicative path.

6.3 Initialization Sensitivity

From the observation that the NALU architecture is very prone to non-optimal initializations, which can lead to vanishing gradients or optimization into undesired local optima. Finding the optimal initialization in general is difficult since it depends on the task and the input distribution, which in a real world scenario is both unknown.

7 Future Work

7.1 Independent Weights

The summative and the multiplicative paths share their weights Wˆ and Mˆ in the NALU model. Better to use separate weights for each path for two reasons: First, the model can optimize W for the multiplicative and summative path without interfering the other path. For example, in a setting with inputs a, b ¡ 1 with the operation a×b, the result would be a positive number greater than 1 and the optimal parameter setting would be Wa = Wb = 1 and g = 0. However, the only way for the summative path (see Equation 3) to generate positive results is to force the weights Wa and Wb towards 1. In this case, the summative and multiplication path force the weights into opposite directions. With separate weights, the model can learn optimal weights for both paths and select the correct path using the gate. Second, consider the multiplicative path yields huge results whereas the summative path represents the correct solution but yields relatively small results. In that case, the multiplicative path influences the results even if the sigmoid gate is almost closed. For example in a setting with inputs a, b, c ¿ 0 with the desired result a + b, the summative path yields the correct solution and the optimal weight setting is Wa = Wb = 1, Wc = 0 and g = 1. In that case, W may contain very small weights to omit the input c. However, small negative weights for Wc (e.g. 1e 5) will lead to the situation, that the multiplication path divides the inputs a and b by values near to 0 which results in large numbers. Consequently, the multiplicative path influences the results even if the gate (see Equation 5) is almost closed. In this case, the model with independent weights can optimize Wm to smaller values to mitigate influence caused by the leaky gate.

8 References

[1] Stanislas Dehaene. The Number Sense: How the Mind Creates Mathematics. Oxford University Press, 2011.

[2] C. Randy Gallistel. Finding numbers in the brain. Philosophical Transactions of the Royal Society B, 373, 2017.

[3] Jerry A. Fodor and Zenon W. Pylyshyn. Connectionism and cognitive architecture: a critical analysis. Cognition, 28(1–2):3–71, 1988.

[4] Gary F. Marcus. The Algebraic Mind: Integrating Connectionism and Cognitive Science. MIT Press, 2003.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. CoRR, abs/1603.05027, 2016. URL http://arxiv.org/abs/1603.05027.

[6] A. Madsen and A. Rosenberg Johansen. Measuring Arithmetic Extrapola- tion Performance. 2019.

[7] Neural Arithmetic Logic Unit https://arxiv.org/pdf/1808.00508.pdf

[8] Improved Neural Arithmetic Logic Unit https://arxiv.org/pdf/2003.07629.pdf