Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Neural Arithmetic

Shuhailo Oleksii

2021

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Shuhailo Oleksii**	Personal ID number:	**478014**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Electrical Engineering and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Neural Arithmetic

Bachelor’s thesis title in Czech:

Neural Arithmetic

Guidelines:

- Review and embed existing research around Neural Arithmetic and their regularization methods
- Train and compare different arithmetic units on tasks to perform arithmetic on MNIST digits
- Analyse whether the the tasks of classification and arithmetic are seperated into the respective parts of the neural networks

Bibliography / sources:

1. Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015
2. Neural Power Units, Heim et al., NeurIPS 2020
3. Neural Arithmetic Units, Madsen & Johansen, ICML 20204. Neural Arithmetic Logic Unit, Trask et al., NeurIPS 2018
5. https://julialang.org/

Name and workplace of bachelor’s thesis supervisor:

MSc. Niklas Maximilian Heim, Department of Computer Science, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.02.2021** Deadline for bachelor thesis submission: **13.08.2021**

Assignment valid until: **30.09.2022**

MSc. Niklas Maximilian Heim Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student’s signature

Declaration

This bachelor's thesis is a product of my mind. Wherever there is a mansion of someone's work, all efforts are made to make it clear. Bachelor's thesis was done under supervision by Niklas Heim, guarantor ING. Tomáš Pevný.

Signature

Date

Abstract

Neural networks can learn complex functions, but they often have troubles with extrapolating even simple arithmetic operations on real numbers beyond the training range. A sub field of Neural Networks called Neural Arithmetic tries to address this extrapolation problem by making use of arithmetic operations like addition, multiplication, or division.

This thesis provides a comparison between different arithmetic layers on their extrapolation performance for simple functions and on a recurrent task.

Additionally, we exploit how arithmetic models can be used to build more transparent models by trying a simple equation discovery.

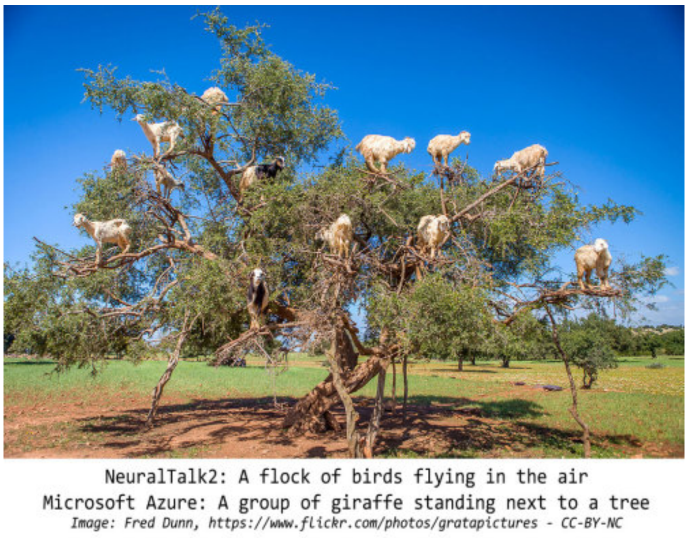
The general introduction is done in Section 1. Section 2 describes Neural Networks and Neural Arithmetic layers. Section 3 contains the function learning, recurrent, and equation discovery experiments and Section 4 the conclusion.

Contents

Chapter 1

Introduction

One of the most complex mechanisms that exist in nature is the human brain. Human brains have biological networks, which scientists in many fields are trying to replicate. One of the products such a trial have called Neural Networks (NN). This technology tries to mimic human brain abilities to some extent. Generally speaking, developing and maintaining NN's mechanisms verified that they are great approximators, meaning that they are capable of approximating no matter what function with arbitrary precision, this feature is also known as universal approximation theorem. There is no need to code them directly, they can learn from the environment itself, and they work great with higher dimensions. They are used in many fields of science. Some examples of usage are Natural Language Processing (NLP) [15], car driving automation [1], business risk minimization [14], image processing.



NeuralTalk2: A flock of birds flying in the air
Microsoft Azure: A group of giraffe standing next to a tree
Image: Fred Dunn, https://www.flickr.com/photos/gratapictures - CC-BY-NC

Figure 1.1: Neural Networks being deceived by the goats on the tree [13].

Presented image, Fig. 1.1, here to show some peculiar result performed by mentioned neural networks technology. One of the cases where NN's are widely used called object recognition. The objective of object recognition is to find or identify certain things on the image. Human brain can correctly manage this task without any problems.

After being trained with huge amount of different images network still can not correctly identify goat in an unexpected environment. When the goats were mostly presented on images with grass, mountains, etc., any image that have goat that are not within this environment will not be classified correctly. This particular type of problem that most of the networks are suffered from called extrapolation.

It is observed that even though Neural Network models are capable of approximating functions of many types with exceeding accuracy, given an appropriate amount of training data, they still tend to memorize more than see the general pattern. As complicated as it seems at first, the picture below shows the problem of extrapolation with a simple Neural Network in a much easier way. Looking at Fig. 1.2, it can be said that the network does a pretty good job inside the training range boundaries. Thus, it is interpolating well, can perform correctly within the provided data. But after receiving data outside the training range model does not generalize and thus can no longer correctly estimate points of a simple parabola function, it is hitting the extrapolation problem.

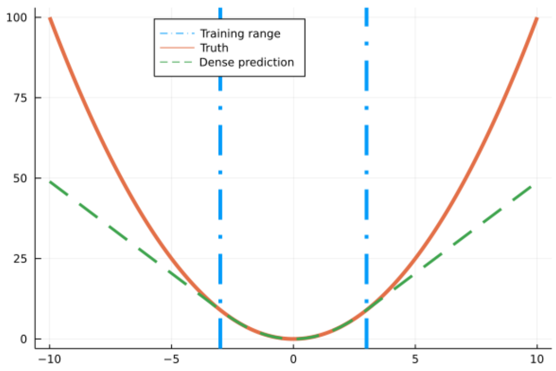


Figure 1.2: Simple Neural Network trying to approximate function $f(x) = x^2$, where blue line is a graph of x^2 and orange line is a graph approximated by the network after being trained in the range of a training data, which marked by a greed dash.

The young field of Neural Networks, Neural Arithmetic, was developed to make modules capable of doing arithmetic operations that are being relatively easy to interpret and can possibly solve extrapolation issues usual networks are facing. Analysing already proposed solutions to the Neural Arithmetic is a main focus of this thesis and will be discussed later.

The other interesting application of Neural Arithmetic networks is ease of interpretability. In general, interpretability characteristic is ability of something to be easily described or explained. Having a glimpse inside small linear NN models, it is clear how they operate and how much each part in the network contributes to the final result. But as complexity of the tasks progress the larger size of the network is needed, which makes model uninterpritable. By now complicated models are a black box that no one knows how it performs inside nor what parts of it are responsible for what [9]. This problem will be partially covered in later chapters.

Chapter 2

Theory

2.1 Feedforward Neural Networks

Neural Networks are mathematical models with a learning architecture similar to the brain algorithm of storing information [8]. They are also known as function approximation algorithms that are trying to approximate function that would fit provided input data. Previous statement also arises the question, considering some large function, is there any model that could approximate such function? And the answer is provided by universal approximation theorem. That states that there always exist a feedforward Neural Network with some architecture that can approximate such a function.

All types of neural networks are commonly referred as Artificial Neural Networks (ANN) since they are used in conjunction with some machinery, any type of mechanical or electrical device which have to perform certain actions in different environments. ANN's can learn complex patterns from different sources of data, such as images, signals from sensors, etc., and make a decision of actions that will be proceeded to some device or mechanism.

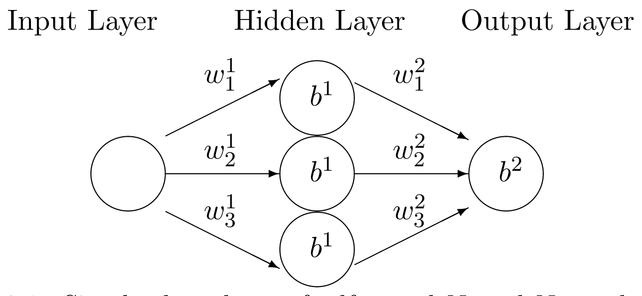


Figure 2.1: Simple three layers feedforward Neural Network model with weights w and biases b marked respectively to each layer.

In a nutshell, mathematical models, neural networks, are assembled from cells, later referred to as neurons. They are interconnected with each other in the path, which allows the transfer of the data between the connected cells. The neuron cell's main tasks in the network are extracting and memorizing features from the input data. Extraction and memorization processes are done by varying initial neurons coefficients named weights $\boldsymbol{W}_n$ and biases $\boldsymbol{b}_n$, where n is a n-th layer.

$$\boldsymbol{a}_n = \boldsymbol{W}_n \boldsymbol{a}_{n-1} + \boldsymbol{b}_n, \tag{2.1}$$

Stacking neurons in layers allow extracting a big chunk of features at a network step. General NN architecture consists of layer types like an input layer, an output layer, and a hidden layer. The purpose of the first two is hidden in their names. As for the last one, it performs the main feature extraction computation. The whole algorithm of proceeding data through the simple network looks this way Fig. 2.1:

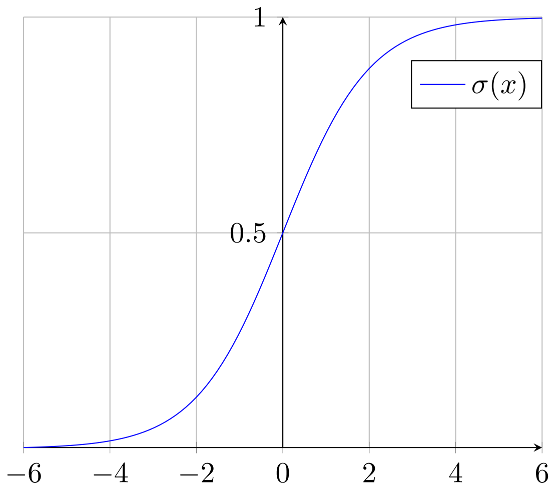


Figure 2.2: Graph of a Sigmoid activation function.

(i) The Input layer accepts encoded input.

(ii) Than input is forwarded through the hidden layer by computing every hidden neurons activation Eq. 2.1.

(iii) The final result is than received by the output layer.

The calculation of the neuron activation is done by applying the weights of the dedicated neuron to the input and insert it inside the activation function. The activation function is a function, which determines the output of the neuron. Those functions are influencing the network's ability to converge as well as convergence speed. Choosing the proper activation function is essential for neural network performance. For the rest of the paper and all experiments Sigmoid function will be used Eq. 2.2, Fig. 2.2.

$$\sigma = f(\boldsymbol{a}_n) = \frac{1}{1+e^{-\boldsymbol{a}_n}}, \tag{2.2}$$

where $\boldsymbol{a}_n$ is n-th layer activation. General mathematical formula of the second network algorithm step (ii) was described in Eq. 2.1, where n is a layer number, $\boldsymbol{W}_n$ is a matrix of weights of each neuron in the n-th layer, $\boldsymbol{b}_n$ is a vector of biases in n-th layer, $\boldsymbol{a}_n$ is the n-th layer vector output, and $\boldsymbol{a}_{n-1}$ is an output vector of $n-1$-th layer. The last layer usually denotes as n, will be the final result of NN.

In machine learning, there is a way of training such networks. It is called supervised learning. This method allows training NN by forwarding the data through the model and then examining the difference between expected and outputted results, allowing the network to improve performance based on previous experiences. This method is used for all kinds of real-world computational problems. But such a technique requires well-labeled data, which sometimes takes a massive amount of time since it has to be done manually.

Now, let's look at more mathematical details behind the learning process. Comparing received output with the expected one is done by the so-called loss function. One of which is called Mean Square Error (MSE) will be used throughout this thesis. It works by subtracting the desired result from the received one squared

$$L(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2}\sum_{i}^{N} ||y_i - NN(x_i)||^2, \tag{2.3}$$

where N is a length of the vectors inputed to the loss function,$NN(\boldsymbol{x}_i)$ is NN output from data point $\boldsymbol{x}_i$, $\boldsymbol{x}$ is an input vector, and $\boldsymbol{y}$ is a labeled vector. The loss function is an indicator of changes performed to the network, as a result, minimizing value of this function will train the network. The minimizing part is done by making adjustments to the model weights with respect to the loss function value, which is done by Gradient Descent (GD). Gradient Descent - is a mathematical algorithm for finding the local minimum of a function by computing the gradient and performing steps in the opposite direction of the gradient.

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \nu \nabla_\theta f(\boldsymbol{x}_n, \theta), \tag{2.4}$$

where $\boldsymbol{\theta}_{n+1}$ is newly computed parameter, and $\boldsymbol{\theta}_n$ is the previous parameter, $\nabla_\theta f(\boldsymbol{x}_n, \theta)$ is the gradient of the loss function with respect to the parameter. The gradient part can be regulated using the learning rate ν, choosing proper learning rate parameter may significantly change the learning process. By increasing it to much the algorithm will go around the desired minimum but will never reach it. On the other hand, making it to small will lead to long training time.

Nevertheless, this algorithm has a few issues that have to be mentioned. Computational speed is slow because Eq. 2.3 iterates through whole data, which can be very large. Thus the bigger the training set, the longer it takes to compute the gradient of the loss function. The second issue is a local minimum problem. In shorter terms, it will be impossible to omit the local minimum by going in the proportional direction, which makes the full learning process stop without reaching the actual minimum.

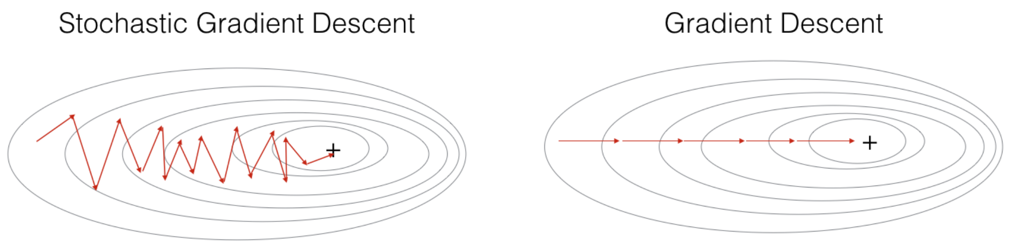


Figure 2.3: Image showing difference between SGD and GD mini-batching [5].

Those issues were addressed in a new version of GD called Stochastic Gradient Descent (SGD) 2.6. The SGD is still the same as GD, with the exception of performing gradient operations not on the entire data set but on the small subsets called batches. Now the loss function computes only difference between some training and label batch $\boldsymbol{x} = [x_1, x_2, ..., y_m]$, $\boldsymbol{y} = [y_1, y_2, ..., y_m]$ of length $m << N$

$$L(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2} \sum_{i}^{N} ||y_i - NN(x_i)||^2, \tag{2.5}$$

which makes gradient make oscillations, allowing it to avoid local minimum more frequently and increase computational speed.

The algorithm that inherits the SGD methodology for training feedforward networks is Backpropagation. Backpropagation calculates the error of the last layer and than as in name states backpropagates it to every previous layer of the model, simultaneously updating the network weights.

Now the full steps of performing such learning can be numerated:

(i) The Input layer accepts the data batch.

(ii) Than input is forwarded through the hidden layer by computing every Hidden neurons activation Eq. 2.1.

(iii) The final result is than received by the output layer.

(iv) The final result is compared to the expected one, label, and loss is computed.

(v) The gradient of the loss with respect to the NN weights is computed, Eq. 2.9, 2.8.

(vi) The last step is to backpropagate the loss to every layer and update the layer weights and biases, Eq. 2.6,2.7, and repeat over the step (i) until desired accuracy is reached or the last chunk of data have been processed.

$$\boldsymbol{w}_l = \boldsymbol{w}_l - \nu \frac{\partial L}{\boldsymbol{w}_l}, \tag{2.6}$$

$$\boldsymbol{b}_l = \boldsymbol{b}_l - \nu \frac{\partial L}{\boldsymbol{b}_l}, \tag{2.7}$$

where $\boldsymbol{w}_l$ is a weight vector of the layer l and $\frac{\partial L}{\boldsymbol{w}_l}$ is the gradient of the loss with respect to weight, accordingly $\boldsymbol{b}_l$ is a bias vector of the layer l and $\frac{\partial L}{\boldsymbol{b}_l}$ is the gradient of the loss with respect to bias, ν is a learning rate, which dictates of how much change will from the gradient will be proceeded to the weights.

As it is stated in the chain rule the gradient can be derived as follows:

$$\frac{\partial L}{\partial \boldsymbol{w}_l} = \frac{\partial L}{\partial \boldsymbol{a}_l}\frac{\partial \boldsymbol{a}_l}{\partial \boldsymbol{w}_l} = (\boldsymbol{y} - NN(\boldsymbol{x}))f'(\boldsymbol{w}_l\boldsymbol{a}_{l-1} + \boldsymbol{b}_l)\boldsymbol{a}_{l-1}, \tag{2.8}$$

$$\frac{\partial L}{\partial \boldsymbol{b}_l} = \frac{\partial L}{\partial \boldsymbol{a}_l}\frac{\partial \boldsymbol{a}_l}{\partial \boldsymbol{b}_l} = (\boldsymbol{y} - NN(\boldsymbol{x}))f'(\boldsymbol{w}_l\boldsymbol{a}_{l-1} + \boldsymbol{b}_l), \tag{2.9}$$

$$\boldsymbol{\delta}_l = (\boldsymbol{y} - NN(\boldsymbol{x}))f'(\boldsymbol{w}_l\boldsymbol{a}_{l-1} + \boldsymbol{b}_l), \tag{2.10}$$

where $\boldsymbol{\delta}_l$ is the error of the last layer l. In order to backpropagate the error to the rest of the layers, $l-1$ layer equation have to be expressed in terms of the error of l layer Eq. 2.10, and thus will look as follows:

$$\boldsymbol{\delta}_{l-1} = \boldsymbol{w}_l\boldsymbol{\delta}_l f'(\boldsymbol{w}_l\boldsymbol{a}_{l-1} + \boldsymbol{b}_l) \tag{2.11}$$

$$\frac{\partial L}{\partial \boldsymbol{w}_l} = \boldsymbol{a}_{l-1}\boldsymbol{\delta}_l \tag{2.12}$$

$$\frac{\partial L}{\partial \boldsymbol{b}_l} = \boldsymbol{\delta}_l. \tag{2.13}$$

2.2 Regularisation

Small NN can perform just fine for simple problems, staying very explanatory and giving the proper precision. But as soon as problems became more complex, much larger networks are required. The bigger the number of free parameters, the wider the range of phenomenons this model can describe. But, unfortunately, the large number of parameters influence the model's ability to make predictions in environments it has never seen before. It was observed that after the training, those networks have all one problem with accuracy values received on the test data. This problem is also known as overfitting. Overfitting as were stated in the previous sentences is collectively a type of NN problems that happens when model was trained on a wide range of data, showing remarkable results on the provided data set, but as soon as data not shown before appears model performance drops seriously.

To solve this problem the set of the process were designed called Regularisation methods. Those methods aims to reduce overfitting in large models, allowing them to show better test and validation data results.

One of the regularisation types that will be mentioned in this section is called L1 regularisation. Generally speaking, regularisation methods focus on making big neural network models prefer learning smaller weights rather than big ones. It is done by adding an extra component to the loss function

$$L = MSE + \lambda \sum_{w} |\boldsymbol{w}|, \tag{2.14}$$

where MSE is the loss function as described in Eq. 2.3, and $\boldsymbol{w}$ is a weights of the model layer, λ is a regularisation parameter. This adjustment will allow filtering large weights that could improve the model from the one that could not, which results in small weights. Combine that with models that have small number of hidden layers and it will bring the most transparent network possible. Transparency in neural networks is a feature of a model to give clear understanding of how much every weight adding to the output, which makes this model easier to understand and debug.

2.3 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are artificial neural networks developed as an extension to dense networks to work more efficiently with sequential data.

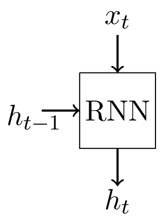


Figure 2.4: The one iteration RNN, with $\boldsymbol{x_t}$ as an input, $\hat{\boldsymbol{h}}_{\boldsymbol{t}}$ as an output and $\boldsymbol{h}_{t-1}$ as a previous or hidden state.

DNN's computes the output from every input without considering connection between previous and new computations, which makes it hard for this type of network to work with sequences. On the other hand, Recurrent Networks are made of recurrent connections, which allows them to take the output as additional information from previous computations and pass it to the next layer. Performing such forwarding requires those types of network somehow to save such an information. It is done with the extra variable that every recurrent layer posses that is also called as the hidden state.

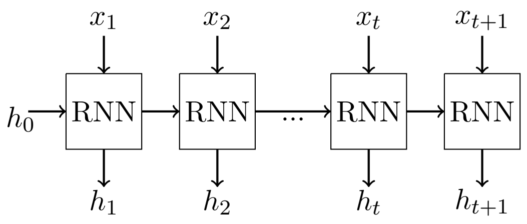


Figure 2.5: Unfolded RNN of the input sequence vector $\boldsymbol{x} = [x_1, x_2, ..., x_{t+1}]$, where h_0 is the initial hidden state and every next computed output h_t is then forwarded to the next step.

In order to use this variable, recurrent layers share the same weights within the network. The weights are adjusted parts of the RNN. General formula that RNN is using to compute the output is as follows

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = \tanh(\boldsymbol{W}_i\boldsymbol{x}_t + \boldsymbol{W}_h\boldsymbol{h}_{t-1} + \boldsymbol{b}), \tag{2.15}$$

where $\boldsymbol{h}_{t-1}$ is the previous hidden state and $\boldsymbol{h}_t$ is a newly computed output, which will be proceeded for the next computation, $\boldsymbol{W}_i$,$\boldsymbol{W}_h$,$\boldsymbol{b}$ are a parameters of RNN. To make the first computation of the time $t = 0$, the recurrent layer must have a predefined hidden state $\boldsymbol{h}_0$. Usually, it is initialized to zero since we do start from scratch. It is straightforward to compute with those networks data sequences of many sizes. As well as computing sequences of the same size at once. In the Fig. 2.5 sequence of values $\boldsymbol{x}_t$ can be changed to the sequence of vectors of the same size $\boldsymbol{x}_t$.

The Backpropagation of RNN is differ from the same algorithm proceeded for DNN . The first difference in the loss since we have t outputs for every time step. Thus the full loss is a sum of all unrolled layer errors

$$L(X, Y) = \sum_{t=0}^{n} E_t = \sum_{t=0}^{n} ||\boldsymbol{h}_t - \boldsymbol{d}_t||, \tag{2.16}$$

where L is the loss of the model, E_t is the error or difference between the desired and received output of t-th step and n is the length of the sequence. Thus the weight adjustment will change as well

$$\boldsymbol{W} = \boldsymbol{W} - \nu \sum_t \frac{E_t}{\boldsymbol{W}} \tag{2.17}$$

As a result of storing a hidden state value at every layer, RNN's are very slow to train. Those networks as well are suffering from the problems of explosive and vanishing gradients. Gradient vanishing is an issue, which occurs during the learning process. When the gradient is too small, it has tendencies to continue becoming even smaller, which makes it update weights until they become zero. The previously mentioned problem will lead to such weights inability to learn anything from the training data.

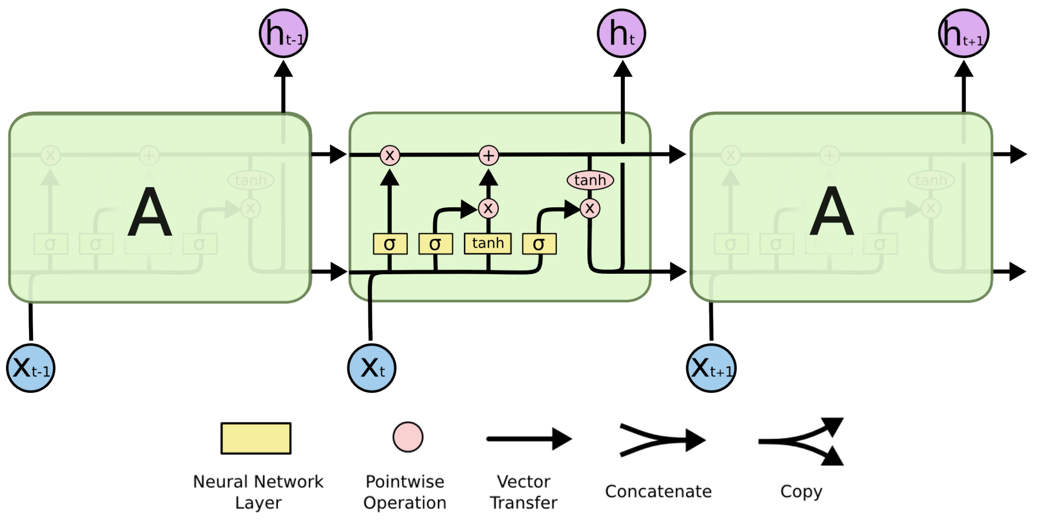


Figure 2.6: Descriptive picture with notation of the LSTM cell work [12].

As were mentioned previously, in theory RNN should have had been able to work properly with very long sequences of data, but in reality the opposite was observed. Solutions to the problems is updated RNN or as it called Long-Short Term Memory (LSTM) [7].

Instead of memorizing everything shortly LSTM have a gating mechanism, which allows it to chose at every time step if it should add or disregard additional information to the cell state.

Considering all previously mentioned things about RNN and LSTM it is now time to find where those models are the most useful. For example, predicting next letters or words since the network has to consider previously predicted one. The same holds for sound processing, predicting stock market etc.

2.4 Neural Accumulator

Neural Networks can learn any function in theory, but they have some issues with generalizing learning outside the training range. Hence new direction appeared called Neural Arithmetic, which aims to make models that can consistently extrapolate with numerical computations. The first of such models is Neural Accumulator (NAC). The Neural Accumulator - is a NN architecture presented here [16], that was directed to approximate functions with simple arithmetical operations like addition and subtraction. It was accomplished by merging network weights and restricting the final result to take values -1,1 or 0. To restrict the values of final weight to be -1,1 or 0, tanh and σ functions were used, and the final weight matrix is now looking this way

$$\boldsymbol{W} = \tanh(\hat{\boldsymbol{W}}) \cdot \sigma(\hat{\boldsymbol{M}}), \tag{2.18}$$

where $\hat{\boldsymbol{W}}, \hat{\boldsymbol{M}}$ are actual model weights matrices. The work procedure of this model looks as follows In Fig. 2.7, inputs are arbitrary numbers or vectors in the range of training data. The matmul is our accumulator, which multiplies input to the corresponding weights

$$\boldsymbol{a} = \boldsymbol{W}\boldsymbol{x} \tag{2.19}$$

and defines what actions we are doing with our input. Thus the network output is a weighted sum of the inputs.

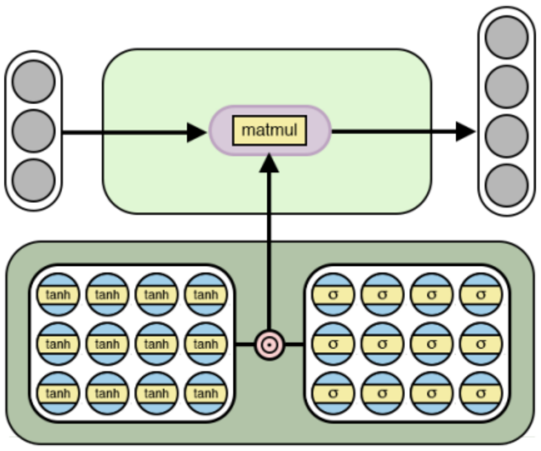


Figure 2.7: NAC architecture [16], bottom dark green space is our weights matrices $\hat{\boldsymbol{M}}$ and $\hat{\boldsymbol{W}}$, light green field is where model perform multiplication on input, received from the left, to the weight matrix computed from below.

2.5 Neural Arithmetic Logic Unit

Previous model were designed to learn simple addition and subtraction operations. But those are not enough to learn any complex function. Thus the new model was developed, Neural Arithmetic Logic Unit (NALU). NALU - is a NN architecture that consists of multiple NAC modules with an additional control gate, which allows switching between those. Now let's look in detail at what is happening inside the NALU model.

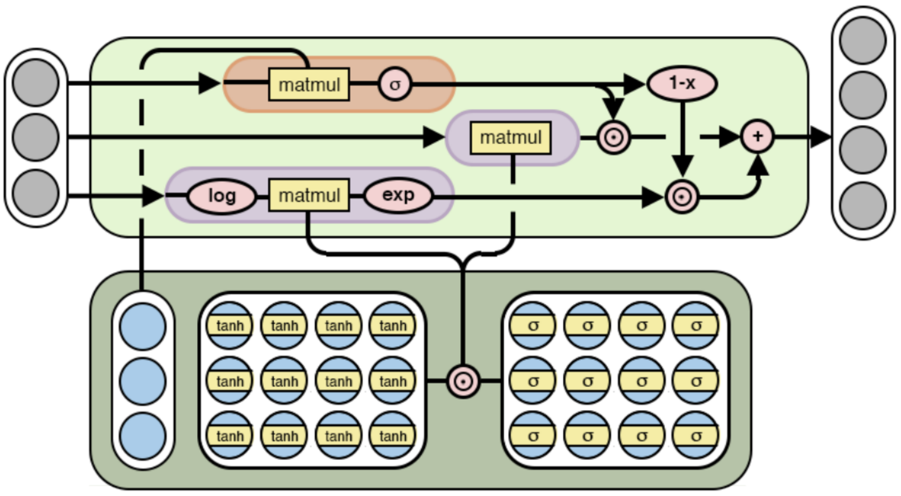


Figure 2.8: NALU architecture [16], where purple matmuls are NAC modules, one ordinary and another one with two additional functions before and after, matmul marked orange is a control gate.

As was mentioned previously in this section NALU does have two NAC models, but in reality, it is one NAC that shared weights that are copied and split as presented in Fig. 2.8. The extra function between the NAC matmul, which is marked as a yellow rectangle, is there to change the input flow to allow this brunch to learn multiplication or division. The orange marked matmull with additional σ function is the so-called control gate. A control gate is a method that tries to activate or deactivate NAC model outputs depending on the arithmetical operations in the function that NALU is trying to approximate.

Finally, let's look at a mathematical expression that defines the whole NALU network

$$NALU : \boldsymbol{y} = \boldsymbol{g} \odot \boldsymbol{a} + (1 - \boldsymbol{g}) \odot \boldsymbol{m}, \tag{2.20}$$

where $\boldsymbol{a}$ is the NAC output vector shown previously in Eq. 2.19, $\boldsymbol{m}$ and $\boldsymbol{g}$ are

$$\boldsymbol{m} = \exp \boldsymbol{W}(\log(|\boldsymbol{x}| + \epsilon), \boldsymbol{g} = \sigma(\boldsymbol{Gx}), \tag{2.21}$$

where $\boldsymbol{m}$ is an modified NAC output that is capable of learning multiplication and division, $\boldsymbol{g}$ is a control gate, $\boldsymbol{W}$ and $\boldsymbol{G}$ are the learning parameters, ϵ is a value that prevents zero input inside the log and $\boldsymbol{x}$ is an input vector.

$$f(\boldsymbol{x}) = \prod x_i^{w_i} \tag{2.22}$$

All those together allow this complex architect model to approximate functions that consists of addition, subtraction, multiplication, division.

However, the NALU model has some issues that stop it from being state of the art for Neural Arithmetic. For example, Suppose one of the most critical NALU mechanisms that allow approximating such a wide range of functions, the gating mechanisms, misbehaves. In that case, it will eventually lead to an inability of the correct NAC module to receive gradient signals. It was observed [10] that this mechanism in NALU works in almost a random way, which disables correct approximation of the functions. NALU also has problems with negative numbers and smaller numbers than ϵ. It also suffers from large gradients, which makes it even more inapplicable to specific tasks.

Nevertheless, this model was the first one that tried to bring some solution to the Neural Arithmetic word, but not the last one.

2.6 Neural Addition Unit

There also exists a much simpler and yet much better solution for addition and subtraction. It is called Neural Addition Unit (NAU) presented here [10] in 2020. Analysing NAC weights values it was empirically observed, that in practice Eq. **??** does not achieve this -1,1 or 0 vales clipping. To achieve results better than NAC with addition and subtraction, it was proposed instead of restricting the values of the weight, instead to map them to the [-1,1] range

$$\hat{\boldsymbol{W}} = \min(\max(\boldsymbol{W}, -1), 1), \tag{2.23}$$

where $\boldsymbol{W}$ model weights, which are clamped to [-1,1] boundary at every iteration and $\hat{\boldsymbol{W}}$ is a result that going to be used to compute the output. That allowed to outperform the NAC model results and decrease model complexity by making it more transparent. The full NAU model looks as follow

$$\text{NAU}: \quad \boldsymbol{a} = \hat{\boldsymbol{W}}\boldsymbol{x}, \tag{2.24}$$

where $\boldsymbol{a}$ is a newly computed output vector from weighted input, $\hat{\boldsymbol{W}}$ is newly computed matrix as in Eq. 2.23, and $\boldsymbol{x}$ is an input vector.

For the experiment part, the NAU weights will not be clipped. Since in the initial paper authors were intended to learn only +,- or 0. But in the experiment section functions also contain extra constant component like $3x$. Hence not clipping the weights will help to learn such variables.

Intuitive understanding of NAU comes from its name. So for now lets treat it like a simple box that takes some input vector $\boldsymbol{x}$ and makes weighed sum of the elements. So lets take a look at at vector $\boldsymbol{x}$ of size 3

$$\begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} ax_1 + bx_2 + cx_3 \end{bmatrix} \tag{2.25}$$

Nevertheless, such model can not approximate well equations with a constant inside, since it was not intended by design. Lets assume that we have a model with three hidden layers NAU(1,3). The input vector from equation $f(x) = x + 2$ will be proceeded through this model as follows

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} ax_1 \\ bx_2 \\ cx_3 \end{bmatrix} \tag{2.26}$$

where a, b, c are NAU weights values. The problem is, NAU weights are incapable of learning the +2 constant. Thus any model that contains this type of layer has to expect and somehow compensate.

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} * [x] + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} w_1x + b_1 \\ w_2x + b_2 \\ w_3x + b_3 \end{bmatrix} \tag{2.27}$$

Interesting fact is Dense network are capable of easily predicting function constants since they have bias term, which is basically a learnt constant, as shown in Eq. 2.27.

2.7 Neural Multiplication Unit

The writers of the NAU paper also [10] introduced there one extra model that was capable of learning multiplication more efficient than mentioned NALU. The Neural Multiplication Unit (NMU) is a neural arithmetic model, which was suggested as a simple way to learn multiplication. Even though NMU is incapable of division, as an opposed model in the referenced paper, it converges much faster and can work well with negative numbers. NMU weights look similar to NAU ones and are computed this way

$$\hat{\boldsymbol{W}} = \min(\max(\boldsymbol{W}, 0), 1), \tag{2.28}$$

where $\boldsymbol{W}$ is a weight matrix of the model. The complete mathematical algorithm behind the NMU looks as follows

$$NMU : \boldsymbol{f(x, \hat{W})} = \prod_i (\hat{\boldsymbol{W}}\boldsymbol{x} + 1 - \hat{\boldsymbol{W}}), \tag{2.29}$$

where $\hat{\boldsymbol{W}}$ is a weight matrix computed in Eq. 2.28, $\boldsymbol{x}$ is an input vector, +1 there to deal with $\hat{\boldsymbol{W}} = 0$. In Eq. 3.3, we now may observe that NMU is not bounded to value of ϵ, as NALU does, and also can properly work with negative or positive input values since it do not have inner sign restriction functions.

Intuitive way of understanding how does the model proceeds the input is as follows

$$\begin{bmatrix} \hat{w}_{11} & \hat{w}_{12} \\ w_{21} & w_{21} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} \hat{w}_{11}x_1 \cdot \hat{w}_{12}x_2 & w_{21}x_1 \cdot w_{22}x_2 \end{bmatrix}, \tag{2.30}$$

where first matrix is an NMU model weights, and $[x_1, x_2]$ is an input vector. From the equation above it is clearly seen how NMU can approximate polynomial type of functions by just performing multiplication on the input.

In the paper [10] was also mentioned how regularising this layers weights clip them to -1,1 or 0 values, which allows this model converging to the simplest solutions.

Both NAU and NMU show high standards for addition and multiplication, making state of the art for learning functions with addition and multiplication operations inside. However, their limited learning capability does not allow them to be an individual model but rather conjunction to something more advanced. Nevertheless, stacking those model together in order or using them as a layers in more complex networks, will eventually becomes a potential solution to the Neural Arithmetic problem capable of showing extrapolating results. In addition stacking NAU, NMU layers together is way better than gating mechanism, that was presented in NALU, which have huge convergence issues. The tasks that stacked model can solve are approximating functions with addition, subtraction, multiplication operations inside, type of functions that have all of the mentioned operations inside is polynomial one. And indeed it will be observed in the experiment section how good this model is in approximating polynomials.

2.8 Neural Power Unit

The last model that was proposed is Neural Power Unit (NPU) that was published by my supervisor and his colleagues [6]. Previous models demonstrate great results for two simple arithmetical operations, addition and multiplication. Still, there was no satisfactory score for the rest of the arithmetical operations as well as all mentioned solutions have some strong and weak sides in their implementations individually.

In a nutshell, NPU tries to improve the NALU work by changing some architectural decisions made in its equations. Its goal is to show practical solutions for learning division, multiplication, and square root operations. All three arithmetic methods can be interpreted as just a product of power functions

$$f(\boldsymbol{x}) = \prod x_i^{w_i}, \tag{2.31}$$

where $\boldsymbol{x}$ is an input vector, and w_i is a power value, which can be natural, integer, or rational number. And that is exactly what NPU tries to do. It tries to learn the power value parameter for every input.

To achieve those results, modification to the Natural Arithmetic Logic Unit was done in Eq. 2.21, more precisely in the $\boldsymbol{m}$ value part. The updates were done by adding complex logarithm and complex weight to the Eq. 2.21. The received equation looks this way

$$\boldsymbol{m} = \exp\left((\boldsymbol{W}_{real} + i\boldsymbol{W}_{img}) \log \boldsymbol{x}\right), \tag{2.32}$$

where $\boldsymbol{W}_{real}$ are the real part of the model weights, and $\boldsymbol{W}_{img}$ is an imaginary part of the weights, i is imaginary number. The necessary changes allow for processing correctly positive and negative values and give extra flexibility to work with complex weights. Since the imaginary part is not needed for the real solutions, it can be omitted. The math interpretation after the transformation looks this way

$$NPU : \boldsymbol{y} = \exp\left(\boldsymbol{W}_r \log \boldsymbol{r} - \pi \boldsymbol{W}_i \boldsymbol{k}\right) \odot cos(\boldsymbol{W}_i \log \boldsymbol{r} + \pi \boldsymbol{W}_r \boldsymbol{k}), \tag{2.33}$$

$$\boldsymbol{r} = \hat{\boldsymbol{g}} \odot (|\boldsymbol{x}| + \epsilon) + (1 - \hat{\boldsymbol{g}}), \quad k_i = \begin{cases} 0, & \text{if } x_i \geq 0 \\ 1, & \text{otherwise} \end{cases}, \quad \hat{g}_i = \min(\max(g_i, 0), 1), \tag{2.34}$$

where $\boldsymbol{r}$ is our input, and $\boldsymbol{k}$ is a switch, which is influenced by the sign of the input data, $\hat{\boldsymbol{g}}$ is gating mechanism that does complicated work of choosing which direction to go. In conjunction with NAU, appeared model can approximate all range of functions constructed with simple arithmetical operations.

Even this Neural Arithmetic model experience some issues with convergence, but increasing number of parameters will eventually solve it.

2.9 Arithmetic models in Recurrent Networks

In the Recurrent Neural Network section 2.3, it was described how do they work. It is essential since this section will try to extend some of its functionality for some different experiments with the data sequences. The previously mentioned Arithmetic models will extend RNN functionalities to achieve better results in extrapolation than the state-of-the-art LSTM.

The Arithmetic Recurrent Neural Networks (ARNN) are the same RNN except it has replaced original weights to one of the Neural Arithmetic Logic Module (NALM) mentioned earlier. So after some changes, the RNN equation will look like this

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = NALM(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) + \boldsymbol{b}, \tag{2.35}$$

where $NALM()$ is one of the arithmetic models, $\boldsymbol{x}_t$ is input, $\boldsymbol{h}_{t-1}$ is previous hidden state and $\boldsymbol{b}$ is bias. The tanh function was erased since it bounds the output to the predefined range, which prevails computing with such an architecture.

```julia
function (m::ArithmeticRNNcell)(h, op, x)
    s,op_net, a_net, b = m.s,m.Wi, m.Wh, m.b
    ho = op_net([op;h .* ones(Float32,size(x))])
    h = a_net([x;ho]) .+ b
    sz = size(x)
    return h, reshape(h, :, sz[2:end]...)
end
```

Listing 2.1: Julia code example for Recurrent part.

Chapter 3

Experiments

3.1 Simple functions

Experiment section is aiming to show practical example of the Arithmetic Models usage, as well as showing their performance. This experiment will approximate different types of functions to observe behavior with data outside the training range. Programming models is not a complicated task using modern and powerful libraries, for example one of the Julia libraries available in GitHub (https://github.com/FluxML/Flux.jl). In this experiment, five simple functions $f(x) = x^2$,$f(x) = |x|$, $f(x) = \sqrt{x}$,$f(x) = \ln x$, $f(x) = -0.1x^4+3x^2+2$ was approximated by four different models Tab. 3.2.

Experimented functions	Training data size	Training grid range	Training step dx	Testing grid range	Testing step dx
Polynomial	80000	[-1.5,1.5]	0.67	[-3,-1.5],[1.5,3]	0.67
Absolute value	80000	[-1.5,1.5]	0.67	[-3,-1.5],[1.5,3]	0.67
Square root	80000	[0,1.5]	0.67	[1.5,3]	0.67
Natural logarithm	80000	[0.01,1.5]	0.67	[1.5,3]	0.67
Parabola	80000	[-1.5,1.5]	0.67	[-3,-1.5],[1.5,3]	0.67

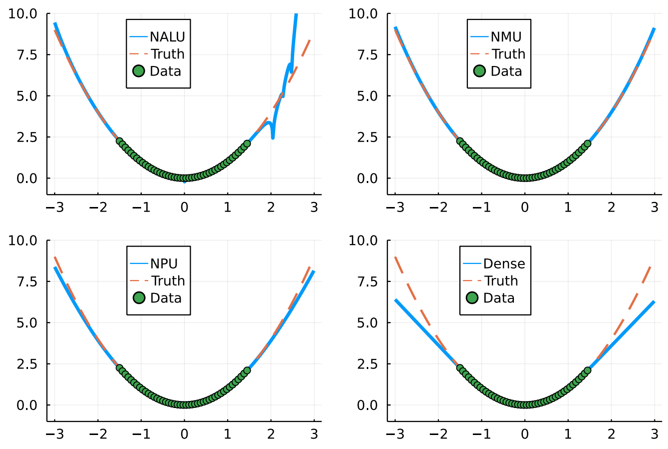
Table 3.1: Data generation settings for the Simple functions experiment section.

Models	Architecture
NALU	NALU(1,24),NALU(24,1)
NMU	NAU(1,12),NMU(12,12),NAU(12,1)
NPU	NPU(1,24),NAU(24,1)
Dense	Dense(1,24,sigmoid),Dense(24,1)

Table 3.2: Models architectures for the Simple functions experiment section.

In every experiment models architecture and training settings were the same, Tab. 3.1. The mean squared error (MSE) loss function were used, which was mentioned previously in this paper.

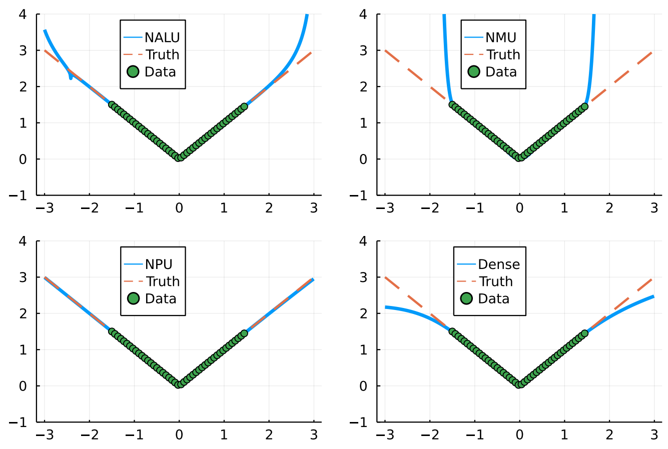
Experiment proceeded as follows, every model were trained with the same amount of training data for the same number of epochs and than were tested with the data in the test range. In order to gather more representative data number of neurons were chosen optimally, meaning some manual training was done. The data were generated as static grid in ranges described here Tab. 3.1.



Models	Train	Test
NALU	6.97e-7	5.48
NMU/NAU	**$3.52e-11$**	**$1.49e-3$**
NPU/NAU	1.74e-6	2.26e-2
Dense	4.62e-7	2.22e-1

Figure 3.1: Left image represent experiment results of every model as a graph, where Truth line is a graph of a function, Data are points that were used in training models, and blue line represents each model extrapolation results. Right table shows numerical representation of the graphical results on the left. Approximated function is parabola

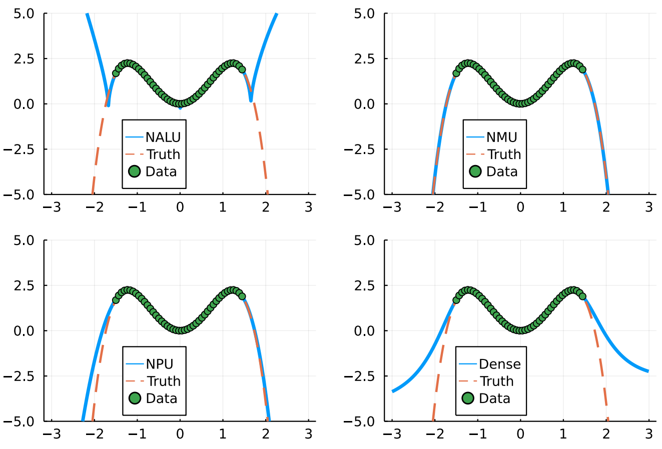
In the image Fig. 3.1 the simple parabola function was approximated by 3 different NALM's and one Dense models. Results above show that NALU, NPU, and NMU models can extrapolate parabola, when the Dense Network can not. The DNN shows the problem of extrapolation in models containing only dense layers very clearly. As expected from NPU, its architecture allows it to do fractional powers x^w, which is precisely the case of the simple parabola function. The results are not that great because the tested model have some convergence problem. NPU mode is used in conjunction with NPU to allow such a network better approximate functions with different arithmetical operations. As intended for the NMU model, it performs best with tasks it was initially made for and outputs exceeding extrapolation results. The NALU case did well, but it is hard for this model to converge to the real solution.



Models	Train	Test
NALU	**$1.07e-7$**	1.30e-1
NMU/NAU	1.25e-3	6.24e6
NPU/NAU	1.32e-7	**$1.13e-4$**
Dense	4.22e-5	6.31e-2

Figure 3.2: The same as in Fig. 3.1. Approximated function is absolute value.

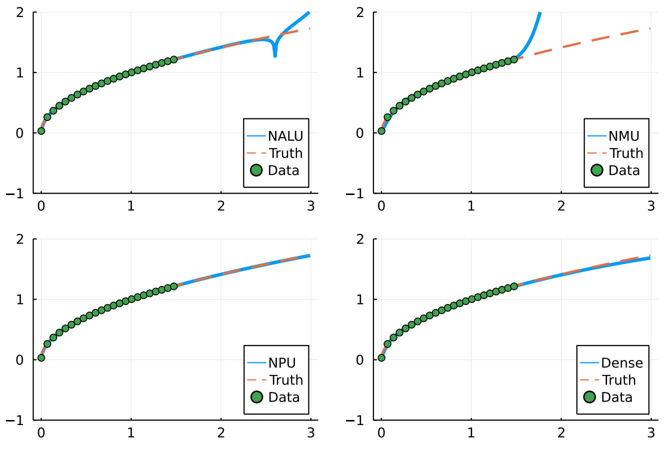
In the Fig. 3.2 we see that Dense layer model performance again shows that it can not extrapolate well, especially when approximated function is not similar to its activation. This result can be increased by changing model activation function. The NPU network achieved great results. In NALU case absolute value operation is originally inside its implementation in the multiplication part $\exp(\boldsymbol{W}(\log|x| + \epsilon))$, which resulted in good training results. But it as soon as it gets to far from the training data range it experience with converge to the real solution. NPU shows such exceeding results because it tries to approximate mirrored square root functions and the NAU layer adds them together. Since NMU/NAU only can do multiplication and addition of the input they can not approximate absolute value function.



Models	Train	Test
NALU	1.65e-6	4.01e2
NMU/NAU	$\mathbf{2.36e-9}$	$\mathbf{1.08e-1}$
NPU/NAU	2.44e-4	6.36e1
Dense	8.36e-5	2.26e2

Figure 3.3: The same as in Fig. 3.1. Approximated function is polynomial, $f(x) = -x^4 + 3x^2$.

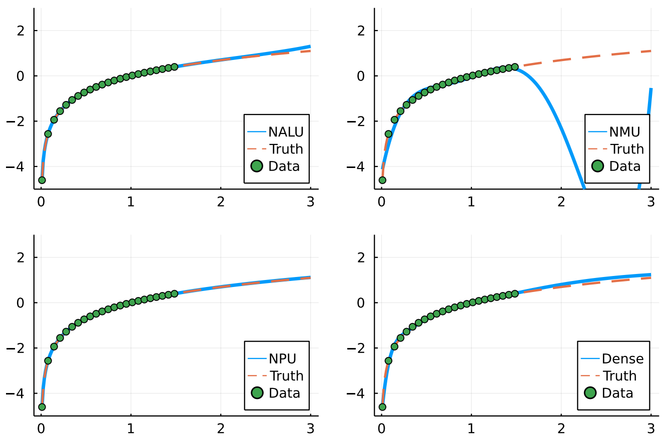
The extrapolation problem in the Dense network ,left part of Fig. 3.3, appears as soon as tested points are going outside the boundaries of the training range. Results received in Fig. 3.3 for the NALU model shows problems of its architecture. As were mentioned in the Theory part this model suffers from convergence issues. NMU/NAU model can do such tasks with such outstanding accuracy due to its initial reasons of implementation, approximate functions with multiplication and addition operations. The model also shows a great convergence for this function. The NPU/NAU model shows good results for this task, as expected from two modules, separately best at approximation power and addition operations.



Models	Train	Test
NALU	2.94e-8	3.14e-3
NMU/NAU	2.93e-4	6.32e2
NPU/NAU	$\mathbf{1.19e-8}$	$\mathbf{2.93e-4}$
Dense	3.07e-6	3.74e-4

Figure 3.4: The same as in Fig. 3.1. Approximated function is square root.

From the observation of the square root case at Fig. 3.4 Dense layer network numerical results in training part present its ability to learn well the training data. Also show not great results with a testing data, which is due to its activation function σ. The NPU model did very grate almost fully converged to the function, unlike NMU and NALU. In NALU case it again can experience issues with convergence, which is very common thing for this model. NMU model layers, NMU and NAU, were not designed to approximate square root function, meaning it is expected that models that was not intentionally created for a specific type of operations will show any reasonable result. That is why after the training range, it tries to approximate the tested function as some sort of polynomial.



Model	**Train**	**Test**
NALU	$\mathbf{1.15e-8}$	2.72e-3
NMU/NAU	2.86e-2	1.44e-1
NPU/NAU	1.31e-6	$\mathbf{4.74e-5}$
Dense	4.21e-3	6.95e-3

Figure 3.5: The same as in Fig. 3.1. Approximated function is natural logarithm.

Results and reasons those results were received in Figure 3.6 are similar to the previously mentioned.

Tested functions are functions of rational type. Now experiment will be conducted using one that is irrational, to observe whether proposed NALM's can extrapolate irrational types of functions. For the previous experiments as data were used random uniformly distributed sample. For this experiment predefined grid will be used in order to avoid undefined points

$$f(x) = \frac{x^3}{(x-0.5)*(x+0.5)} \tag{3.1}$$

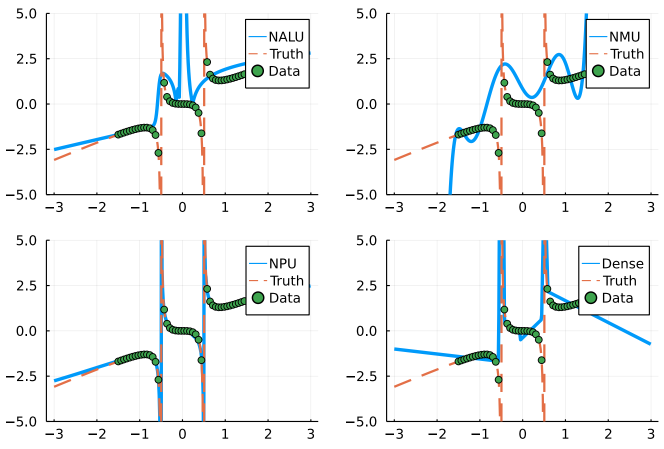
In the Eq. 3.1, we may see irrational polynomial function that has inf value at $x = \pm 0.5$ point. Avoiding this point in the experiment is vital because there exist no Neural Network models that can work with value of that type. To avoid this point our training and testing data range will look this way Not only the data have encounter transformation but also models architecture. Now it looks as follows

Models	Architecture
NALU	Dense(1,10),NALU(10,1)
NPU	Dense(1,10),NPU(10,1)
NMU	Dense(1,10),NMU(10,10),NAU(10,1)
Dense	Dense(1,10,sigma),Dense(10,10,sigma),Dense(10,1)

Table 3.3: Models architectures for the experiment with irrational type of function Eq. 3.1

The Dense layer part in every model make the $x - 0.5$ and $x + 0.5$ in Eq. 3.1 terms possible via bias. Not only that but in order to receive any results the learning rate variable have to be chosen small other wise it would make models unable to perform any score.

The observation from the images above shows that the only models that are capable of getting close to approximating tested irrational function are NPU and Dense. For NPU



Model	Train	Test
NALU	1.61e1	3.13e2
NMU/NAU	1.6e1	1.28e6
NPU	$\mathbf{5.21e-3}$	$\mathbf{1.85e-1}$
Dense	7.81e-1	7.58

Figure 3.6: The same as in Fig. 3.1. Approximated function is irrational, $f(x) = \frac{x^3}{(x-0.5)*(x+0.5)}$.

it is easy since first layer is Dense, which without any problem learns addition inside the denominator and forwards it to the last layer. The rest operations are proceeded and handled well by the NPU layer.

As for the Dense layer, it did shows some results within the data points grid, showing again that it can memorize training data well. But as soon as it gets out of it, results dropped since model with only Dense layers inside can not extrapolate much.

3.2 Simple function equation discovery

This experiment will cover how regularised NALM models will solve interpretability problems. From the previous experiments, it is clear that models can approximate many kinds of arithmetical functions. But the interesting feature that some of them have is transparency. By only looking into regularised network weights, it is easily seen what function it was trying to approximate. Hence, manual equation recovery is possible.

As an example, the small NMU/NAU model will be explored, with an architecture NAU(1,2),NMU(2,2),NAU(2,1). The model layers were chosen specifically for this task since they are very simple and can be interpreted intuitively, and regularisation gives model with weights clamped to -1,1 or 0 values.

The training data will be a fixed grid in the range [-1.5,1.5] with a step 0.067. The regularisation method will be of an L1 type, which was previously mentioned in the Regularisation section of the Theory chapter. The equation discovery procedure begins after a small regularised model successfully converged to the approximated function f. Exploration starts from observing regularised weights of every layer separately.

Observing weights values of the layer, heatmap plot Fig. A.7, show that the first NAU layer has converged its weights numbers to -1 and 1. Hence first layer output will look this way.

$$\begin{bmatrix} 0.99 \\ -0.99 \end{bmatrix} [x] \rightarrow \begin{bmatrix} 0.99x \\ -0.99x \end{bmatrix} \tag{3.2}$$

The second layer is represented by a 2x2 NMU weight matrix, with values 1 and 0. As we know from the Neural Multiplication Unit section in the Theory chapter, the main goal of NMU is to learn and perform multiplication operations. That allows assuming that the model's output will be just the product of the corresponding inputs, which looks as follows.

$$\begin{bmatrix} 0.99 & 0.99 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0.99x \\ -0.99x \end{bmatrix} \to \begin{bmatrix} -0.96x^2 \\ 0 \end{bmatrix} \tag{3.3}$$

As the last layer, we have NAU layer with values -1 and 0 again. So, proceeding with the NMU layer output as the second NAU layer input, we will receive this intuitive equation.

$$\begin{bmatrix} -0.99 & 0 \end{bmatrix} \begin{bmatrix} -0.96x^2 \\ 0 \end{bmatrix} \to \begin{bmatrix} 0.95x^2 \end{bmatrix} \tag{3.4}$$

In the end, a simple

$$f(x) = 0.95x^2, \tag{3.5}$$

component was received, that highly reminds of a parabola function. And indeed it was a simple parabola function

$$f(x) = x^2, \tag{3.6}$$

that this model trained to approximate.

3.3 Polynomial function equation discovery

Lets now look at more complex polynomial function that the same model will try to approximate. For example polynomial of 4-th power. To do so our model would need to increase the number of hidden layers to 4, since NMU model does multiplication in order for this model to learn function with a x^4 component it will need at least 4 hidden layers inside. After the training regularised layers weights looks can be found here Fig. A.8.

The same as in previous experiment the equation discovery procedure will be held. Looking at the regularised weights, from the Fig. A.7, first layer equation will be held this way

$$\begin{bmatrix} 1.7 \\ -0.8 \\ 1.7 \\ -0.8 \end{bmatrix} \begin{bmatrix} x \end{bmatrix} \to \begin{bmatrix} 1.7x \\ -0.8x \\ 1.7x \\ -0.8x \end{bmatrix}, \tag{3.7}$$

where first matrix is NAU weights matrix, $[x]$ is an input vector.

Second step is to forward previously received output to the NMU layer.

$$\begin{bmatrix} 0.99 & 0 & 0.99 & 0 \\ 0 & 0 & 0 & 0 \\ 0.788 & 0.884 & 0.788 & 0.884 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1.7x \\ -0.8x \\ 1.7x \\ -0.8x \end{bmatrix} \to \begin{bmatrix} 2.83x^2 \\ 0 \\ 0.89x^4 \\ 0 \end{bmatrix}, \tag{3.8}$$

where first matrix is NMU weights, second matrix is an input vector.

As always the last NAU layer will output the final result by proceeding the output from the multiplication model.

$$\begin{bmatrix} 0.93 & 0 & -0.86 & 0 \end{bmatrix} \begin{bmatrix} 2.83x^2 \\ 0 \\ 0.89x^4 \\ 0 \end{bmatrix} \to \begin{bmatrix} 2.63x^2 - 0.76x^4 \end{bmatrix}, \tag{3.9}$$

where first matrix is a layer weights, second one is an input vector.

Final discovered equation have this form

$$f(x) = -0.76x^4 + 2.63x^2, \tag{3.10}$$

and initial approximated equation was

$$f(x) = -x^4 + 3x^2. \tag{3.11}$$

The acquired equation is look almost similar to the initial but have constant differences, which is normal.

There were also attempts to perform on more complex function of irrational type. But there were no representative example received with L1 regularisation, that could be presented. Meaning that there were no models that have converged to irrational function. It is still a way to go for presented models in terms of weights transparency.

Even though, from presented sections of equation discovery experiment it is clear that having a small model with regularised weights values almost certainly can bring an intuitive understanding of what the model is doing. Bringing more transparency to the black NN box.

3.4 Sequential task

This experiment is made in order to show if the Neural Arithmetic models can extrapolate well on the sequential data. In order to make a model perform on the sequences, as was mentioned in the Theory chapter Sec. 2.9 of this thesis, we will use Arithmetic Recurrent Neural Networks. Thus we will substitute network weights with a tested NALM's model A.1.

In the experiment Neural Frankenstein's will be compared to Long-Short Term Memory (LSTM) model, which is also a type of recurrent neural network except it posses some advanced memorization architecture. They will be compared in terms of the sequence length and data range extrapolation. Meaning that training will be performed with predefined sequence length and data range boundary and validation will go further to observe the extrapolation. Experiment settings can be observed hear A.3. The same as in the previous experiment training settings are equal for every model.

The experiment will try to approximate sequences of randomly chosen operations, now it is only addition and multiplication

$$((a + b) * c) - d \tag{3.12}$$

In order to make this experiment happen first data and operations going to be encoded in order as shown bellow:

$$\begin{array}{ccccccc} a & + & b & * & c & + & -d \\ a & 0 & b & 0 & c & 0 & -d \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}, \tag{3.13}$$

where every number n in sequence is encoded as $[n, 0, 0]$, addition and multiplication are encoded as $[0, 1, 0]$, $[0, 0, 1]$ respectfully.

Than there some minor prepossessing done where first data, number in a sequence, is always stuck to the addition operation, which also allows to merge value with the operation that stands before it. It is done for making model learn operation and value rather than making it also learn when the input is an operation or a number.

$$\begin{array}{cccc} (+, a) & (+, b) & (*, c) & (+, -d) \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ a & b & c & -d \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array} \tag{3.14}$$

The models have inside the recurrent part simple linear layer, Operational Network Tab. A.1, which will fetch the encoded numbers and operations at the same time and will approximate the merged version of those two. Next the output from previous step will be fed to the main arithmetic networks Tab. A.1, which will give the product of sequential part of the model to the last NAU layer to sum and output the result. The full models architecture can be found here Tab. A.2. Plot on Fig. 3.7 shows the loss at every range

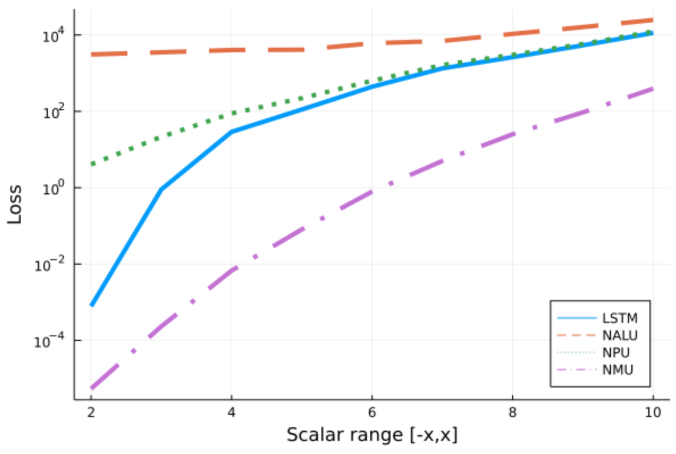


Figure 3.7: Extrapolation comparison on the data range, between Arithmetic Recurrent Neural Network and LSTM.

from [-2,2] to [-10,10], with a step 1, for every model that were used in this experiment. It is clearly seen here, which model performs best. Surprisingly, but created NMU RNN model outperforms state of the art LSTM in in extrapolation. Plot on Fig. 3.8 represents

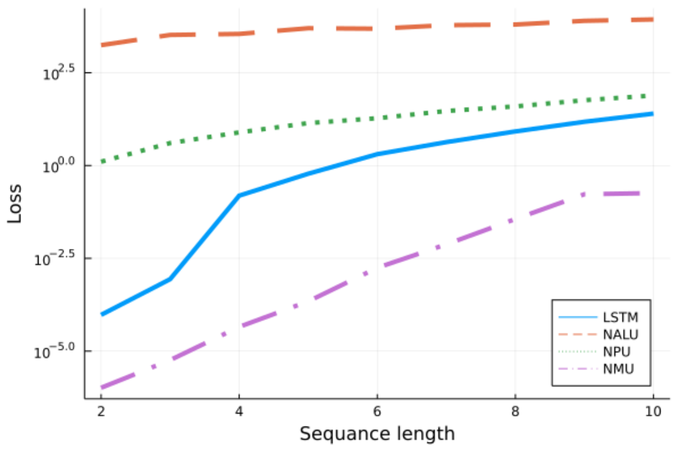


Figure 3.8: Extrapolation comparison of the sequence length, between Arithmetic Recurrent Neural Network and LSTM.

showcase of how models extrapolate in terms of sequence length. And again the careful observation shows that simple model RNN model in conjunction with NMU outperforms in this experiment LSTM, which again highlights how good and useful arithmetic neural networks are.

Chapter 4

Conclusion

In conclusion, the first experiment shows that simple arithmetic tasks can be extrapolated with NMU and NPU models. However, presented models are still experiencing convergence issues with harder tasks that include more complicated structure and arithmetic operations, like division, etc.

The following two function discovery experiments were there as proof of equation discovery concept. In order to make it work in practice, other regularisation methods might be more preferable as well as better training strategies.

The last experiment, sequential task, shows that NMU can be injected in a larger network and demonstrate good results. On the other hand, the rest of the tested models, NALU and NPU, do not improve or even reduce overall model performance. That happens because both of mentioned models have convergence issues.

It is for sure that the extrapolation problem can be resolved using tested models. But the field still has to come up with something better to solve the Neural Arithmetics fully. Besides using arithmetic models in the arithmetic field, they can still be applicable to any task composed of simple arithmetical operations, such as financial or engineering modeling.

There is much stuff that should be solved before. Future works will focus on faster convergence and extrapolation to other functions like periodical, exponential, logarithmic, and irregular. In the future, those types of models will succeed in representing exact ordinary differential equations (ODE).

Bibliography

[1] Miguel Alvarez et al. *Reducing Road Vehicle Fuel Consumption by Exploiting Connectivity and Automation: A Literature Survey.* 2020. arXiv: `2011.14805 [eess.SY]`.

[2] Omkar Bhalerao. *Universal Approximation Theorem.* `https://medium.com/swlh/universal-approximation-theorem-d1a1a67c1b5b`. July 2020.

[3] VIOLET R. CANE. *MATHEMATICAL MODELS FOR NEURAL NETWORKS.* 2018. URL: `https://digitalassets.lib.berkeley.edu/math/ucb/text/math_s5_v4_article-04.pdf`.

[4] Geoffrey E. Hinton Ronald J. Williams David E. Rumelhart. "Learning representations by back-propagating errors". In: *Nature* (1986).

[5] Fisseha Berhane. *Optimization Methods.* [Online; accessed May 17, 2020]. 2013. URL: `https://datascience-enthusiast.com/DL/Optimization_methods.html`.

[6] Niklas Heim, Tomáš Pevný, and Václav Šmídl. *Neural Power Units.* 2020. arXiv: `2006.01681 [cs.LG]`.

[7] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[8] Anna DePold Hohler and Marcus Ponce de Leon. "Movement Disorders". In: *Encyclopedia of Clinical Neuropsychology.* Ed. by Jeffrey Kreutzer, John DeLuca, and Bruce Caplan. Cham: Springer International Publishing, 2017, pp. 1–1. ISBN: 978-3-319-56782-2. DOI: `10.1007/978-3-319-56782-2_468-2`. URL: `https://doi.org/10.1007/978-3-319-56782-2_468-2`.

[9] Zachary Chase Lipton. "The Mythos of Model Interpretability". In: *CoRR* abs/1606.03490 (2016). arXiv: `1606.03490`. URL: `http://arxiv.org/abs/1606.03490`.

[10] Andreas Madsen and Alexander Rosenberg Johansen. *Neural Arithmetic Units.* 2020. arXiv: `2001.05016 [cs.NE]`.

[11] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015. URL: `http://neuralnetworksanddeeplearning.com`.

[12] Christopher Olah. *Understanding LSTM Networks.* 2015. URL: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[13] Janelle Shane. *Do neural nets dream of electric sheep?* `https://aiweirdness.com/post/171451900302/do-neural-nets-dream-of-electric-sheep`. Mar. 2018.

[14] Niek Tax et al. "Predictive Business Process Monitoring with LSTM Neural Networks". In: *CAiSE.* 2017.

[15] Amirsina Torfi et al. *Natural Language Processing Advancements By Deep Learning: A Survey.* 2020. arXiv: `2003.01200 [cs.CL]`.

[16] Andrew Trask et al. "Neural Arithmetic Logic Units". In: *CoRR* abs/1808.00508 (2018). arXiv: `1808.00508`. URL: `http://arxiv.org/abs/1808.00508`.

Appendix A

Appendix title

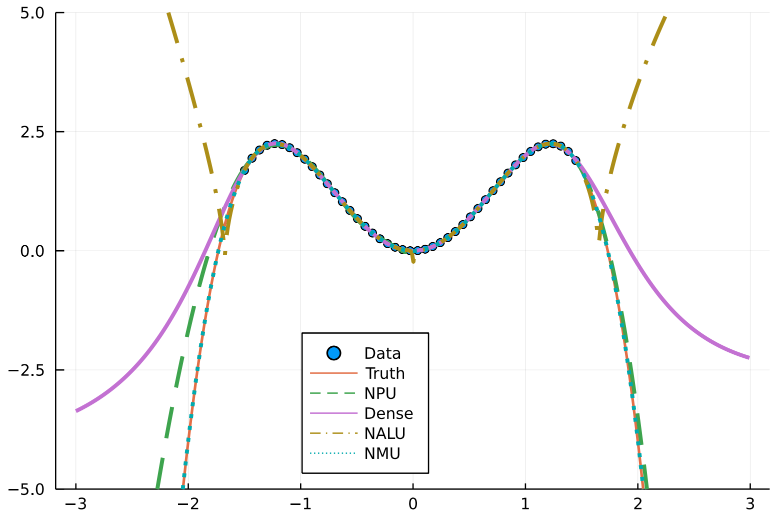


Figure A.1: Experiment with polynomial function.

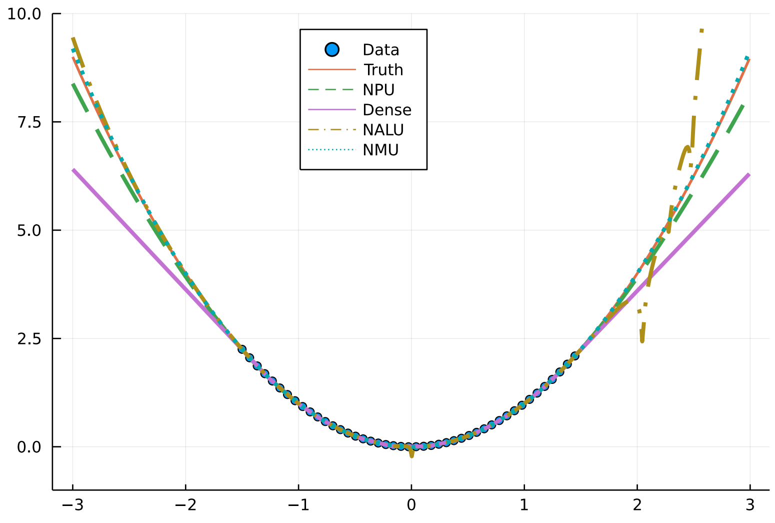


Figure A.2: Experiment with simple parabola function.

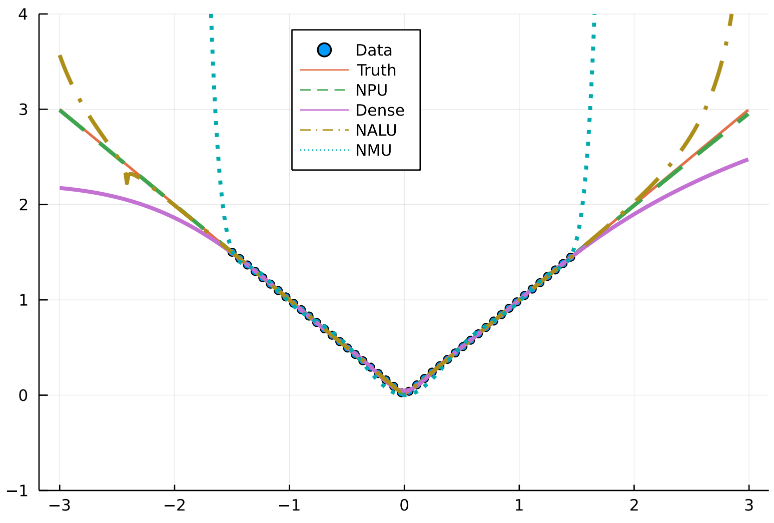


Figure A.3: Experiment with absolute value function.

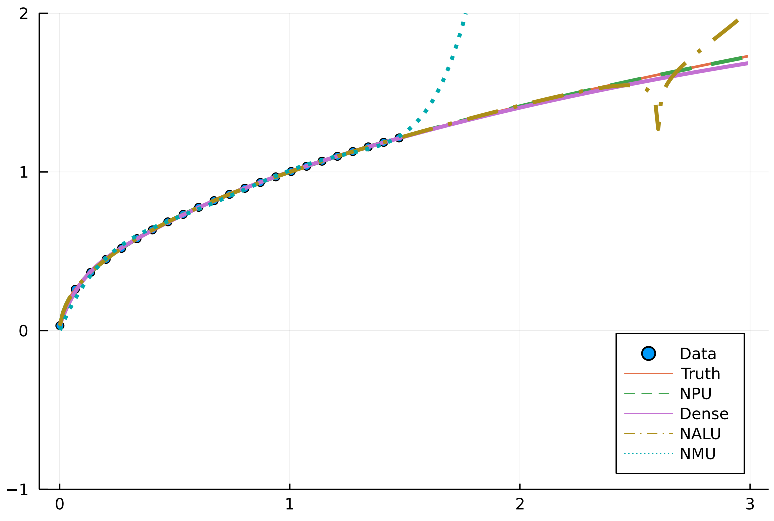


Figure A.4: Experiment with square root function.

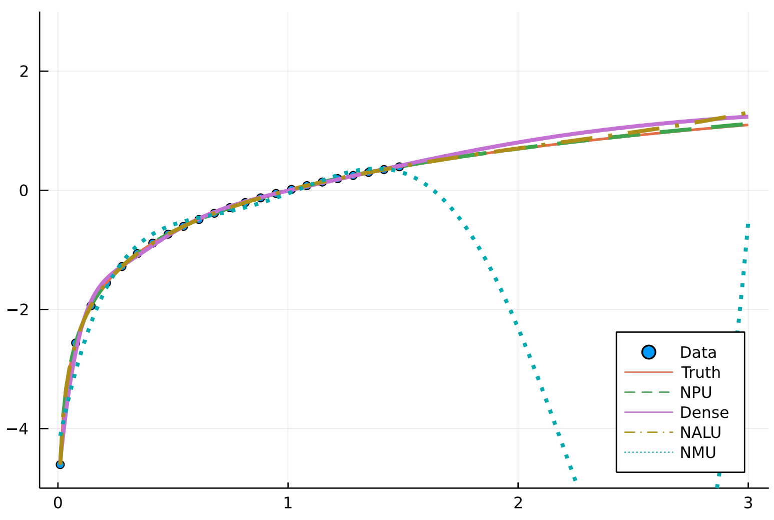


Figure A.5: Experiment with natural logarithm function.

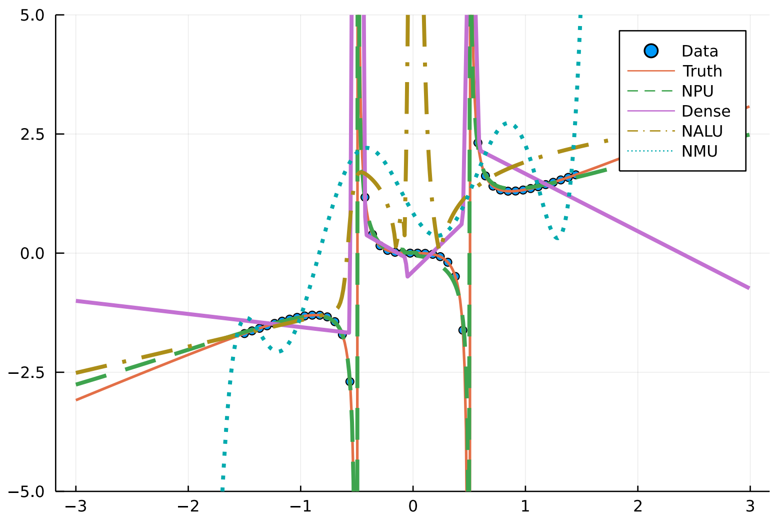


Figure A.6: Experiment with square root function.

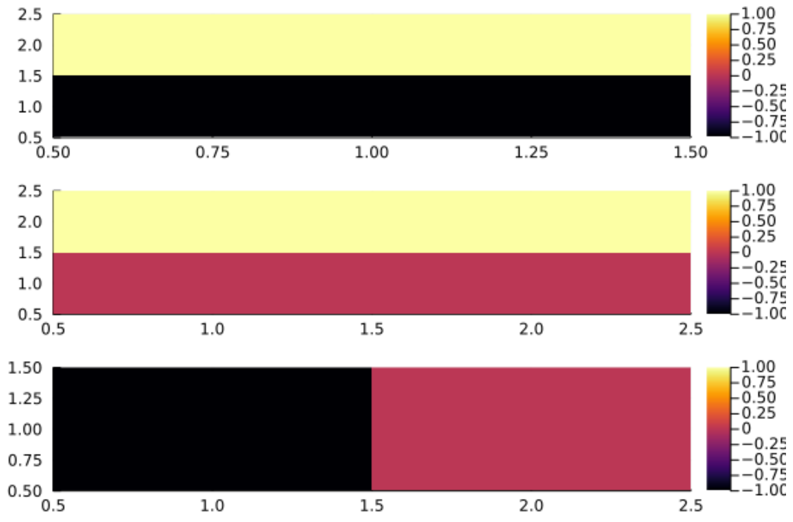


Figure A.7: Heat map of the model weights, where color represents rounded value of the corresponding layer weight.

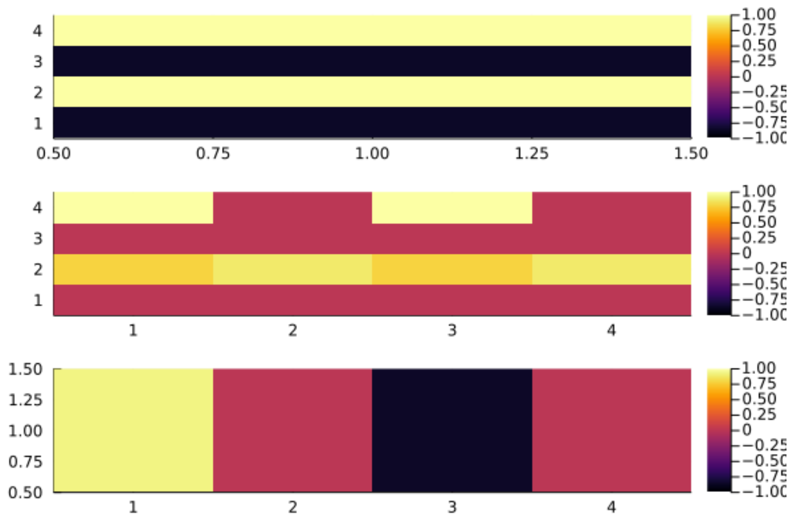


Figure A.8: Heat map of the model weights, where color represents rounded value of the corresponding layer weight.

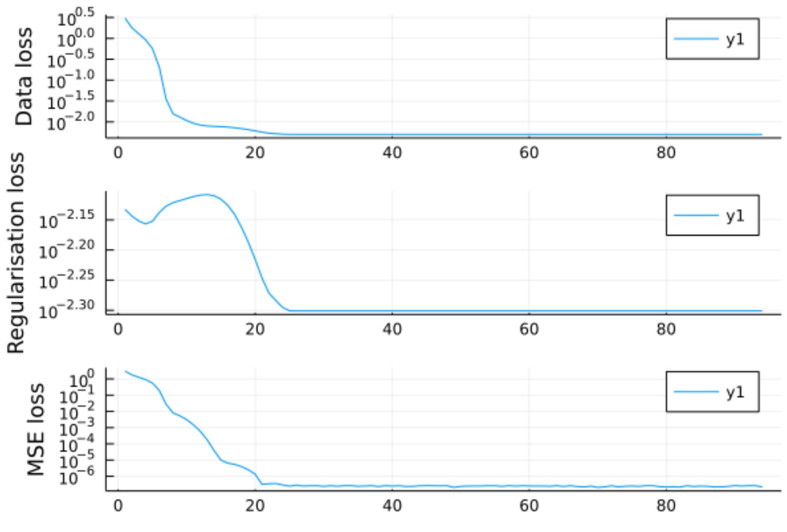


Figure A.9: The loss plots representing the convergence accuracy of the model, that was used for equation recovery of a simple parabola function.

Table A.1: Networks architectures for the "Sequence task" subsection.

Networks	Architecture
Operation	Dense(6,10),Dense(10,3)
NMU	NAU(6,6),NMU(6,3)
NPU	NAU(6,6),NPU(6,3)
NALU	NALU(6,6),NALU(6,3)
NAU	NAU(3,1)

Table A.2: Models architectures for the "Sequence task" subsection, where some networks (Operation, NMU, NPU, NALU) were taken from Tab. A.1.

Models	Architecture
NMU	Operation,NMU,NAU
NPU	Operation,NPU,NAU
NALU	Operation,NALU,NAU
LSTM	LSTM(6,30),LSTM(30,10),Dense(10,3)

Table A.3: Test settings for "Sequence task" section.

	Training	Testing
Maximum sequence length	3	10
Data values ranges	[-2,2]	[-10,10]
Data size	180000	8000