Neural Power Units

Niklas Heim, Tomáš Pevný, Václav Šmídl
Artificial Intelligence Center
Czech Technical University
Prague, CZ 120 00
{niklas.heim, tomas.pevny, vasek.smidl}@aic.fel.cvut.cz

Abstract

Conventional Neural Networks can approximate simple arithmetic operations, but fail to generalize beyond the range of numbers that were seen during training. Neural Arithmetic Units aim to overcome this difficulty, but current arithmetic units are either limited to operate on positive numbers or can only represent a subset of arithmetic operations. We introduce the Neural Power Unit (NPU)\(^1\) that operates on the full domain of real numbers \(\mathbb{R}\) and is capable of learning arbitrary power functions in a single layer. The NPU thus fixes the shortcomings of existing arithmetic units and extends their expressivity. We achieve this by using complex arithmetic without requiring a conversion of the network to complex numbers \(\mathbb{C}\). A simplification of the unit to the RealNPU yields a highly transparent model. We show that the NPUs outperform their competitors in terms of accuracy and sparsity on artificial arithmetic datasets, and that the RealNPU can discover the governing equations of a dynamical system only from data.

1 Introduction

Numbers and simple algebra are essential not only to human intelligence but also to the survival of many other species [Dehaene, 2011; Gallistel, 2018]. A successful, intelligent agent should, therefore, be able to perform simple arithmetic. State of the art neural networks are capable of learning arithmetic, but they fail to extrapolate beyond the ranges seen during training [Suzgun et al., 2018; Lake and Baroni, 2018]. The inability to generalize to unseen inputs is a fundamental problem that hints at a lack of understanding of the given task. The model merely memorizes the seen inputs and fails to abstract the true learning task. The failure of numerical extrapolation on simple arithmetic tasks has been shown by [Trask et al., 2018], who also introduced a new class of Neural Arithmetic Units with good extrapolation performance on some arithmetic tasks.

Including Neural Arithmetic Units in standard neural networks promises to significantly increase their extrapolation capabilities due to their inductive bias towards numerical computation. This is especially important for tasks in which the data generating process contains mathematical relationships. They also promise to reduce the number of parameters needed for a given task, which can improve the explainability of the model. We demonstrate this in a Neural Ordinary Differential Equation (NODE, Chen et al. [2019]), where a handful of neural arithmetic units can outperform a much bigger network built from dense layers (Sec. 4.1). Moreover, our new unit can be used to directly read out the correct generating ODE from the fitted model. This is in line with recent efforts to build transparent models instead of attempting to explain black-box models [Rudin, 2019], like conventional neural networks. We refer to the terminology by [Lipton, 2017] which defines the potential of understanding the parameters of a given model as transparency by decomposability.

\(^1\)Implementation of Neural Arithmetic Units: \url{github.com/nmheim/NeuralArithmetic.jl} The code to reproduce our experiments is available at \url{github.com/nmheim/NeuralPowerUnits}

The currently available arithmetic units all have different strengths and weaknesses, but none of them solve simple arithmetic completely. The Neural Arithmetic Logic Unit (NALU) by Trask et al. [2018], chronologically, was the first arithmetic unit. It can solve addition (+, including subtraction), multiplication (×), and division (÷), but is limited to positive inputs. The convergence of the NALU is quite fragile due to an internal gating mechanism between addition and multiplication paths as well as the use of a logarithm which is problematic for small inputs. Recently, Schlör et al. [2020] introduced the improved NALU (iNALU, to fix the NALU’s shortcomings. It significantly increases its complexity, and we observe only a slight improvement in performance. Madsen and Johansen [2020] solve (+, ×) with two new units: the Neural Addition Unit (NAU), and the Neural Multiplication Unit (NMU). Instead of gating between addition and multiplication paths, they are separate units that can be stacked. They can work with the full range of real numbers, converge much more reliably, but cannot represent division.

Our Contributions

Neural Power Unit. We introduce a new arithmetic layer (NPU, Sec. 3) which is capable of learning products of power functions (∏x^w_i) of arbitrary real inputs x_i and power w_i, thus including multiplication (x_1 × x_2 = x_1^1 x_2^1) as well as division (x_1 ÷ x_2 = x_1^1 x_2^{-1}). This is achieved by using formulas from complex arithmetic (Sec. 3.1). Stacks of NAUs and NPUs can thus learn the full spectrum of simple arithmetic operations.

Convergence improvement. We address the known convergence issues of neural arithmetic units by introducing a relevance gate that smooths out the loss surface of the NPU (Sec. 3.2). With the relevance gate, which helps to learn to ignore variables, the NPU reaches extrapolation errors and sparsities that are on par with the NMU on (×) and outperforms NALU on (÷, √).

Transparency. We show how a power unit can be used as a highly transparent model for equation discovery of dynamical systems. Specifically, we demonstrate its ability to identify a model that can be interpreted as a SIR model with fractional powers (Sec. 4.1) that was used to fit the COVID-19 outbreak in various countries [Taghvaei et al., 2020].

2 Related Work

Several different approaches to automatically solve arithmetic tasks have been studied in recent years. Approaches include Neural GPUs [Kaiser and Sutskever, 2016], Grid LSTMs [Kalchbrenner et al., 2016], Neural Turing Machines [Graves et al., 2014], and Neural Random Access Machines [Kurach et al., 2016]. They solve tasks like binary addition and multiplication, or single-digit arithmetic. The Neural Status Register [Faber and Wattenhofer, 2020] focuses on control flow. The Neural Arithmetic Expression Calculator [Chen et al., 2018], a hierarchical reinforcement learner, is the only method that solves the division problem, but it operates on character sequences of arithmetic expressions. Related is symbolic integration with transformers [Lample and Charton, 2019]. Unfortunately, most of the named models have severe problems with extrapolation [Madsen and Johansen, 2019, Saxton et al., 2019]. A solution to the extrapolation problem could be Neural Arithmetic Units. They are designed with an inductive bias towards systematic, arithmetic computation. However, currently, they are limited in their capability of expressing the full range of simple arithmetic operations (+, ×, ÷). In the following two sections, we briefly describe the currently available arithmetic layers, including their advantages and drawbacks.

2.1 Neural Arithmetic Logic Units

Trask et al. [2018] have demonstrated the severity of the extrapolation problem of dense networks for even the simplest arithmetic operations, such as summing or multiplying two numbers. To increase the power of abstraction for arithmetic tasks, they propose the Neural Arithmetic Logic Unit (NALU), which is capable of learning (+, ×, ÷). However, the NALU cannot handle negative inputs correctly due to the logarithm in Eq. 2.

---

2 as defined by Lipton [2017]
**Definition (NALU).** The NALU consists of a (+) and a (×) path with shared weights \( W \) and \( M \).

**Addition:**
\[
\hat{W}x = \text{tanh}(W) \odot \sigma(M)
\]

**Multiplication:**
\[
m = \exp(\hat{W} \log(|x| + \epsilon))
\]

**Output:**
\[
y = a \odot g + m \odot (1 - g), \quad g = \sigma(Gx)
\]

with inputs \( x \) and learnt parameters \( W, M, \) and \( G \).

Additionally, the logarithm destabilizes training to the extent that the chance of success can drop below 20\% for (+, ×), it becomes practically impossible to learn (÷) and difficult to learn from small inputs in general [Madsen and Johansen 2019]. Schlör et al. [2020] provide a detailed description of the shortcomings of the NALU, and they suggest an **improved NALU** (iNALU). The iNALU addresses the NALU’s problems through several mechanisms. It has independent addition and multiplication weights for Eq. 1 and Eq. 2, clips weights and gradients to improve training stability, regularizes the weights to push them away from zero, and, most importantly, introduces a mechanism to recover the sign that is lost due to the absolute value in the logarithm. Additionally, the authors propose to reinitialize the network if its loss is not improving during training. We include the iNALU in one of our experiments and find that it only slightly improves the NALU’s performance (Sec. 4.2) at the cost of a significantly more complicated unit. Our NPU avoids all these mechanisms by internally using complex arithmetic.

### 2.2 Neural Multiplication Unit & Neural Addition Unit

Instead of trying to fix the NALU’s convergence issues, Madsen and Johansen [2020] propose a new unit for (×) only. The **Neural Multiplication Unit** (NMU) uses explicit multiplications and learns to gate between identity and (×) of inputs. The NMU is defined by Eq. 4 and is typically used in conjunction with the so-called **Neural Addition Unit** (NAU) in Eq. 5.

**Definition (NMU & NAU).** NMU and NAU are two units that can be stacked to model (+, ×).

**NMU:**
\[
y_j = \prod_i \hat{M}_{ij}x_i + 1 - \hat{M}_{ij}
\]

**NAU:**
\[
y = \hat{A}x
\]

with inputs \( x \), and learnt parameters \( M \) and \( A \).

Both NMU and NAU are regularized with \( R = \sum_{ij} \min(|W_{ij}|, |1 - W_{ij}|) \), and their weights are clipped, which biases them towards learning an operation or pruning it completely. The combination of NAU and NMU can thus learn (+, ×) for both positive and negative inputs. Training NAU and NMU is stable and succeeds much more frequently than with the NALU, but they cannot represent (÷), which we address with our NPU.

### 3 Neural Power Units

To fix the deficiencies of current arithmetic units, we propose a new arithmetic unit (inspired by NALU) that can learn arbitrary products of power functions (\( \prod x_i^w \)) for positive and negative numbers, and still train well. Combined with the NAU, we solve the full range of arithmetic operations. This is possible through a simple modification of the (×)-path of the NALU (Eq. 6). We suggest to replace the logarithm of the absolute value by the complex logarithm and to allow \( W \) to be complex as well. Since the complex logarithm is defined for negative inputs, the NPU does not have a problem with negative numbers. A complex \( W \) improves convergence at the expense of transparency (see Sec. 4.1). The improvement during training might be explained by the additional imaginary parameters that make it possible to avoid regions with an uninformative gradient signal.

#### 3.1 Naive Neural Power Unit – NaiveNPU

With the modifications introduced above we can extend the multiplication path of the NALU from

\[
m = \exp(W \log_{\text{real}}(|x| + \epsilon))
\]
to use the complex logarithm (log := log\(_{\text{complex}}\)) and a complex weight \(W\) to
\[
y = \exp(W \log x) = \exp ((W_r + iW_i) \log x),
\]
where the input \(x\) is still a vector of real numbers. With the polar form for a complex number 
\(z = re^{i\theta}\) the complex log applied to a real number \(x = re^{i\pi}\) is
\[
\log x = \log r + i\pi k,
\]
where \(k = 0\) if \(x \geq 0\) and \(k = 1\) if \(x < 0\). Using the complex log in Eq. 7 lifts the positivity 
constraint on \(x\), resulting in a layer that can process both positive and negative numbers correctly. A 
complex weight matrix \(W\) somewhere in a larger network would result in complex gradients in other 
layers. This would effectively result in doubling the number of parameters of the whole network. As 
we are only interested in real outputs, we can avoid this doubling by considering only the real part of 
the output \(y\):
\[
\text{Re}(y) = \text{Re}(\exp((W_r + iW_i)(\log r + i\pi k))) \\
= \exp(W_r \log r - \pi W_i, k) \odot \cos(W_i \log r + \pi W_r, k).
\]

Above we have used Euler’s formula \(e^{ix} = \cos x + i \sin x\). A diagram of the NaiveNPU is shown in 
Fig. 1.

**Definition (NaiveNPU).** The Naive Neural Power Unit, with matrices \(W_r\) and \(W_i\) representing real 
and imaginary part of the complex numbers, is defined as
\[
y = \exp(W_r \log r - \pi W_i k) \odot \cos(W_i \log r + \pi W_r k), \\
where \r = |x| + \epsilon, \ k_i = \begin{cases} 
0 & x_i \geq 0 \\
1 & x_i < 0,
\end{cases}
\]
with inputs \(x\), machine epsilon \(\epsilon\), and learnt parameters \(W_r\) and \(W_i\).

### 3.2 The Relevance Gate – NPU

The NaiveNPU has difficulties to converge on large scale tasks, and to reach sparse results in cases 
where the input to a given row is small. We demonstrate this on a toy example of learning the function 
\(f : \mathbb{R}^2 \to \mathbb{R}\), which is the identity on one of two inputs. The task is defined by the loss \(\mathcal{L}\):
\[
\mathcal{L} = \sum_i |m(x_1, x_2) - f(x_1, x_2)| = \sum_i |m(x_1, x_2) - x_1, i|,
\]
where \(m = \text{NaiveNPU} \text{ with } (W_r, W_i) \in \mathbb{R}^{1 \times 2}\)
and \(x_1 \sim \mathcal{U}(0, 2), \ x_2 \sim \mathcal{U}(0, 0.05)\)

The left plot in Fig. 3 depicts the gradient norm \(\mathcal{G}\)
\[
\mathcal{G}(W_r) = \left\| \frac{\partial \mathcal{L}}{\partial W_r} \right\|_2
\]
of the NaiveNPU for a batch of two-dimensional inputs. Even in this simple example, the gradient 
of the NaiveNPU is close to zero in large parts of the parameter space. This can be explained as 
follows. One row of NaiveNPU weights effectively raises each input to a power and multiplies them: 
\(x_1^{|w_1|}x_2^{|w_2|} \ldots x_n^{|w_n|}\). If a single input \(x_i\) is constantly close to zero (i.e. irrelevant), the whole row will 
be zero, no matter what its weights are and the gradient information on all other weights is lost. 
Therefore, we introduce a gate on the input of our layer that can turn irrelevant inputs into 1s. A 
diagram of the NPU is shown in Fig. 4.
4 Experiments

(UDEs, Rackauckas et al. [2020]) were introduced which aim to combine Differential Equations with data-driven models with physically informed differential equations to maximize interpretability/explainability of the resulting models.

If an ODE model is composed of dense layers, its direct interpretation is problematic and has to be performed retrospectively. The class of models based on SINDy is transparent by design, however it can only provide explanation within a linear combination of predefined set of basis functions. Thus, it cannot learn models with unknown fractional powers. With this experiment we aim to show that

Definition (NPU). The NPU extends the NaiveNPU by the relevance gate $g$ on the input $x$.

$$ y = \exp(W_r \log r - \pi W_i k) \odot \cos(W_i \log r + \pi W_r k), \quad \text{where} $$

$$ r = g \odot (|x| + \epsilon) + (1 - g), \quad k_i = \begin{cases} 0 & x_i \geq 0 \\ g_i & x_i < 0 \end{cases}, \quad g_i = \min(\max(g_i, 0), 1), $$

with inputs $x$, and learnt parameters $W_r, W_i$ and $g$.

The central plot of Fig. 3 shows the gradient norm $G$ of the NPU on the identity task with its initial gate setting of $g_1 = g_2 = 0.5$. The large zero-gradient region of the NaiveNPU is gone. The last plot shows the same for $g_1 = 1$ and $g_2 = 0$, which corresponds to the correct gates at the end of NPU training. The gradient is independent of $w_2$, which means that it can easily be pruned by a simple regularization such as $L_1$. In Sec. 4.3 we show how important the relevance gating mechanism is for the convergence and sparsity of large models. Sparsity is especially important in order to use the NPU as a transparent model.

Initialization We recommend initializing the NPU with a Glorot Uniform distribution on the real weights $W_r$. The imaginary weights $W_i$ can be initialized to zeros, so they will only be used where necessary, and the gate $g$ with 0.5, so the NPU can choose to output 1.

Definition (RealNPU). In many practical tasks, such as multiplication or division, the final value of $W_i$ should be equal to zero. We will denote NPU with removed parameters for the imaginary part as RealNPU and study the impact of this change on convergence in Sec. 4.

4 Experiments

In Sec. 4.3 we show how the NPU can help to build better NODE models. Additionally, we use the RealNPU as a highly transparent model, from which we can directly recover the generating equation of an ODE containing fractional powers. Subsequent Secs. 4.2 and 4.3 compare the NPU to prior art (NALU and NMU) on arithmetic tasks typically used to benchmark arithmetic units.

4.1 A Step Towards Equation Discovery of an Epidemiological Model

Data-driven models such as SINDy [Champion et al., 2019] or Neural Ordinary Differential Equations (NODE, Chen et al., 2019) are used more and more in scientific applications. Recently, Universal Differential Equations (UDEs, Rackauckas et al., 2020) were introduced which aim to combine data-driven models with physically informed differential equations to maximize interpretability/explainability of the resulting models.
We train each model for 3000 steps with the ADAM optimizer and a learning rate of 0.005, and we have numerically simulated one realization of the fSIR model with the parameters which is typically taken to be proportional to the product of sensitive area much more post-processing is needed to ensure safe predictions. The third, and the last hidden variable were found to be irrelevant (the relevance gate returns 1). The NPU can potentially discover exact ODE models. An example of an ODE that contains powers is a modification of the well-known epidemiological SIR model [Kermack et al., 1927] to fractional powers (fSIR, Taghvaei et al. [2020]), which was shown to be a beneficial modification for modelling COVID-19 outbreak. The SIR model is built from three variables: S (susceptible), I (infectious), and R (recovered/removed). Arguably the most important part of the model is the transmission rate $r$, which is typically taken to be proportional to the product of S and I. Taghvaei et al. [2020] argue that, especially in the initial phase of an epidemic, the boundary areas of infected and susceptible cells scale with a fractional power, which leads to Eq. 17:

$$\frac{dS}{dt} = -r(t) + \eta R(t), \quad \frac{dI}{dt} = r(t) - \alpha I(t), \quad \frac{dR}{dt} = \alpha I(t) - \eta R(t),$$

(16)

$$r(t) = \beta I(t)\gamma S(t)^\kappa,$$

(17)

We have numerically simulated one realization of the fSIR model with the parameters $\alpha = 0.05$, $\beta = 0.06$, $\eta = 0.01$, $\gamma = \kappa = 0.5$, in 40 time steps that are equally spaced in the time interval $T = (0, 200)$, such that the training data $X = [S_t, I_t, R_t]^T_{t=1}$ contains one time series each for $S, I$, and $R$. The initial conditions $u_0 = [S_0, I_0, R_0]$ are set to $S_0 = 100, I_0 = 0.01$, and $R_0 = 0$, as shown in Figure 4 (right). We fit the data with three different NODEs composed of different model types: a dense network, the NPU, and the RealNPU. An exemplary model is: NPU = Chain(NPU(3, h), NAU(h, 3)) with variable hidden size $h$. The detailed models are defined in Tab. A1. The training objective is the loss $L$ with $L_1$ regularization.

$$L = \text{MSE}(X, \text{NODE}_d(u_0)) + \beta ||\theta||_1.\tag{18}$$

We train each model for 3000 steps with the ADAM optimizer and a learning rate of 0.005, and subsequently with LBFGS until convergence (or for maximum 1000 steps). For each model type, we run a small grid search to build a Pareto front with $h \in \{6, 9, 12, 15, 20\}$ and $\beta \in \{0, 0.01, 0.1, 1\}$, where each hyper-parameter pair is run five times. The resulting Pareto front is shown on the left of Fig. 4. The NPU reaches much sparser and better solutions than the dense network. The RealNPU has problems to converge in the majority of cases, however, there are a few models in the bottom left that reach a very low MSE and have very few parameters. The best of these models is shown in Fig. 5. It looks strikingly similar to the fSIR model in matrix form:

$$\begin{align*}
\begin{bmatrix}
\dot{S} \\
\dot{i} \\
\dot{R}
\end{bmatrix} & = 
\begin{bmatrix}
-\beta & 0 & \eta \\
\beta & -\alpha & 0 \\
0 & \alpha & \eta 
\end{bmatrix}
\begin{bmatrix}
I \gamma S^\kappa \\
I \\
R
\end{bmatrix}.
\end{align*}\tag{19}$$

Reading Fig. 5 from right to left, we can extract the ODE that the RealNPU represents. The first hidden variable correctly identified the transmission rate as a product of two fractional powers $r = I \gamma S^\kappa$ with $\kappa = 0.57$ and $\gamma = 0.62$, which is close to the true values $\gamma = \kappa = 0.5$. The second, third, and the last hidden variable were found to be irrelevant (the relevance gate returns 1). The demonstration given here is not intended to be used in practice. For real-world predictions in such a sensitive area much more post-processing is needed to ensure safe predictions.
Figure 5: Visualization of the best RealNPU. Reading from right to left, it takes the SIR variables as an input, then applies the NPU and the NAU. It correctly identifies $r$ as a fractional product in the NPU, and gets the rest of the fSIR parameters almost right in the NAU.

fourth hidden variable is a selector of the second input $I$, and the fifth hidden variable is selector of a power of $R$, $R^{0.64}$. In the second layer, the NAU combines the correct hidden outputs from the NPU such that $\dot{S}$ is composed of the negative transmission rate $r$ and positive $R$. $\dot{I}$ and $\dot{R}$ are also composed of the correct hidden variables, with the parameters $\alpha, \beta, \eta$ being not far off from the truth. We conclude that even with this very naive approach, the RealNPU can recover something close to the true fractional SIR model.

In summary, the NPU can work well in sequential tasks, and we have shown that we can reach highly transparent results with the RealNPU, but in practice, using the RealNPU might be difficult due to its lower success rate. With a more elaborate analysis, it should be possible to reach the same solutions with the full NPU and e.g. a strong regularization of its imaginary parameters.

4.2 Simple Arithmetic Task

In this experiment we compare six different layers (NPU, RealNPU, NMU, NALU, iNALU, Dense) on a small problem with two inputs and four outputs. The objective is to learn the function $f : \mathbb{R}^2 \rightarrow \mathbb{R}^4$ with a standard MSE loss:

$$f(x, y) = (x + y, xy, x/y, \sqrt{x})^T =: t,$$

$$\mathcal{L} = \frac{1}{4} \sum_{i=1}^4 (\text{model}(x, y)_i - f(x, y)_i) = \text{MSE}(\hat{t}, t).$$

Learning the function $f$ includes not only learning the correct arithmetic operation, but also to separate them cleanly, which tests the gating mechanisms of the layers. Each model has two layers with a hidden dimension $h$. E.g. the NPU model is defined by $\text{NPU} = \text{Chain}(\text{NPU}(2, h = 6), \text{NAU}(h = 6, 4))$. The remaining models that are used in the tables and plots are given in Tab. A3. To obtain valid results in case of division we train on positive, non-zero inputs, but test on negative, non-zero numbers (except for test inputs to the square-root):

$$(x_{\text{train}}, y_{\text{train}}) \sim \mathcal{U}(0.1, 2) \quad (x_{\text{test}}, y_{\text{test}}) \sim \mathcal{R}(-4.1:0.2:4) \quad (x_{\text{test,sqrt}}, y_{\text{test,sqrt}}) \sim \mathcal{R}(0.1:0.1:4)$$

where $\mathcal{R}$ denotes a range with start, step, and end. We train each model for 20 000 steps with the ADAM optimizer, a learning rate of 0.001, and a batch size of 100. The input samples are generated on the fly during training. Fig. 6 shows the error surface of the best of 20 models on each task. Tab. A2 lists the corresponding averaged testing errors of all 20 models.

Both NPUs successfully learn $(+, \times, \div, \sqrt{\cdot})$ and clearly outperform NALU and iNALU on all tasks. Surprisingly, the NALU has problems extrapolating in this task, which as Schlör et al. [2020] suggest, might be due to its gating mechanism. The NPUs are on par with the NMU for $(+)$, but the NMU is better at $(\times)$ due to its inductive bias. The NMU cannot learn $(\div, \sqrt{\cdot})$. The fact that the RealNPU performs slightly better than the NPU indicates that the task is easy enough to not require the
imaginary parameters to help convergence. In such a case, the RealNPU generalizes better because it corresponds to the task it is trying to learn.

## 4.3 Large Scale Arithmetic Task

One of the most important properties of a layer in a neural network is its ability to scale. With the large scale arithmetic task we show that the NPU works reliably on many-input tasks that are heavily over-parametrized. In this section we compare NALU, NMU, NPU, RealNPU, and the NaiveNPU on a task that is identical to the ‘arithmetic task’ that Madsen and Johansen [2020] and Trask et al. [2018] analyse as well. The goal is to sum two subsets of a 100 dimensional vector and apply an operation (like \(\times\)) to the two summed subsets. The dataset generation is defined in the set of Eq. 24 with the parameters from Tab. A5:

\[
\begin{align*}
    a &= \sum_{i=s_{1\text{, start}}}^{s_{1\text{, end}}} x_i, \\
    b &= \sum_{i=s_{2\text{, start}}}^{s_{2\text{, end}}} x_i, \\
    y_{\text{add}} &= a + b, \\
    y_{\text{mul}} &= a \times b, \\
    y_{\text{div}} &= 1/a, \\
    y_{\sqrt{\text{a}}} &= \sqrt{a},
\end{align*}
\]  

where starting and ending values \(s_{1\text{, start}}, s_{1\text{, end}}\) of the summations are chosen such that \(a\) and \(b\) come from subsets of the input vector \(x\) with a given overlap. The training objective is standard MSE, regularized with \(L_1\):

\[
L = \text{MSE}(\text{model}(x), y) + \beta \|\theta\|_1,
\]  

where \(\beta\) is scheduled to be low in the beginning of training and stronger towards the end. Specifics of the used models and their hyper-parameters are defined in Tab. A4 & A6. Madsen and Johansen [2020] perform an extensive analysis of this task with different subset and overlap ratios, varying model and input sizes, and much more, establishing that the combination of NAU/NMU outperforms the NALU. We focus on the comparison of NPU, RealNPU, NMU, and NALU on the default parameters of Madsen and Johansen [2020] which sets the subset ratio to 0.5 and the overlap ratio to 0.25 (details in Tab. A5). We include the NaiveNPU (without the relevance gate) to show how important the gating mechanism is for both sparsity and overall performance.

Fig. 7 plots testing errors over the number of non-zero parameters for all models and tasks. The addition plot shows that NMU, NPU, and RealNPU successfully learn and extrapolate on (+) with
Table 1: Testing errors of the large scale arithmetic task. Each value is obtained by computing median (and median absolute deviation) of 10 runs.

<table>
<thead>
<tr>
<th>Task</th>
<th>NPU</th>
<th>RealNPU</th>
<th>NALU</th>
<th>NMU</th>
<th>NaiveNPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0.092 ± 0.031</td>
<td>0.063 ± 0.014</td>
<td>740.0 ± 330.0</td>
<td><strong>0.00602 ± 0.00019</strong></td>
<td>161.65 ± 0.11</td>
</tr>
<tr>
<td>×</td>
<td>4.28 ± 0.9</td>
<td>3.09 ± 0.74</td>
<td>2.9e83 ± 2.9e83</td>
<td><strong>1.7 ± 1.4</strong></td>
<td>3750.0 ± 870.0</td>
</tr>
<tr>
<td>÷</td>
<td><strong>1.0e-7 ± 1.0e-7</strong></td>
<td>1.4e-6 ± 4.0e-7</td>
<td>530.0 ± 200.0</td>
<td>1.622 ± 0.081</td>
<td>5.4e17 ± 5.4e17</td>
</tr>
<tr>
<td>√</td>
<td>0.054 ± 0.0078</td>
<td><strong>0.017 ± 0.011</strong></td>
<td>7300.0 ± 7200.0</td>
<td>10.96 ± 0.89</td>
<td>9.3e8 ± 9.3e8</td>
</tr>
</tbody>
</table>

5 Conclusion

We introduced the Neural Power Unit which addresses the deficiencies of current arithmetic units: it can learn multiplication, division, and arbitrary power functions for positive, negative, and small numbers. We showed that the NPU outperforms its main competitor (NALU) and reaches performance that is on par with the multiplication specialist NMU (Sec. 4.2 and 4.3). Additionally, we have demonstrated that the NPU converges consistently, even on sequential tasks. The RealNPU can be used as a transparent model that is capable of recovering the governing equations of dynamical systems purely from the data (Sec. 4.1).

Broader Impact

Current neural network architectures are often perceived as black box models that are difficult to explain or interpret. This becomes highly problematic if ML models are involved in high stakes decisions in e.g. criminal justice, healthcare, or control systems. With the NPU, we hope to contribute to the broad topic of interpretable machine learning, with a focus on scientific applications. Additionally, learning to abstract (mathematical) ideas and extrapolating is a fundamental goal that might contribute to more reliable machine learning systems.

However, the inductive biases that are used to increase transparency in the NPU can cause the model to ignore subgroups in the data. This is not an issue for learning arithmetic operations, but could lead to biased models in more general use cases.

The methodology presented in the experiments in Sec. 4.1 is not indented to be used for real-world epidemiological predictions. They are merely demonstrating that the NPU can learn an ODE with
fractional powers. For an application of the NPU much more post-processing has to be done to ensure the reliability of the results.

Acknowledgements and Disclosure of Funding

The research presented in this work has been supported by the Grant Agency of Czech Republic no. 18-21409S. The authors also acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 “Research Center for Informatics”.

We thank the authors of the Julia packages Flux.jl [Innes et al., 2018] and DifferentialEquations.jl [Rackauckas and Nie, 2017]).

References


