DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, SWEDEN 2015

A Novel Transfer Function for Continuous Interpolation between Summation and Multiplication in Neural Networks

WIEBKE KÖPP

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

A Novel Transfer Function for Continuous Interpolation between Summation and Multiplication in Neural Networks

Kontinuerlig interpolering mellan addition och multiplikation med hjälp av en lämplig överföringsfunktion i artificiella neuronnät

WIEBKE KÖPP

wiebkek@kth.se

Master's Thesis in Machine Learning

August 25, 2016

Supervisor:	Erik Fransén
Examiner:	Anders Lansner
Project Provider:	TUM BRML Lab
TUM BRML Supervisor:	Patrick van der Smagt
TUM BRML Advisor:	Sebastian Urban

Acknowledgments

This thesis, conducted at the biomimetic robotics and machine learning (BRML) group at Technische Universität München (TUM), would not have been possible without the support and encouragement of many people to whom I am greatly indebted.

First, I would like to express my gratitude to my supervisor Patrick van der Smagt for giving me the opportunity to work on this very interesting topic and warmly welcoming me into the BRML family.

I would also like to thank my advisor Sebastian Urban. His knowledge and feedback have been crucial in advancing and completing this work.

A special thanks goes to my supervisors at KTH, Erik Fransén and Anders Lansner, who not only provided me with their expertise, but also gave me a lot of freedom in working on this project.

Furthermore, I am grateful for many insightful discussions with my colleagues at BRML Max Sölch, Marvin Ludersdorfer and Justin Bayer.

Finally, I would like to thank my family and friends for the unconditional love and support.

Abstract

In this work, we present the implementation and evaluation of a novel parameterizable transfer function for use in artificial neural networks. It allows the continuous change between summation and multiplication for the operation performed by a neuron. The transfer function is based on continuously differentiable fractional iterates of the exponential function and introduces an additional parameter per neuron and layer. This parameter can be determined along weights and biases during standard, gradient-based training. We evaluate the proposed transfer function within neural networks by comparing its performance to conventional transfer functions for various regression problems. Interpolation between summation and multiplication achieves comparable or even slightly better results, outperforming the latter on a task involving missing data and multiplicative interactions between inputs.

Referat

I detta arbete presenterar vi implementationen och utvärderingen av en ny överföringsfunktion till användning i artificiella neuronnät. Den tillåter en kontinuerlig förändring mellan summering och multiplikation för operationen som utförs av en neuron. Överföringsfunktionen är baserad på kontinuerligt deriverbara bråkiterationer av den exponentialfunktionen och introducerar en ytterligare parameter för varje neuron och lager. Denna parameter kan bestämmas längs vikter och avvikelser under vanlig lutningsbaserad träning. Vi utvärderar den föreslagna överföringsfunktionen inom neurala nätverk genom att jämföra dess prestanda med konventionella överföringsfunktioner för olika regressionsproblem. Interpolering mellan summering och multiplikation uppnår jämförbara eller något bättre resultat, till exempel för en uppgift som gäller saknade data och multiplikativ interaktion mellan indata.

Contents

1. Introduction

Conventional artificial neural networks (ANNs) compute weighted sums over inputs in potentially multiple layers. Considering one layer in an ANN with multiple inputs x_j and multiple outputs y_i, the value of one output neuron y_i is computed as

$$y_i = \sigma\left(\sum_j W_{ij} x_j\right). \tag{1.1}$$

where σ is a non-linear transfer function, e.g. $\sigma(x) = 1/(1 - e^{-x})$ or $\sigma(x) = \tanh(x)$. ANNs computing the operation above for all neurons will be referred to as additive ANNs.

An alternative operation has been first explored in Durbin and Rumelhart (1989) where inputs are instead multiplied and weights are included as powers to the inputs. Then, y_i is given by $y_i = \sigma\left(\prod_j x_j^{W_{ij}}\right)$. This expression can be reformulated as

$$y_i = \sigma\left[\exp^{(1)}\left(\sum_j W_{ij} \exp^{(-1)} x_j\right)\right]. \tag{1.2}$$

Here, $f^{(n)}$ denotes the iterated application of an invertible function $f : \mathbb{C} \to \mathbb{C}$, so that $f^{(1)}$ is function f itself, $f^{(0)}(z) = z$ and $f^{(-1)}$ is the inverse of f. In general, the following two equations hold:

$$f^{(n)}(z) = \underbrace{f \circ f \circ \cdots \circ f}_{n \text{ times}}(z) \tag{1.3}$$

$$f^{(n)} \circ f^{(m)} = f^{(n+m)}, \quad n, m \in \mathbb{Z} \tag{1.4}$$

Through close examination of (1.2), we see that replacing the iteration arguments $n = 1$ and $n = -1$ of $\exp^{(n)}$, occurring in (1.2), by 0 leads exactly to expression (1.1). Therefore, finding a function $\exp^{(n)}$ that is defined for values $n \in \mathbb{R}$, which obviously includes all real values between -1 and 1, can be used for smooth interpolation between the two settings. $\exp^{(n)}$ is called a *fractional iterate* of the exponential function. In order to be

used with ANNs trained by gradient-based optimization methods, such as stochastic gradient descent, the function additionally needs to be continuously differentiable.

In this report, we will show how fractional iterates of the exponential function can be used within a novel transfer function that enables interpolation between summation and multiplication for the operation of a neuron. With the idea first introduced in Urban and Smagt (2015), one of the contributions of this work is the implementation of two different approaches for fractional iteration of the exponential function for integration within the widely used machine learning library Theano. Also, we address the trade-off between accuracy and computational efficiency in computationally intense operations by both error-analysis and interpolation of the involved functions. Furthermore, the proposed transfer function is evaluated by comparing its performance to conventional transfer functions for various regression problems. Here, we mainly focus on datasets that incorporate multiplicative interaction between inputs, as we expect these will benefit most from the novel transfer function.

The remainder of this report is structured as follows. First, a brief overview on artificial neural networks and the procedure to train them is given. Additionally, this chapter contains related work and provides necessary background information for concepts applied in the implementation of the proposed transfer function. The following chapter serves as a summary of Urban and Smagt (2015), where the transfer function interpolating between summation and multiplication has first been suggested. We discuss how a solution for $\exp^{(n)}$ can be derived and then show how this solution is used as a transfer function within ANNs. The proof of differentiability for the first alternative is an original contribution of this work. Experimental results obtained by applying the novel transfer function within ANNs are discussed in Ch. 5. Finally, in concluding this report, some pointers to future work are given.

2. Background

This chapter covers some concepts that are either necessary for understanding how the transfer function based on fractional iterates of the exponential function is derived or have been applied in its implementation and evaluation. Additionally included is a brief overview on used software components, namely Theano and CUDA.

2.1. Feed-Forward Artificial Neural Networks

As universal function approximators (Hornik, Stinchcombe, and White, 1989), feed-forward neural networks or multi-layer perceptrons have been a popular tool for solving regression and classification problems. Here, we recall their basic elements. A more detailed introduction can be found in Bishop (2013, Ch. 5). For an in-depth historical review of research in the field, the reader is referred to Schmidhuber (2015).

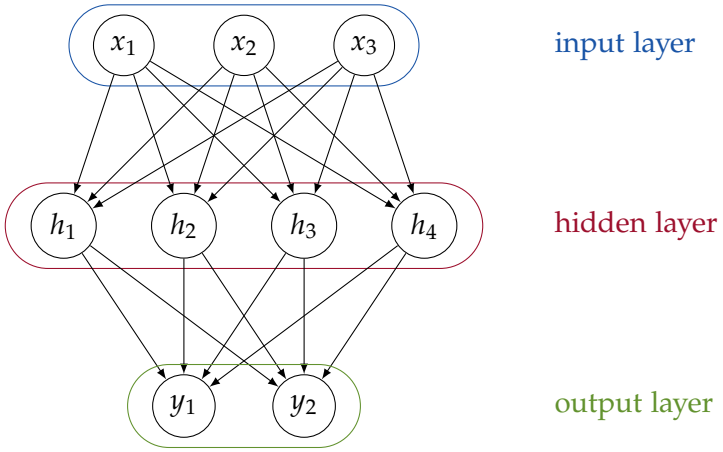


Figure 2.1.: A two-layer feed-forward neural network with three input neurons, one hidden layer of four neurons and two outputs.

A multi-layer perceptron, as depicted in Fig. 2.1, consists of an input and an output layer of neurons as well as an arbitrary number of hidden layers. The number of input

and output neurons depends on the dimensions of the data the network will be trained on. An ANN for approximation of function $f\colon X \to Y$ with $X \subseteq \mathbb{R}^{d_x}, Y \subseteq \mathbb{R}^{d_y}$ will have an input layer with d_x and an output layer of d_y neurons. As mentioned in Ch. 1, in conventional ANNs, the output of one layer is computed as the weighted sum of its inputs. Often, a so-called *bias* is then added, before propagating the result through a non-linear transfer function. The output vector $\mathbf{y} \in \mathbb{R}^{d_y}$, where each entry contains the output for one neuron of the output layer, is then computed as

$$\begin{aligned}\mathbf{y} &= \sigma_2[\mathbf{W^2 h} + \mathbf{b}^2] \\ &= \sigma_2[\mathbf{W^2}\sigma_1(\mathbf{W^1 x} + \mathbf{b^1}) + \mathbf{b}^2]\end{aligned}$$

for an ANN with one hidden layer. Here, the input vector is denoted with $\mathbf{x} \in \mathbb{R}^{d_x}$ and $\mathbf{h}$ represents the result vector of the hidden layer with a given number of entries d_h. $\mathbf{W^1} \in \mathbb{R}^{d_h \times d_x}$ and $\mathbf{W^2} \in \mathbb{R}^{d_y \times d_h}$ refer to the weights, while $\mathbf{b^1} \in \mathbb{R}^{d_h}$ and $\mathbf{b^2} \in \mathbb{R}^{d_y}$ constitute the biases for the first and second layer respectively. Recalling the graphical representation of the ANN in Fig. 2.1, one can think of each entry in the weight matrices as belonging to one of the edges. Finally, σ_1 and σ_2 are non-linear *transfer functions*. Together, $\mathbf{W^1}, \mathbf{W^2}, \mathbf{b^1}$ and $\mathbf{b^2}$ form the set of parameters that are optimized in training the two-layer ANN. The evaluation of the equations above is considered the *forward-propagation* through the network.

Without an additional non-linear transfer or activation functions, an ANN would only be able to model linear combinations of inputs, which occur only rarely in real-word datasets. The most commonly used transfer functions (see Fig. 2.2) are the sigmoid, tanh and rectifier functions. Neurons or units with the latter as their transfer function, are also called rectified linear units (ReLUs).

Neural networks are trained through *back-propagation* (Rumelhart, G. E. Hinton, and Williams, 1986). This procedure involves the computation of an *error function*, that gives an estimate on how well the network is able to approximate given training data for the current parameter configuration. Alternative terms for the error function are loss or cost. Assuming a differentiable cost C, the gradient of that cost with respect to each individual parameter of the network can be computed by iterated application of the chain rule of differential calculus. This approach is very similar to reverse-mode automatic differentiation on expression graphs which is detailed in Sec. 2.3.2. Cost C can thus be minimized with first-order gradient-based optimization methods, the simplest of which is gradient descent (GD). During learning, an individual weight is updated following

$$w \leftarrow w - \alpha \frac{\partial C}{\partial w}, \tag{2.1}$$

where α represents the *learning rate* controlling the speed and nature of learning.

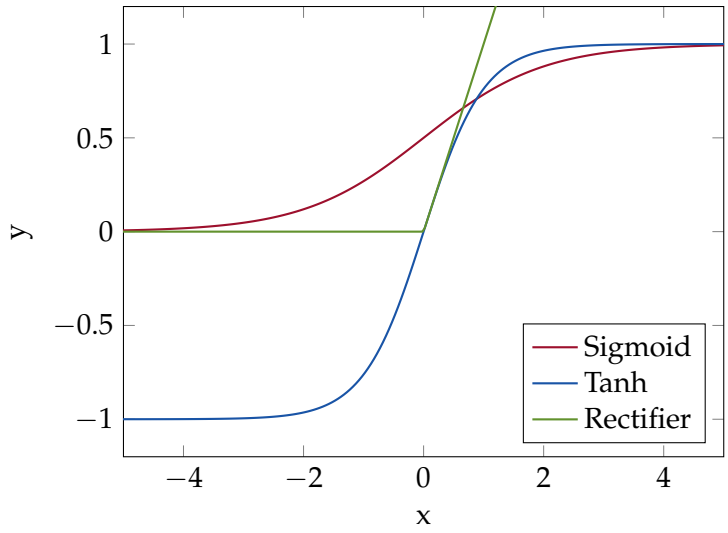


Figure 2.2.: Commonly used non-linear transfer functions for ANNs: Sigmoid ($\sigma_1 = \frac{1}{1+e^{-x}}$), Tanh ($\sigma_2 = \tanh(x)$) and Rectifier ($\sigma_3 = \max(x, 0)$).

An example for a loss function often used in regression problems is the mean-squared error (MSE) function. Its equation is given below as

$$\mathbf{C_{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y_i} - \mathbf{t_i})^2, \tag{2.2}$$

with $\mathbf{y_i}$ denoting the output of the ANN for training sample i and $\mathbf{t_i}$ containing the target values for that training sample.

Typically in order to accelerate learning, not the entire training data is used for computing weight updates, but a so-called *mini-batch* or even just a single data point. This optimization approach is called *stochastic gradient-descent*. Over the years many extensions to gradient-based optimization in the context of ANNs have been introduced. Among these are the use of a momentum (Rumelhart, G. E. Hinton, and Williams, 1986) which incorporates the parameter update of the previous step, or the use of entirely adaptive optimizers such as Adam (Kingma and Ba, 2015), RMSProp (Tieleman and G. Hinton, 2012) or Adadelta (Zeiler, 2012). These adaptive optimizers have in common that they change the effective learning rate over the course of training and use past gradients or gradient-magnitudes in order to accommodate the different nature of individual parameters.

2.2. Mathematical Concepts

2.2.1. Differentiation of Real- and Complex-Valued Functions

The derivative of a function $f\colon \mathbb{R} \to \mathbb{R}$ at point x_0 in its domain is defined as

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}\,. \tag{2.3}$$

f is differentiable if the limit above exists for all points x_0 in its domain (Freitag and Busam, 2009).

In complex analysis, the derivative of a function $f\colon \mathbb{C} \to \mathbb{C}$ is defined in the same way as in real analysis,

$$f'(z_0) = \lim_{z \to z_0} \frac{f(z) - f(z_0)}{z - z_0} = \lim_{h \to 0} \frac{f(z_0 + h) - f(z_0)}{h}\,. \tag{2.4}$$

A function f is said to be complex differentiable or analytic if these limits exist for all z_0. Other concepts from real differentiability such as product and quotient rule apply as well. However, z and h are complex values and therefore z_0 cannot only be approached from left and right, but any direction.

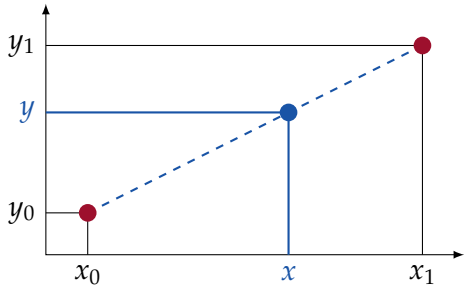
For a function $f(z) = f(x + iy) = u(x, y) + iv(x, y)$, we derive

$$\begin{aligned}
&\lim_{h \to 0} \frac{f((x + h) + iy) - f(x + iy)}{h} \\
=&\lim_{h \to 0} \frac{u(x + h, y) - u(x, y)}{h} + \frac{iv(x + h, y) - v(x, y)}{h} \\
=&\frac{\partial u}{\partial x} + i\frac{\partial v}{\partial x}
\end{aligned}$$

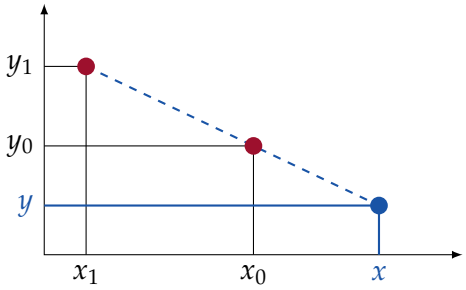
for the limit along the real axis and

$$\begin{aligned}
&\lim_{h \to 0} \frac{f(x + i(y + h)) - f(x + iy)}{ih} \\
=&\lim_{h \to 0} \frac{u(x, y + h) - u(x, y)}{ih} + \frac{iv(x, y + h) - v(x, y)}{ih} \\
=&-i\frac{\partial u}{\partial y} + \frac{\partial v}{\partial y}
\end{aligned}$$

for the limit along the complex axis.



(a) Linear extrapolation of $y \approx f(x)$ for value x based on the two closest known points.



(b) Linear extrapolation of $y \approx f(x)$ for value x based on the two closest known points. In this example $x_1 < x_0 < x$ holds. Obviously, there exists a second case in one-dimensional extrapolation, where $x < x_0 < x_1$.

Figure 2.3.: Linear interpolation and extrapolation in one dimension.

The limits above must be equal in order for f to be complex differentiable. By equating coefficients, we derive that the following equations, also known as Cauchy-Riemann equations, must hold for complex differentiable functions,

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y}, \quad \text{and} \quad \frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}. \tag{2.5}$$

2.2.2. Linear Interpolation and Extrapolation

A widely-used approach for reducing processing time for evaluation of computationally intense function are look-up tables (Tang, 1991), where a fixed number of function values $f(x) = y$ are precomputed either at program startup or stored and then read from file. Values for inputs x not included in the table are approximated linearly. With known input-value pairs (x_0, y_0) and (x_1, y_1), the linear approximation $f(x)$ with $x_0 < x < x_1$ is computed as

$$f(x) \approx \frac{x_1 - x}{x_1 - x_0} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1). \tag{2.6}$$

An illustration of this principle can be seen in Fig. 2.3a.

By first interpolating in one dimension and then using the same approach for consecutive dimensions (2.6) can be generalized to multiple dimensions (Cheney and Kincaid, 2012)). Bilinear interpolation with known function values for points p_{00}, p_{01}, p_{10} and p_{11}, where $p_{00} = (x_0, y_0), p_{01} = (x_0, y_1), p_{10} = (x_1, y_0)$ and $p_{11} = (x_1, y_1)$ and unknown value for $p = (x, y)$ falling inside the rectangle spanned by these four point is computed in two steps. First, linear interpolation in x-direction yields

$$\begin{aligned} f(x, y_0) &\approx \frac{x_1 - x}{x_1 - x_0} f(p_{00}) + \frac{x - x_0}{x_1 - x_0} f(p_{10}) \\ f(x, y_1) &\approx \frac{x_1 - x}{x_1 - x_0} f(p_{01}) + \frac{x - x_0}{x_1 - x_0} f(p_{11}) \,. \end{aligned} \tag{2.7}$$

Linear interpolation in the values computed above results in the final approximation f(x, y) where

$$f(x, y) \approx \frac{y_1 - y}{y_1 - y_0} f(x, y_0) + \frac{y - y_0}{y_1 - y_0} f(x, y_1) \,. \tag{2.8}$$

Trilinear interpolation follows the same principle, but with one additional dimension.

Linear extrapolation (Fig. 2.3b) is a common method for computing values for input values falling outside of the range of precomputed values. Considering again the one-dimensional case, the point closest to x and the point next to that point are used to approximate $f(x)$. Expressed in terms of these two points x_0 and x_1, where x_0 is the point closer to x, $f(x)$ is approximated by

$$f(x) \approx f(x_0) + \frac{x - x_0}{x_1 - x_0} (f(x_1) - f(x_0)) \,. \tag{2.9}$$

2.3. Software Components

2.3.1. CUDA

CUDA (Nvidia Cooperation, 2015a) is a programming framework by NVIDIA that enables the usage of GPUs for general purpose parallel computing. CUDA C is its interface to the C programming language.

A GPU consists of a number of multiprocessors composed of multiple cores, which execute groups of threads concurrently. Managing those threads is done using an architecture called SIMT (Single-Instruction, Multiple-Thread), where all cores of one multiprocessor execute *one common instruction* at the same time (see Fig. 2.4). This means that programs not involving data-dependent branching, where all threads follow the same execution path, are most efficient.

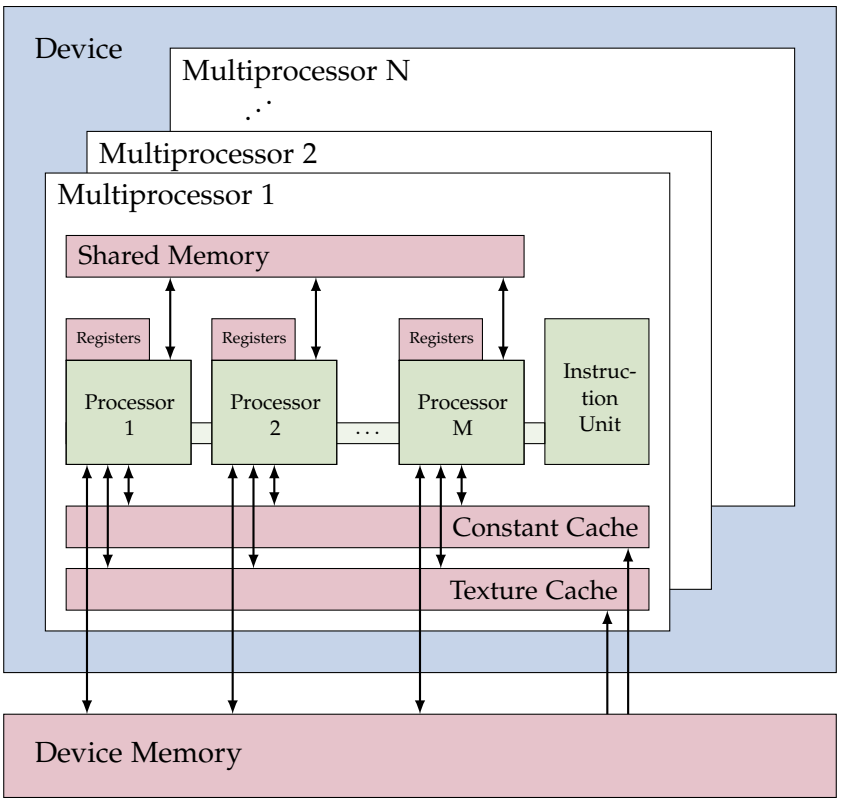


Figure 2.4.: A set of SIMT multiprocessors where multiple processors and threads share one instruction unit and have access to memory of various kind [Adapted from Nvidia Cooperation (2015b, Ch. 3)].

Threads have access to several types of memory which differ in properties like latency, lifetime and scope. One of those memory types is a special kind of read-only memory called texture memory. It can be used for accessing one-, two- or three-dimensional data and provides different filtering and addressing modes. The filtering mode refers to the manner in which a value is computed based on the input coordinates. One possibility is linear filtering which performs linear, bilinear or trilinear interpolation between neighboring elements using 9-bit fixed point numbers for the coefficients weighing the elements. Handling of coordinates that fall out of the range is specified by the addressing mode. Here, it is possible to either return zero, return the value at the border of the coordinate range, return the value by warping or mirroring the texture at the border.

2.3.2. Theano

Theano (Bergstra, Breuleux, Bastien, et al., 2010; Bastien, Lamblin, Pascanu, et al., 2012) is a Python library that uses a symbolic representation of mathematical expressions to build expression graphs which are then used to generate and compile C++ or CUDA C code for computation on a central processing unit (CPU) or graphical processing unit (GPU). The nodes in expression graphs represent either variables of specified datatype and dimension or mathematical operators defining how variables are combined. Before generating code, expression graphs are optimized in terms of computational efficiency and numerical stability, e.g. by elimination of unnecessary computations such as devision by one or manipulation of the computational order.

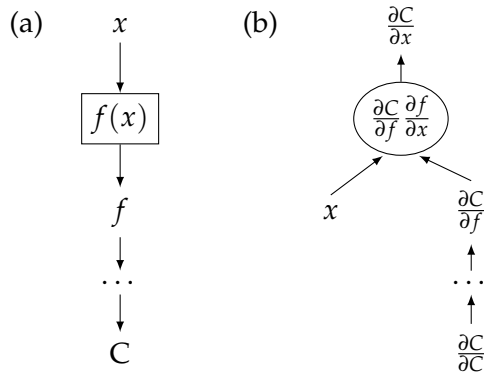


Figure 2.5.: (a) shows the forward traversal of an expression graph while (b) depicts the reverse mode for differentiation: The gradient of an operator computing $f(x)$ has to be specified as an expression graph. Its inputs are x and $\partial C/\partial f$ which has been derived by previous back-propagation steps that started with $\partial C/\partial C = 1$ and its output has to evaluate to $\partial C/\partial x$. $\partial f/\partial x$ can usually be either expressed through a graph containing the operator itself or a designated gradient operator.

Expression graphs additionally allow to automatically derive the gradient with respect to any parameter involved in the graph by a method called reverse-mode automatic differentiation (Griewank, 2003; Baydin, Pearlmutter, and Radul, 2015). The gradient of scalar value $C\colon \mathbb{R}^n \to \mathbb{R}$ is computed by traversing its expression graph from output $C(x_1, \ldots, x_n)$ to inputs $x_1, \ldots, x_n$ while iteratively applying the chain rule of differential calculus. The process builds a new expression graph using specifications of the gradient of each node's output with respect to its inputs. The computation scheme is displayed in Fig. 2.5, where a node with one input and output is involved in the computation of a cost C.

For computations that cannot be expressed by combining already provided operators, a custom operator can be introduced by directly providing an implementation in C++ or CUDA C and a symbolic expression for the gradient. It is possible to define a new operator for a scalar only and let Theano automatically generate versions that apply the scalar operator element-wise to vectors, matrices or tensors.

On the GPU, Theano computations always use single precision floating-point numbers. Thus, even though the intermediate values in self-introduced computations can have double precision, all inputs and outputs of operators have single precision.

2.4. Further Related Work

Even though multiplicative interactions have been found useful for certain regression and classification problems a long time ago (Durbin and Rumelhart, 1989), training of such networks was challenging due to complex error landscapes (Schmitt, 2002; Engelbrecht and Ismail, 1999). Some approaches therefore utilized other optimization procedures, such as genetic algorithms to train ANNs with product units (Janson and Frenzel, 1993). Additionally, a deliberate decision determining which of the two operations a specific neuron should perform was necessary, as no efficient algorithm to learn automatically whether a particular neuron should sum or multiply its inputs was known. As a result, most research in the past 25 years focused on pure additive networks. Recently, multiplicative interactions have resurfaced (Droniou and Sigaud, 2013; Sutskever, Martens, and G. E. Hinton, 2011), but to the best of our knowledge efficiently learning from data whether a particular neuron should perform an additive or multiplicative combination of its inputs has not been attempted. Additive ANNs are highly inspired by biology. As Schmitt (2002) suggests, multiplicative interaction between neurons are biologically justified as well. This has since been investigated by others, e.g. Schnupp and King (2001) and Gabbiani, Krapp, Hatsopoulos, et al. (2004).

There exist other approaches where the transfer function is learned together with the usual parameters in ANNs. In both Agostinelli, Hoffman, Sadowski, and Baldi (2014) and He, Zhang, Ren, and Sun (2015), learnable variants of ReLUs are suggested, that incorporate learning the transfer function in standard gradient-based training. Previous approaches followed a hybrid approach, where the transfer function is adapted through a genetic algorithm and other parameters are found in a conventional way (Yao, 1999; Turner and Miller, 2014).

We will investigate two approaches for fractional iteration of the exponential function, one of which introduces the use of complex numbers into ANNs. Some considerations in training of complex-valued ANNs have been studied in Hirose (2012).

3. Fractional Iterates of the Exponential Function

With the necessity to find an explicit solution for fractional iteration of the exponential function, the following sections elaborate two possible solutions. Note that the first solution lacks support for a range of input values, which motivates the second one, which in turn exhibits singularities at a distinct number of inputs. Additionally, the use of fractional iteration within ANNs is explained. For a thorough overview on functional equations for function iteration, see Kuczma, Choczewski, and Ger (1990).

3.1. Abel's Functional Equation

Abel's functional equation is given by

$$\psi(f(x)) = \psi(x) + \beta \tag{3.1}$$

for $f\colon \mathbb{C} \to \mathbb{C}$ and $\beta \neq 0$. Simplifying $\psi(f^n(x))$ using (3.1) leads to

$$f^{(n)}(x) = \psi^{-1}(\psi(x) + n\beta)\,. \tag{3.2}$$

The equation above is not restricted to whole numbers n. We therefore assume $n \in \mathbb{R}$ and use (3.1) with $f(x) = \exp(x)$ to find the fractional exponential function $f^{(n)}(z) = \exp^{(n)}(z)$. With $\beta = 1$ and $f^{-1}(x) = \log(x)$, Abel's functional equation can be reformulated as

$$\psi(x) = \psi(\log(x)) + 1\,. \tag{3.3}$$

Assuming $\psi(x) = x$ for $0 \leq x < 1$, we can find a solution for ψ which is valid on all of $\mathbb{R}$. Since we have $\log(x) < x$ for any $x \geq 1$, multiple applications of (3.3) will, at some point, lead to an argument for ψ that falls into the interval where we assume ψ to be linear. In case $x < 0$, we can use $\psi(x) = \psi(\exp(x)) - 1$ to reach the desired interval, since $0 < \exp(x) < 1$ for $x < 0$. In combination, this leads to a piece-wise defined solution for ψ,

$$\psi(x) = log^{(k)}(x) + k \tag{3.4a}$$

$$\text{with } k \in \mathbb{N} \cup \{-1, 0\} : 0 \leq \log^k(x) < 1\,. \tag{3.4b}$$

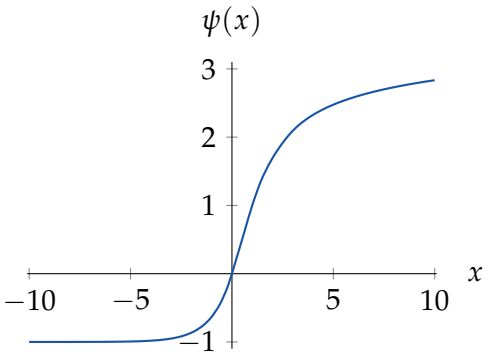


Figure 3.1.: A continuously differentiable solution $\psi(x)$ for Abel's equation (3.1) with $f(x) = \exp(x)$.

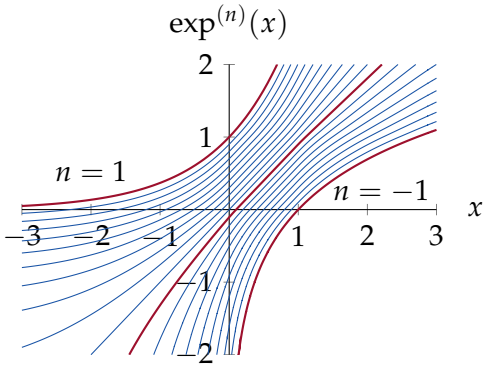


Figure 3.2.: Real-valued fractional exponential function $\exp^{(n)}(x)$ as specified in (3.6) for $n \in \{-1, -0.9, \ldots, 0, \ldots, 0.9, 1\}$.

The function is displayed in Fig. 3.1.

Because ψ is monotonically increasing on $\mathbb{R}$ with range $(-1, \infty)$, its inverse ψ^{-1} exists and is computed as

$$\psi^{-1}(\psi) = \exp^{(k)}(\psi - k) \tag{3.5a}$$

$$\text{with } k \in \mathbb{N} \cup \{-1, 0\} : 0 \leq \psi - k < 1 \tag{3.5b}$$

The real-valued fractional exponential function, depicted in Fig. 3.2, can then be defined by using

$$\exp^{(n)}(x) = \psi^{-1}(\psi(x) + n)\,. \tag{3.6}$$

Note that this solution is not necessarily unique. The derivation of other solutions under different initial assumptions might be possible.

3.1.1. Continuous Differentiability of the Real-Valued Fractional Exponential

We first prove that ψ is differentiable and thus also continuous on $\mathbb{R}$. Here, we only have to consider points where the number of necessary iterations k changes. The pieces with constant k are differentiable since they are composed of differentiable functions. k increases at each $\exp^{(k)}(1)$ with $k \in \mathbb{N} \cup \{0\}$. Recalling the definition of real differentiability in (2.3), we show that the limit given there exists for $x_0 = \exp^{(k)}(1)$ by computing the left- and right-hand limit. The left-hand limit is

$$\begin{aligned}
&\lim_{x \to x_0^-} \frac{\psi(x) - \psi(\exp^{(k)}(1))}{x - \exp^{(k)}(1)} \\
&= \lim_{x \to x_0^-} \frac{\log^{(k)}(x) + k - [\log^{k+1}(\exp^{(k)}(1)) + (k+1)]}{x - \exp^{(k)}(1)} \\
&= \lim_{x \to x_0^-} \frac{\log^{(k)}(x) - 1}{x - \exp^{(k)}(1)} \overset{l'H}{=} \lim_{x \to x_0^-} \prod_{j=0}^{k-1} \frac{1}{\log^{(j)}(x)} \\
&= \prod_{j=0}^{k-1} \frac{1}{\exp^{(k-j)}(1)} \,.
\end{aligned}$$

In addition to l'Hôpital's rule (l'H), we make use of the fact that $\log^j(\exp^{(k)}(x)) = \exp^{k-j}(x)$ and $\log(1) = 0$. The right-hand limit is

$$\begin{aligned}
&\lim_{x \to x_0^+} \frac{\psi(x) - \psi(\exp^{(k)}(1))}{x - \exp^{(k)}(1)} \\
&= \lim_{x \to x_0^+} \frac{\log^{k+1}(x) + (k+1) - [0 + (k+1)]}{x - \exp^{(k)}(1)} \\
&= \lim_{x \to x_0^+} \frac{\log^{k+1}(x)}{x - \exp^{(k)}(1)} \overset{l'H}{=} \lim_{x \to x_0^+} \prod_{j=0}^{k} \frac{1}{\log^{(j)}(x)} \\
&= \prod_{j=0}^{k} \frac{1}{\exp^{(k-j)}(1)} \,.
\end{aligned}$$

The only difference in the two limits above is the upper limit of the product, introducing an additional factor $1/\exp^{(k-k)} = 1$ for $x \to x_0^+$. The limits are therefore equal.

Using an analogous proof, it can be shown that ψ^{-1} is differentiable as well. Furthermore, derivatives ψ' and $\psi^{-1'}$ are continuous. This can be shown with a similar analysis of left- and right-hand limit for $\exp^{(k)}(1)$.

3.1.2. Derivatives Involved in the Real-Valued Fractional Exponential

The derivates for ψ defined by (3.4a) and ψ^{-1} specified in (3.5a) are given as

$$\psi'(x) = \begin{cases} \prod_{j=0}^{k-1} \frac{1}{\log^{(j)}(x)} & k \geq 0 \\ \exp(x) & k = -1 \end{cases} \tag{3.7}$$

with k as specified in (3.4b) and

$$\psi^{-1'}(\psi) = \begin{cases} \prod_{j=1}^{k} \exp^{(j)}(\psi - k) & k \geq 0 \\ \frac{1}{\psi+1} & k = -1 \end{cases} \tag{3.8}$$

with k fullfiling (3.5b).

Consequently, the derivatives of $\exp^{(n)}$ defined by (3.6) are given by

$$\exp'^{(n)}(x) = \frac{\partial \exp^{(n)}(x)}{\partial x} = \psi'^{-1}(\psi(x) + n)\, \psi'(x) \tag{3.9a}$$

$$\exp^{(n')}(x) = \frac{\partial \exp^{(n)}(x)}{\partial n} = \psi'^{-1}(\psi(x) + n)\ . \tag{3.9b}$$

3.2. Schröder's Functional Equation

The solution found for $\exp^{(n)}$ in the previous section evaluates to $-\infty$ or is undefined in some cases, e.g. for $x < 0$ and $n = -1$, so that $\exp^{(n)}(x) = \log(x)$. To deal with negative x for $n \leq -1$ fractional iterates of the complex exponential function have to be used. The fractional complex exponential function can be obtained as a solution of Schröder's functional equation, defined as

$$\chi(f(z)) = \gamma\, \chi(z) \tag{3.10}$$

with $\gamma \in \mathbb{C}$.

Schröder's functional equation can be solved similarly to how we found a solution to Abel's equation before. Repetitive application of (3.10) leads to

$$f^{(n)}(z) = \chi^{-1}(\gamma^n\, \chi(z)) \tag{3.11}$$

and as before, we start with a local solution for χ and then generalize so that χ is defined on all $\mathbb{C}$.

Two important properties of the complex exponential function are that it is not injective since

$$\exp(z + 2\pi i) = \exp(z), \quad n \in \mathbb{Z}, i = \sqrt{-1} \tag{3.12}$$

and that it has a fixed point at, amongst others,

$$\exp(c) = c \approx 0.3181315 + 1.3372357i\,. \tag{3.13}$$

This means that the range for the inverse of exp has to be restricted on an interval such that the complex logarithm $\log(z)$ is an injective mapping. We choose $\log : \mathbb{C} \to \{z \in \mathbb{C} : -1 \leq \operatorname{Im} z \leq -1 + 2\pi\}$. This is commonly referred to as choosing a branch of the logarithm. For this branch of the logarithm, the fixed point c is attractive[1], which means that iterated application of the logarithm will converge to c. An exception are points $z \in \{0, 1, e, e^e \dots\}$ due to the occurrence of $\log^n(z) = 0$ at some iteration n.

Examining the behavior of the complex exponential function within a small circle with radius r_0 around fixed point c leads to approximation

$$\exp(z) = cz + c - c^2 + O(r_0^2) \tag{3.14}$$

It follows that points on a circle around c with radius r for a sufficiently small r are mapped to points on a larger circle with radius $|c|r$ since

$$\begin{aligned} \exp(c + re^{i\phi}) &= c(c + re^{i\phi}) + c - c^2 \\ &= c + cre^{i\phi}\,. \end{aligned} \tag{3.15}$$

All points are rotated by $\operatorname{Im} c \approx 76.618°$.

Substituting approximation (3.14) into (3.10) as $\chi(cz + c - c^2) = \gamma\chi(z)$ leads to a local solution of χ with

$$\chi(z) = z - c \quad \text{where } |z - c| \leq r_0 \text{ and } \gamma = c\,. \tag{3.16}$$

Since c is an attractive fixed point, we can use the reformulated version

$$\chi(z) = c\,\chi(\log(z)) \tag{3.17}$$

of (3.10) to find a solution to Schröder's equation for the complex-valued exponential function,

$$\chi(z) = c^k(\log^k(z) - c) \tag{3.18a}$$

$$\text{with } k = \arg\min_{k' \in \mathbb{N}_0} |\log^{(k')}(z) - c| \leq r_0\,. \tag{3.18b}$$

[1] See Urban and Smagt (2015) or Kneser (1950) for a detailed proof.

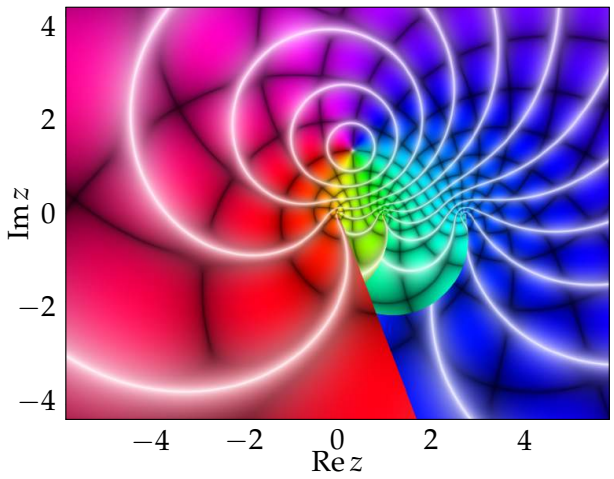


Figure 3.3.: Domain coloring plot (see App. A.1) of $\chi(z)$. Discontinuities arise at 0, 1, $\mathrm{e}, \mathrm{e}^{\mathrm{e}}, \ldots$ and stretch into the negative complex half-plane. They are caused by log being discontinuous at the polar angles -1 and $-1 + 2\pi$.

Solving for z gives the inverse,

$$\chi^{-1}(\chi) = \exp^{k}(c^{-k}\chi + c) \tag{3.19a}$$

$$\text{with } k = \arg\min_{k' \in \mathbb{N}_0} |c^{-k'}\chi| \leq r_0\,. \tag{3.19b}$$

Finally, the complex-valued fractional exponential is computed as

$$\exp^{(n)}(z) = \chi^{-1}(c^n\,\chi(z)) \tag{3.20}$$

Figs. 3.3 and 3.4 display function χ and samples of $\exp^{(n)}$ respectively.

3.2.1. Derivatives Involved in the Complex-Valued Fractional Exponential

Kneser (1950) shows that χ is an analytic function (and thus complex differentiable) for all points in $\mathbb{C}$ except $\{0, 1, e, e^e, \ldots\}$.

The derivatives of χ defined in (3.18a) and χ^{-1} specified in (3.19a) are given as

$$\chi'(z) = \prod_{j=0}^{k-1} \frac{c}{\log^{(j)} z} \tag{3.21}$$

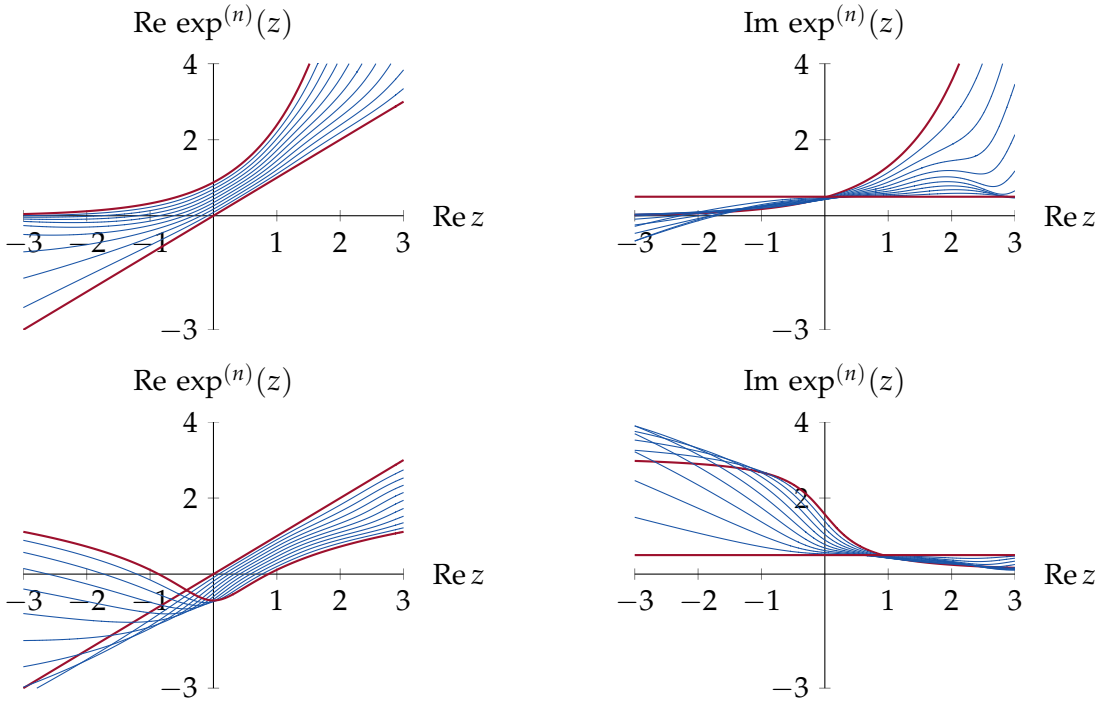


Figure 3.4.: Iterates of the exponential function $\exp^{(n)}(z)$ with $\operatorname{Im} z = 0.5$ for $n \in \{0, 0.1, \ldots, 0.9, 1\}$ (upper plots) and $n \in \{0, -0.1, \ldots, -0.9, -1\}$ (lower plots) obtained using the solution (3.20) of Schröder's equation. Exp, log and the identity function are highlighted in orange.

with k fulfilling (3.18b) and

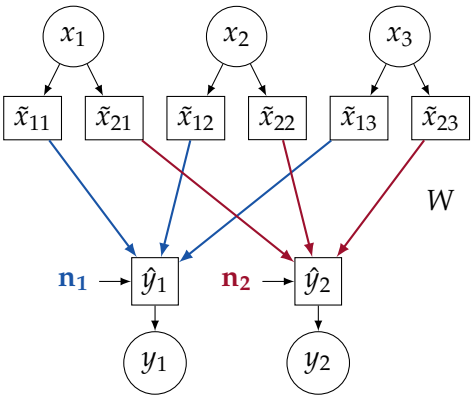
$$\chi^{-1'}(\chi) = \frac{1}{c^k} \prod_{j=1}^{k} \exp^{(j)}(c^{-k}\chi + c) \tag{3.22}$$

with k according to (3.19b).

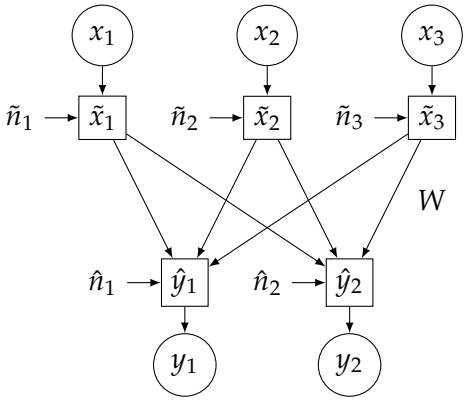
Fractional iteration of the complex exponential function defined in (3.20) is thus differentiated as

$$\exp'^{(n)}(z) = c^n \chi'^{-1}[c^n \chi(z)]\, \chi'(z) \tag{3.23a}$$

$$\exp^{(n')}(z) = c^{n+1} \chi'^{-1}[c^n \chi(z)]\, \chi(z)\,. \tag{3.23b}$$



(a) One layer of an ANN using the fractional exponential function, where output y_i is computed according to (3.24).



(b) One layer of an ANN using the fractional exponential function, where output y_i is computed according to (3.25).

Figure 3.5.: Possible structures for ANNs using the fractional exponential function. It is possible to use an additional non-linear transfer function σ after its second application in one layer. For complex valued fractional exponential, the weight matrix W contains complex numbers.

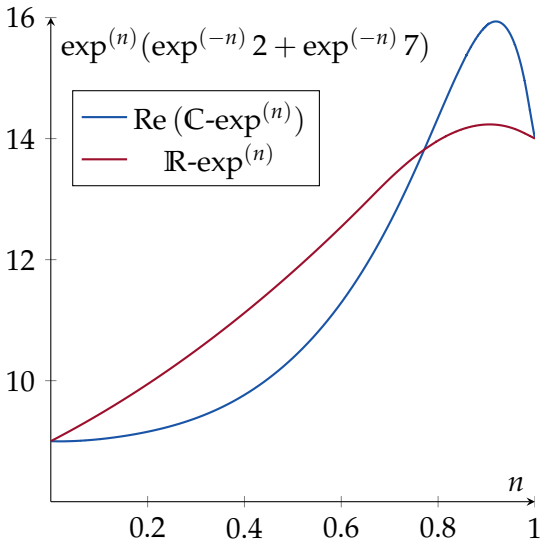


Figure 3.6.: Continuous interpolation between summation and multiplication for values 2 and 7 using real-valued ($\mathbb{R}$-exp$^{(n)}$) and complex-valued ($\mathbb{C}$-exp$^{(n)}$) fractional iteration of the exponential function.

3.3. Architecture of a Neural Net using Fractional Iterates of the Exponential Function

One possibility to use the fractional exponential within an ANN is by introducing a parameter n_i for each neuron y_i in a layer and compute its value as

$$y_i = \sigma\left[\exp^{(n_i)}\left(\sum_j W_{ij} \exp^{(-n_i)}(x_j)\right)\right]. \tag{3.24}$$

However, this requires to compute $\exp^{(-n_i)}(x_j)$ separately for each pair of neurons (i, j), increasing computational complexity significantly. Therefore, a different net structure is proposed by Urban and Smagt (2015) where

$$y_i = \sigma\left[\exp^{(\hat{n}_i)}\left(\sum_j W_{ij} \exp^{(-\tilde{n}_j)}(x_j)\right)\right]. \tag{3.25}$$

This decouples the parameters of $\exp^{(n)}$ and consequently arbitrary combinations of $\hat{n}_i$ and $\tilde{n}_j$ are possible. A visualization of the two possible structures above is given in Fig. 3.5.

The major advantage of interpolating between summation and multiplication for the operation of a neuron with fractional iterates of the exponential function is that the results can be computed for a whole layer utilizing matrix arithmetics, whose implementation on both CPU and GPU is very efficient, just as we showed in Sec. 2.1. An example for interpolating between summation and multiplication for specific values is shown in Fig. 3.6.

4. Implementation Details

This section details how the real- and complex fractional exponential functions are implemented in an efficient manner and discusses their integration in Theano. Considerations about numerical accuracy and improvements in computational efficiency are covered thereafter.

4.1. Implementation of the Real- and Complex-Valued Fractional Exponential

The functions ψ and ψ^{-1} for the real-valued fractional exponential as well as χ and χ^{-1} for the complex-valued fractional exponential and their derivatives are introduced to Theano as custom operators with both C++ and CUDA C code. The implementations for CPU and GPU differ mostly in syntax, thus pseudocode is presented here. All algorithms have a common basic structure. First, corner cases if any exist are handled. Then, the number of necessary iterations k is determined either through direct computation or iteration until the respective condition on k is fulfilled. If possible, the iterated logarithm or exponential is computed while attempting to fulfill the condition on k.

Algorithm 1 Computation of $\psi(x)$.

```
if x < 0 then return exp(x) − 1
else
    k ← 0
    while x > 1 and k < k_max(= 5) do
        x ← log(x); k ← k + 1
    end while
    return x + k
end if
```

Algorithm 1 and 2 compute ψ (3.4a) and ψ^{-1} (3.5a). The algorithms for the respective derivatives follow the same computational flow. However, for input values not applicable to the corner cases, a parameter initialized with one is added before the while-loop.

Algorithm 2 Computation of $\psi^{-1}(\psi)$. The value for ψ^{-1} with $\psi < -1$ would mathematically be $-\infty$. Similarly, inputs with $\psi \gtrsim 4.406$ evaluate to values larger than can be represented with single precision floating point numbers. Since infinite values cause problems in gradient-based learning, we restrict ψ^{-1} to the interval $[-10, 10]$. ψ_{min} and ψ_{max} are set as $\psi_{min} = -0.999955$ and $\psi_{max} = 2.83403$, so that $\psi^{-1}(\psi_{min}) = -10$ and $\psi^{-1}(\psi_{max}) = 10$.

if $\psi < \psi_{min}$ **then return** $\psi^{-1}(\psi_{min})$
else if $x > \psi_{max}$ **then return** $\psi^{-1}(\psi_{max})$
else
  $k = \lceil \psi - 1.0 \rceil$
  if $k < 0$ **then**
    return $\log(\psi - k)$
  end if
  $\psi \leftarrow \psi - k$
  while $k > 0$ **do**
    $\psi \leftarrow \exp(\psi)$; $k \leftarrow k - 1$
  end while
  return ψ
end if

Algorithm 3 Computation of $\chi(z)$. In order to evaluate χ even for the singularities at $0, 1, \mathrm{e}, \ldots$ a small epsilon is added to the input for those numbers. Choosing $\epsilon = 0.01$ and $\delta = 10^{-7}$ leads to a small absolute error for known relations $\exp^{(0)}(z) = z$ and $\exp^{(1)}(z) = \mathrm{e}^z$.

if $z \in \{0, 1, e\}$ **then** ▷ when e.g. $|z - 1| < \delta$
  $z \leftarrow z + \epsilon$
end if
$k \leftarrow 0$; cpow $\leftarrow 1.0$
while $|z - c| \geq r_0$ **and** $k < k_{max} (= 50)$ **do**
  $z \leftarrow \log(z)$; cpow $\leftarrow$ cpow $\cdot\, c$; $k \leftarrow k + 1$
end while
return cpow $\cdot\, (z - c)$

Algorithm 4 Computation of $\chi^{-1}(\chi)$. Here, a second loop over k is required as the argument of exp depends on k as well.

$k \leftarrow 0$;
while $|\chi| \geq r_0$ **and** $k < k_{max}(= 50)$ **do**
  $\chi \leftarrow \chi/c$; $k \leftarrow k + 1$
end while
$\chi \leftarrow \chi + c$
for $j \leftarrow 0$; $j < k$; $j \leftarrow j + 1$ **do**
  $\chi \leftarrow \exp(\chi)$
end for
return χ

It is used to accumulate intermediate logarithms or exponentials inside the loop for computing the products occurring in (3.7) and (3.8).

Algorithm 3 and 4 compute χ (3.18a) and χ^{-1} (3.19a). r_0 is set to 10^{-4}. As before, derivative computations follow the same algorithmic structure, but require an additional parameter for accumulation of intermediate logarithms and exponentials. Due to the lack of a complex datatype in Theano, complex-valued operators are realized with two in- and outputs representing real and imaginary part.

All algorithms above include a certain number of iterations k. Fig. 4.1 shows how k evolves for different inputs. This gives both an order of computational steps in the newly introduced operators and also motivates the choice for k_{max} as 5 and 50. Plots 4.1a and 4.1b display ranges for k, e.g. in Fig. 4.1a, a value of $x = 2$, requires $k = 1$ iterations. Plots 4.1c and 4.1d show the areas in which a certain number of iterations is required, e.g. in Fig. 4.1c, we can see that $k = 35$ holds for $z = 6 - 4i$. The large number of iterations in computations for the complex-valued fractional exponential function causes problems with both accuracy and speed.

4.2. Integration in Theano

In addition to an implementation, custom operators need to provide information on how their gradient is computed. For real-valued operators ψ and ψ^{-1} this is achieved by following the principle introduced in Section 2.3.2. Fig. 4.2 extends this principle to operators computing a complex-valued function $f(z) = f(x + iy) = u(x, y) + iv(x, y)$, e.g $\chi(z)$.

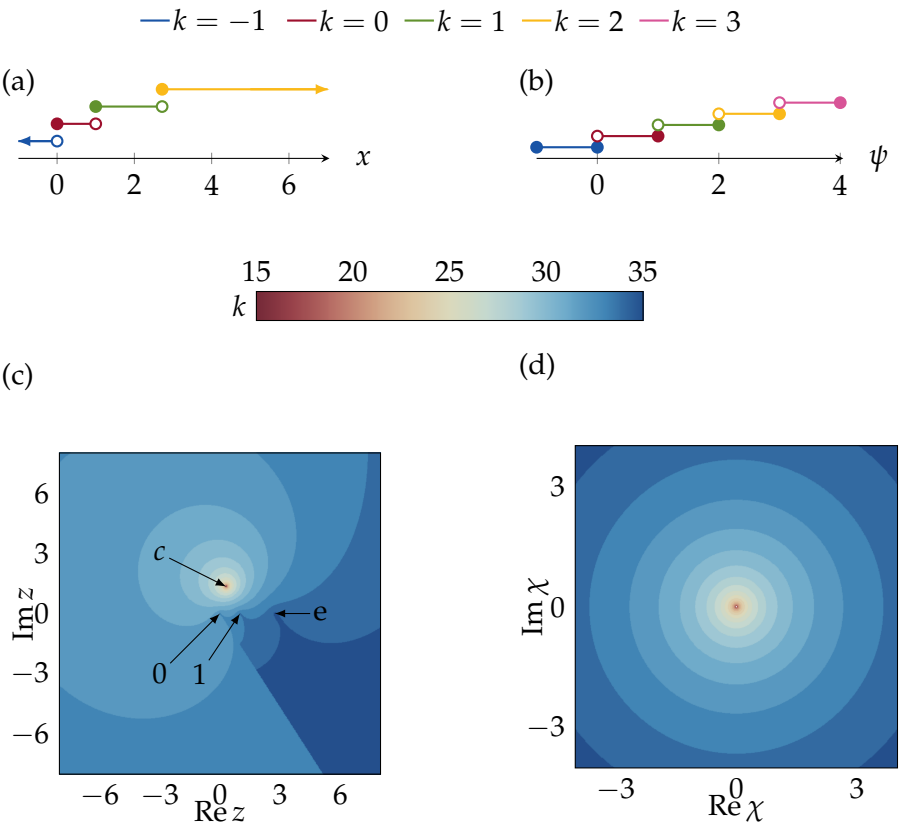


Figure 4.1.: Number of iterations k, thus order of computational steps for real ((a),(b)) and complex ((c),(d)) input values for (a) $\psi(x)$, (b) $\psi^{-1}(\psi)$, (c) $\chi(z)$, (d) $\chi^{-1}(\chi)$.

In our case, $f'(z)$, e.g $\chi'(z)$ is computed by another custom operator. As both χ and χ^{-1} fulfill the Cauchy-Riemann equations (2.5), we know that $\operatorname{Re} f' = \partial u/\partial x$ and $\operatorname{Im} f' = \partial v/\partial x$. Therefore, the expressions to be computed within reverse-mode differentiation of cost C simplify to

$$\frac{\partial C}{\partial x} = \frac{\partial C}{\partial u}\operatorname{Re} f' + \frac{\partial C}{\partial v}\operatorname{Im} f' \tag{4.1a}$$

$$\frac{\partial C}{\partial y} = -\frac{\partial C}{\partial u}\operatorname{Im} f' + \frac{\partial C}{\partial v}\operatorname{Re} f' . \tag{4.1b}$$

We have now fully defined custom operators for scalar values. Using a wrapper that generates a version that operates element-wise on multidimensional data, the operators can be combined for computing the real- and complex-valued fractional exponential within ANNs.

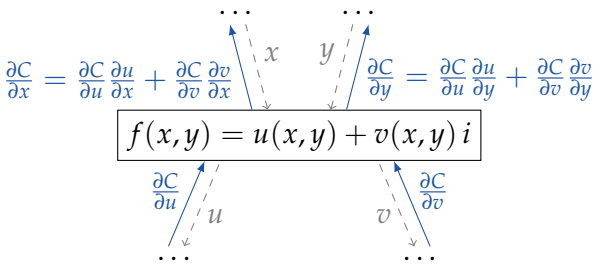


Figure 4.2.: The node computing the complex function $f(x+iy)$ with inputs x, y and outputs u, v has to specify the gradients of scalar cost C with respect to each input. The inputs x, y as well as the gradients $\partial C/\partial u$ and $\partial C/\partial v$ are given. $\partial C/\partial x$ and $\partial C/\partial y$ are computed as shown above and serve as inputs for the gradient computation of the nodes for which x, y are the outputs.

4.3. Issues with Precision and Accuracy

Floating point arithmetics using a fixed number of bits are subject to rounding errors of precision-dependent magnitude. Accumulation of those errors can lead to inaccurate results (Higham, 2002). This is particularly true for computations involving many steps, as each step potentially produces less accurate results due to previously made rounding errors.

Especially iterates of exp as they occur in calculating ψ^{-1} and χ^{-1} can amplify small input errors significantly even with a considerably small number of iterations. With a maximum of 4 steps on the considered interval with input between -8 and 8, numerical accuracy is not an issue for the real-valued fractional exponential. However, the complex-valued version typically involves a much larger number of steps. Subsequent sections provide an estimate on the relative error of iterating exp and show the influence of choosing either single or double precision on all expressions in numerical experiments.

4.3.1. Error Estimation

Assuming independence between input variables, the common formula to calculate propagation of error by estimating the standard variation σ_f for function $f(x, y, \dots)$ based on standard variations $\sigma_x, \sigma_y, \dots$ is given (Philip R. Bevington, 2003) by

$$\sigma_f^2 = \left(\frac{\partial f}{\partial x}\right)^2 \sigma_x^2 + \left(\frac{\partial f}{\partial y}\right)^2 \sigma_y^2 + \dots \tag{4.2}$$

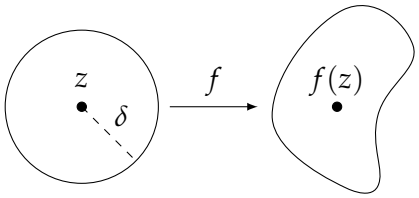


Figure 4.3.: For a continuous function f and an input z, that is subject to some small absolute error δ, the function value for the erroneous input will be located within an area bounded by the mapping of the δ-circle around z through f.

Thus, for $\exp(x+iy) = \exp(x)\,[\cos(y) + i\sin(y)]$ with $\hat{x} = \operatorname{Re}\exp(x+iy)$ and further assuming that $\sin(y)^2$ and $\cos(y)^2$ can be at most 1, we derive

$$\begin{aligned} \sigma^2_{\exp(x)} &= \exp(x)^2\sigma_x^2 \\ \sigma^2_{\cos_y} &= \sin(y)^2\sigma_y^2 \approx \sigma_y^2 \\ \Rightarrow \sigma_{\hat{x}} &= \exp(x)^2\sigma^2_{\cos(y)} + \cos(y)^2\sigma\exp(x)^2 \\ &\approx \exp(2x)\sigma_y^2 + \exp(4x)\sigma_x^4. \end{aligned} \tag{4.3}$$

This means that, in the worst case, the error grows exponentially in the input for just the real part of one iteration of exp. Consequently the estimate yielded by the generic method for error propagation shown above is too coarse for our purpose.

A better approximation on how an error in the input propagates for multiple iterations of exp can be derived as follows. When computing χ^{-1}, condition (3.19b) ensures that $\exp(z)$ is only evaluated for points $z \in C_0 = \{c + re^{i\phi} \mid \phi \in [0, 2\pi], r \leq r_0\}$. An error in the original input χ to χ^{-1} or rounding error introduced by division χ/c^k can also only lead to erroneous points within C_0. Furthermore, the exponential function is continuous. Therefore, values in close proximity in the input, will also be close to each other in the input (see Fig. 4.3).

An estimate on the average relative error $E(n)$ for computing n iterates of the exponential function of points $z \in C_0$ can therefore be derived by considering the expansion of C_0 under the same transformation. We approximate $E_{area}(n)$ as the square-root of the relation between the area covered by computing iterates of points within C_0 and the area of C_0.

$$E_{area}(n) = \sqrt{\frac{A_n}{\operatorname{area}(C_0)}} = \sqrt{\frac{\operatorname{area}(\exp^{(n)}(z) \mid z \in C_0)}{2\pi r_0^2}} \tag{4.4}$$

As mentioned before (see Section 3.2), points on a circle with small radius r around c are mapped to points on a larger circle with radius $|c|r$. With a rising number of

n	$E_{circle}(n)$	$E_{area}(n)$	$E_{single}(n)$	$E_{double}(n)$
26	3 910	3 978	3 949	7 319
27	5 375	5 552	5 463	10 125
28	7 389	7 860	7 604	14 102
29	10 156	11 441	10 764	19 963
30	13 960	17 621	15 733	29 136
31	19 189	30 623	24 504	45 340
32	26 376	70 222	43 765	80 839
33	36 255	328 253	131 981	245 512

Table 4.1.: Error estimation and numerically computed errors for n iterations. $E_{circle}(n)$ denotes the error computed under the assumption that the covered shape is and stays a circle. $E_{area}(n)$ refers to the error derived by (4.4). A notable difference begins to occur at iteration 27. The two estimations are compared to the numerically derived relative error of computing n iterations of exp for exact test-points with single and double precision.

iterations for $\exp^{(n)}$ this condition does not hold anymore and the covered area diverges from a circular shape. Then, the area can be derived by sampling points on C_0 and computing the area of the bounding polygon after iterating the points with exp. For $n \geq 33$ the shape becomes self-intersecting, since exp is not an injective function on $\mathbb{C}$. The development of the covered area is shown in Fig. 4.4. Table 4.1 shows the computed error estimations. Since numeric computation for iteration of exp for exact values yields relative errors of the same order of magnitude as our estimations, it can be concluded that the approximation by considering the development of the area is valid.

4.3.2. Numerical Verification

The findings above are verified in a numerical experiment, where $\chi(z)$, $\chi'(z)$, $\chi^{-1}(\chi)$, $\chi^{-1'}(\chi)$, $\exp^{(n)}(z)$, $\exp'^{(n)}(z)$ and $\exp^{(n')}(z)$ are evaluated for random numbers $z \in \{-8-8i, 8+8i\}$ and $n \in \{-1, 1\}$. The ground-truth values for the experiment are generated with Mathematica, which while generally using double precision computations as well, is optimized to use higher precision where it is deemed necessary for accurate results. χ^{-1} and $\chi^{-1'}$ use these 'true' values for χ as input.

Fig. 4.5 displays the relative errors computed as $|z - \hat{z}|/|z|$, where $\hat{z}$ refers to the numerically computed and z to the true value. As suspected, expressions involving

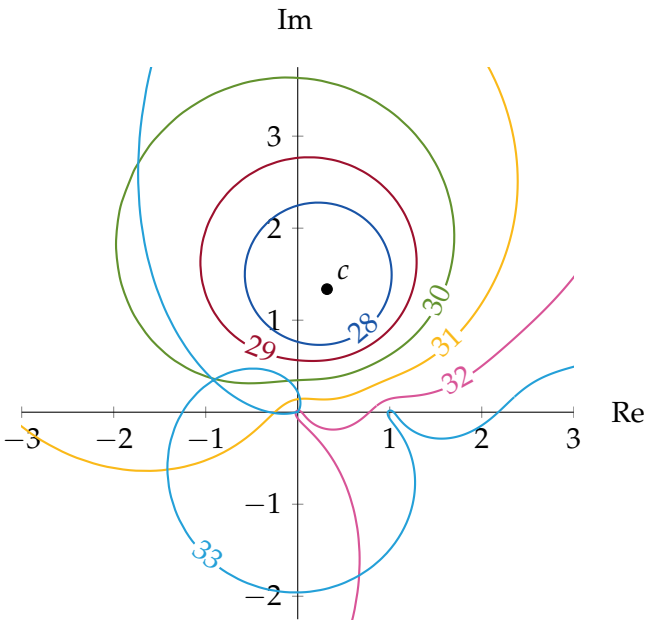


Figure 4.4.: Development of the area covered by points in C_0 after application of $\exp^{(n)}$ for different values of n.

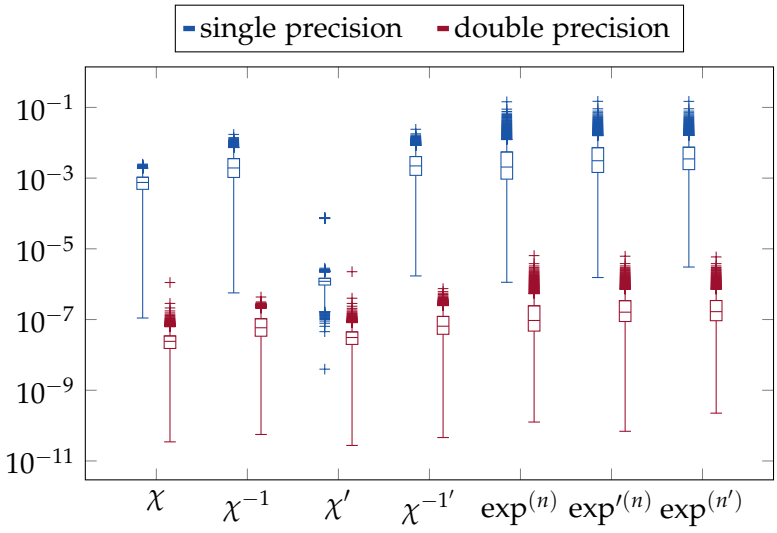


Figure 4.5.: Box-and-whiskers plot (see App. A.2) for the relative error $|z - \hat{z}|/|z|$ for the result of computing the respective expressions for one million complex test points with single and double precision.

the exponential function, have a higher relative error, even for exact inputs. Also, the difference between single and double precision is significant for all expressions.

The precision-related accuracy issues discussed above have a significant effect in the case of usage within an ANN, where $\exp^{(n)}$ is evaluated at least once per layer and error-prone gradient calculations can hinder learning. While the use of double precision provides higher accuracy, execution times on GPUs are significantly longer due to less optimized arithmetics. Except for the obvious use of more memory and even slower computation times due to less parallelization, a similar effect cannot be observed for CPU when switching from single to double precision.

4.4. Interpolation

In addition to the issues with accuracy, calculating the complex-valued $\exp^{(n)}$ and its derivatives is rather time-consuming as a result of the high number of computational steps when compared to regular transfer functions. Training ANNs with the suggested transfer function within reasonable time thus requires a more efficient computation procedure. We will consider two interpolation methods. Method A uses 2d-interpolation[1] to interpolate χ, χ^{-1} and derivatives for real and imaginary input and uses exact computation for c^n. Fractional iteration of the exponential function is then computed according to (3.20) with interpolated χ and χ^{-1}. Method B additionally utilizes parameter n to perform 3d-interpolation of $\exp^{(n)}$ directly.

Precomputed values for $\chi(z)$, $\chi'(z)$, $\exp^{(n)}(z)$, $\exp'^{(n)}(z)$ and $\exp^{(n')}(z)$ with inputs $z \in \{-8-8i, 8+8i\}$ and $n \in \{-1, 1\}$ sampled at different resolutions are derived with Mathematica and then saved in binary files. For inverse functions $\chi^{-1}(\chi)$ and $\chi^{-1'}(\chi)$, the range for χ is chosen so that it covers the range of $c^n\chi(z)$, thus $\chi \in \{-6.9-4i, 5.5+7.5i\}$. For computations on the CPU these binary files are then accessed through memory-mapping and an implementation of bi- or trilinear interpolation following the equations in Section 2.2.2. For computations on the GPU, the precomputed values are copied over to texture memory and can then be accessed through linear filtering (see Section 2.3.1).

For input values falling outside of the ranges specified above, one approach yielding continuous transitions at borders would be linear extrapolation. However, determining on which side or face of the shape defined by the sampling ranges an input point lies, requires a case differentiation with $3^d - 1$ cases, where d is the dimension (see Algorithm 5). Such data-dependent branching is not ideal for a SIMT architecture,

[1] The real and imaginary part of the argument are treated as two separate dimensions.

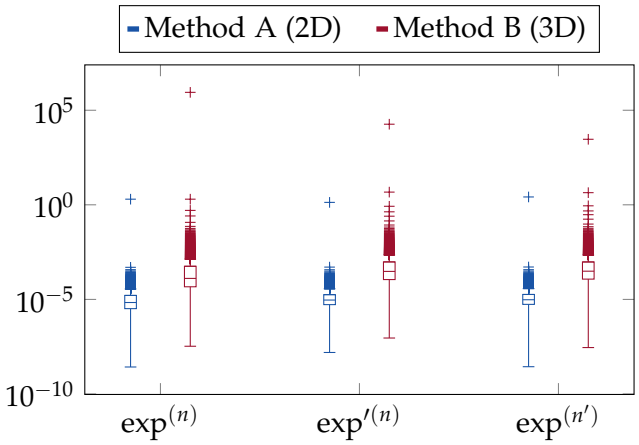


Figure 4.6.: Relative error for the result of computing the respective expressions for one million complex test points with the highest sampling resolution for method A (2d) and method B (3d).

where multiple threads share one instruction unit and therefore *all* threads have to execute a branch, if branching occurs for *at least one* of the threads. Thus, we use the mirroring addressing mode of texture memory instead. This yields both continuous transitions at borders and provides gradient-information at all input points.

Algorithm 5 Extrapolation in one dimension according to (2.9), where function $f(x)$ was precomputed with $x \in [x_{min}, x_{max}]$ and resolution r_x.

if $x_{min} \leq x \leq x_{max}$ **then**
  return regular interpolation
else
  if $x < x_{min}$ **then**
    $x_0 \leftarrow x_{min}, x_1 \leftarrow x_{min} - r_x$
  else
    $x_0 \leftarrow x_{max}, x_1 \leftarrow x_{max} + r_x$
  end if
  return $f(x_0) + \frac{x - x_0}{x_1 - x_0}(f(x_1) - f(x_0))$
end if

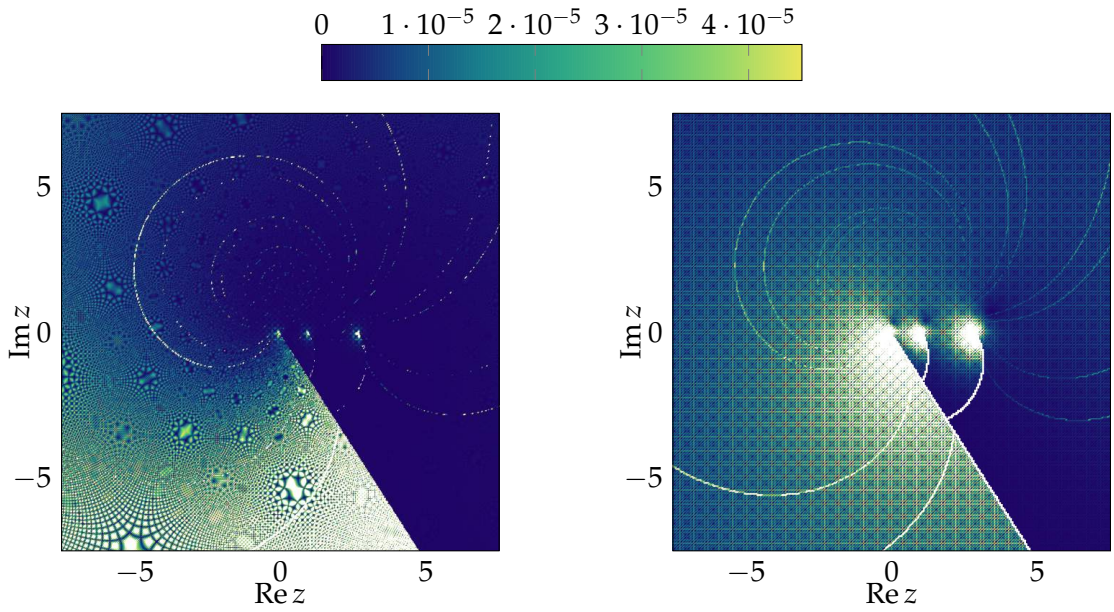
In order to evaluate different sampling resolutions with respect to runtime, accuracy and memory usage, we use the test setup discussed in Section 4.3.2. The results are shown in Tables 4.2 and 4.3. Higher resolutions than the ones specified are not feasible due to memory limitations.

	CPU		GPU		
r_z	e_{rel} [10^{-5}]	t [ms]	e_{rel} [10^{-5}]	t [ms]	M [MB]
0.1	27.45	269.13	31.05	9.29	0.62
0.05	6.757	274.57	9.698	10.38	2.45
0.01	0.278	388.48	1.313	15.80	60.91
0.0075	0.160	421.63	0.987	15.72	108.20
0.005	0.080	478.90	0.685	16.56	243.46
0.0035	0.050	490.75	0.541	16.85	496.60

Table 4.2.: Results for the interpolation of $\exp^{(n)}$ with method A. r_z is the sampling resolution used for choosing the values to be precomputed. t denotes the runtime of computing function values for one million test points and M gives the required memory for the respective setting. All GPU computations have been conducted on a Nvidia Quadro K2200. The used CPU is a Intel Xeon E3-1226 v3 @ 3.30 GHz.

		CPU		GPU		
r_z	r_n	e_{rel} [10^{-3}]	t [ms]	e_{rel} [10^{-3}]	t [ms]	M [MB]
0.1	0.1	9.865	63.83	9.877	6.58	12.46
0.1	0.05	2.487	123.31	2.497	7.59	24.32
0.05	0.1	9.790	168.43	9.799	7.65	49.52
0.05	0.05	2.456	180.06	2.465	8.54	96.70
0.1	0.01	0.116	187.69	0.138	8.00	119.25
0.05	0.025	0.619	190.78	0.628	8.55	191.03
0.075	0.01	0.101	200.08	0.129	8.72	210.69
0.075	0.005	0.003	212.79	0.005	8.12	420.32

Table 4.3.: Results for the interpolation of $\exp^{(n)}$ with method B. r_z and r_n are the sampling resolutions used for choosing the values to be precomputed. All other columns correspond to the respective columns in Table 4.2.



(a) Relative error for $\exp^{(0.75)}$ using 2d-interpolation and $r_z = 0.0035$. Interpolation is expectedly difficult at singularities $0, 1, e, \ldots$ and around the branch-cut. Other areas would benefit from a finer sampling resolution.

(b) Relative error for $\exp^{(0.75)}$ using 3d-interpolation and $r_z = 0.075$ and $r_n = 0.005$. Compared to the plot for 2d-interpolation the grid visible is not distorted due to the direct interpolation.

Figure 4.7.: Relative error for $\exp^{(0.75)}$ using method A and B with the respective highest resolutions.

Method A is more accurate for almost all settings. A similar accuracy can only be reached for the highest sampling resolution in n for method B. The higher impact r_n on the error can also be seen in the fact that settings with the same r_n, but higher sampling resolution r_z give only marginally better results. A direct comparison between the two settings with the respective highest resolutions (see Fig. 4.6) shows that even through their median relative errors lie close to each other, the interpolation with Method B still covers a wider range. However, as less computational effort is needed in method B, it is generally faster than method A, which involves two interpolation operations and the exact computation of c^n.

The differences in accuracy between CPU and GPU on the same sampling resolution arise from the internal use of 9-bit fixed point numbers for the coefficients in texture fetches on GPU, which have lower precision than the 32-bit floating point numbers used

on CPU. Furthermore, processing time increases with higher memory consumption. This is most likely the result of more cache misses deriving from accessing files or textures in a very random manner. Even though an accuracy better than the one seen for single-precision numbers in Section 4.3.2 can be achieved by interpolation, Fig. 4.7 shows that there are some problematic areas for interpolation. As the primary goal of using interpolation is a more efficient computation procedure, Method B will be used in further testing. While this definitely reduces processing time, it remains to be verified if interpolation has a negative impact on learning behavior within ANNs.

5. Experimental Results

In the following, we investigate how ANNs using fractional iterates of the exponential function as transfer functions and therefore interpolation between summation and multiplication perform on different datasets. Alongside standard gradient descent we also apply adaptive optimizers such as Adam (Kingma and Ba, 2015), RMSProp (Tieleman and G. Hinton, 2012) or Adadelta (Zeiler, 2012) which are provided by Bayer (2013) for simple use with Theano.

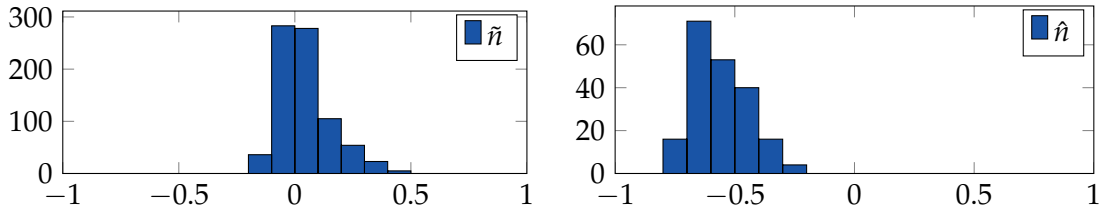
5.1. Recognition of Handwritten Digits

We perform multi-class logistic regression with one hidden layer on the MNIST dataset (LeCun, Bottou, Bengio, and Haffner, 1998). It contains 60 000 images of handwritten digits zero to nine in its training set and 10 000 images in its test set. 10 000 samples of the training set are used as a validation set. Each image has 784 pixels, which form the input to the hidden layer.

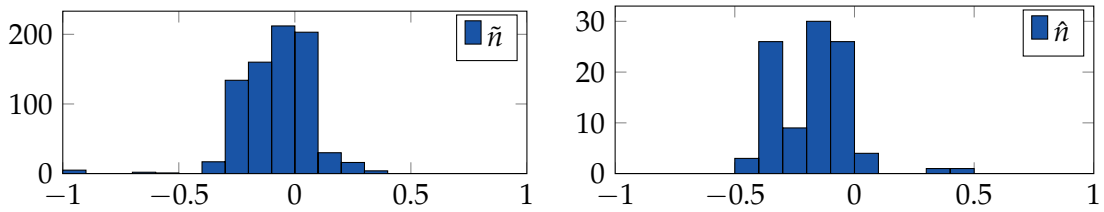
We compare a conventional additive ANN to ANNs using the real- and complex-valued fractional exponential function as follows. In order for all networks to have approximately the same number of trainable parameters, the additive network and the network using the real fractional exponential function use 200 hidden units while the ANN with the complex fractional exponential function has 100 hidden units. The setup resembles the one used for presenting benchmark results for Theano in Bergstra, Breuleux, Bastien, et al. (2010). Additionally, all nets include a sigmoid transfer function $\sigma(x) = 1/(1 - \mathrm{e}^x)$ within the hidden layer. Outputs are thus computed according to (1.1) and (3.25). For the sake of investigation of the impact of interpolating $\exp^{(n)}$, both a network calculating exact values and a network interpolating $\exp^{(n)}$ with presented interpolation method B[1] are tested.

All networks are trained with stochastic gradient descent (SDG) using an initial learning rate of 10^{-1}, a momentum of 0.5, and a minibatch size of 60 samples. The learning rate

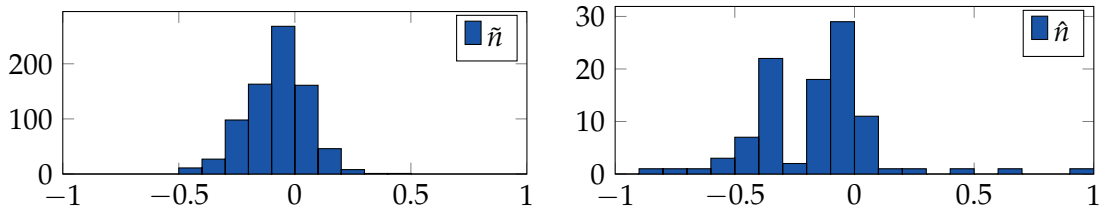
[1] Interpolation in three dimensions for real- and imaginary part of z and parameter n. Here, we use the highest resolution tested, where $r_z = 0.075$ and 0.005 (see Section 4.4).



(a) Histograms of for the ANN using the real fractional exponential function ($\mathbb{R}$-exp$^{(n)}$).



(b) Histograms for the ANN using the complex fractional exponential function ($\mathbb{C}$-exp$^{(n)}$).



(c) Histograms for the ANN using the complex fractional exponential function with interpolation method B ($\mathbb{C}$-exp$_B^{(n)}$).

Figure 5.1.: Histograms (each with 20 bins) for parameters $\hat{n}$ and $\tilde{n}$ within the hidden layer of an ANN using the real- and complex-valued fractional exponential function and multi-class logistic regression on the MNIST dataset.

ANN type	n_{pars}	n_d / s	i	error [%]
additive	159 010	32387	283	2.03
$\mathbb{R}$-exp$^{(n)}$	159 994	21253	245	2.53
$\mathbb{C}$-exp$_B^{(n)}$	159 894	17147	263	2.67
$\mathbb{C}$-exp$^{(n)}$	159 894	707	402	2.81

Table 5.1.: Results for using ANNs of different type for classification of handwritten digits. $\mathbb{R}$-exp$^{(n)}$ and $\mathbb{C}$-exp$^{(n)}$ stand for nets with real- and complex fractional exponential function. Subscript B indicates that interpolation method B has been used. i is the iteration at which the best validation loss has been reached. The error denotes how many digits within the test set have been wrongly classified at that iteration. Furthermore, the number of parameters n_{par} and number of data samples processed per second n_d / s are given.

is reduced whenever the validation loss does not improve for 100 consecutive iterations over the whole training set. Initially, all weights are set to zero with exception of the real-part of the weights within the hidden layer. These are initialized by randomly sampling from the uniform distribution over the interval

$$\left[-4\sqrt{\frac{6}{n_{in}+n_{out}}}, 4\sqrt{\frac{6}{n_{in}+n_{out}}}\right], \tag{5.1}$$

following a suggestion from Glorot and Bengio (2010) for weight initialization in ANNs with a sigmoid transfer function. Here, n_{in} and n_{out} denote the number of input and output neurons for the hidden layer.

Table 5.1 shows the results for training the four networks until convergence. Both structures using interpolation between summation and multiplication achieve similar though not surpassing performance on the MNIST dataset. Furthermore, there is a difference in performance between the two complex-valued nets. The net using interpolation is numerically more stable when large gradients cause $\hat{n}$ and $\tilde{n}$ to become large. Values outside $[-1, 1]$ are mapped back into this interval due to the mirroring addressing mode used in interpolation. Fig. 5.1 displays histograms for parameters $\hat{n}$ and $\tilde{n}$ for each of the tested ANN architectures at the iteration with the best validation loss. While many of the parameters remain close to initialization value zero, some have changed notably and thus the ANNs with our proposed transfer function performs operations beyond pure additive interactions. It is also clearly visible just from the histograms that the versions with and without interpolation reach significantly different solutions.

optimizer	mom	test loss $[10^{-6}]$	i	σ
Adam	0.0	1.76	828	linear
Adam	0.5	2.53	610	linear
Adam	0.99	2.54	728	linear
SGD	0.99	2.95	981	tanh
RMSProp	0.99	3.19	735	linear

Table 5.2.: The best performing models for an ANN learning a multivariate polynomial with the real fractional exponential function. i denotes the iteration at which the test loss given in the next column has been reached. Note that σ_i is the additional non-linear transfer function in the first layer. The second layer uses no additional non-linear transfer function.

5.2. Multinomial Regression

With the motivation to analyze the behavior of parameter n in ANNs that use interpolation between summation and multiplication, we examine a synthetic dataset that exhibits multiplicative interactions between inputs. The function to be learned is a randomly generated multivariate polynomial or multinomial

$$f(x, y) = a_{00} + a_{10}\, x + a_{01}\, y + a_{11}\, xy + a_{20}\, x^2 + a_{02}\, y^2 \tag{5.2}$$

of second degree with two inputs x and y. This function can be computed by a two-layer ANN with six hidden units, where the first layer performs multiplication with the weight-matrix containing the respective powers for each summand and the second layer uses addition with coefficients a_{ij} as weights.

We generate a dataset with a total of 10 000 samples by first randomly sampling the six coefficients a_{ij} uniformly from the interval $[0, 1]$. Then, input values x, y are sampled from the same interval and the respective output is computed according to (5.2). From those 10 000 samples, 8 000 randomly chosen samples are used as the training set. The remaining samples are split into a validation set and a test set of equal size.

Using a network with the same structure as needed for an exact solution, meaning two layers and six hidden units, we perform a hyper-parameter search to find a combination of parameters where function $f(x, y)$ is approximated well for both a net using the real and complex fractional exponential function. Tested combinations include different optimizers (SDG, Adam, RMSProp and Adadelta), momentums, and setting of an additional non-linear transfer function σ (linear, sigmoid, tanh) in the first layer. Since

optimizer	mom	test loss $[10^{-6}]$	i	σ
Adam	0.99	0.99	732	tanh
Adam	0.0	1.08	601	linear
RMSProp	0.99	1.10	200	linear
Adam	0.5	1.13	396	linear
RMSProp	0.99	1.16	325	linear

Table 5.3.: The best performing models for an ANN learning a multivariate polynomial with the complex fractional exponential function. All other columns correspond to the respective columns in Table 5.2.

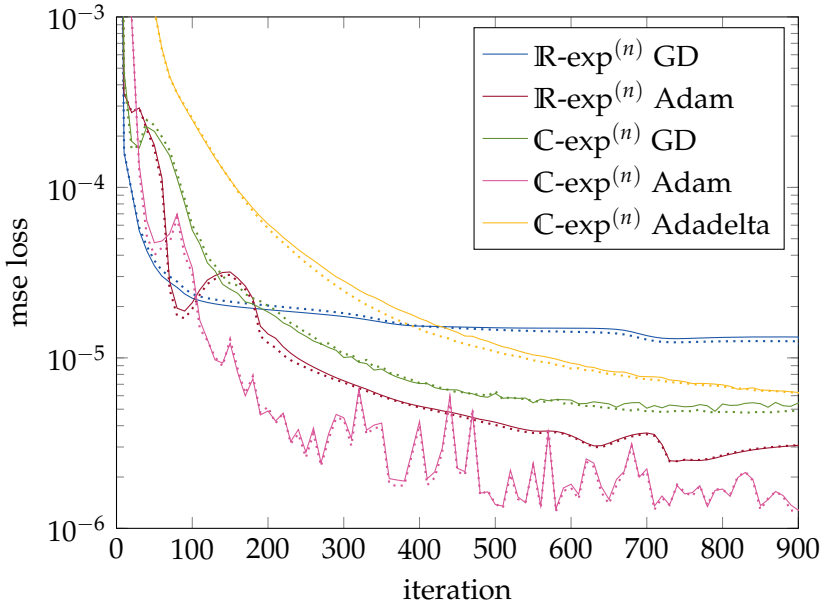
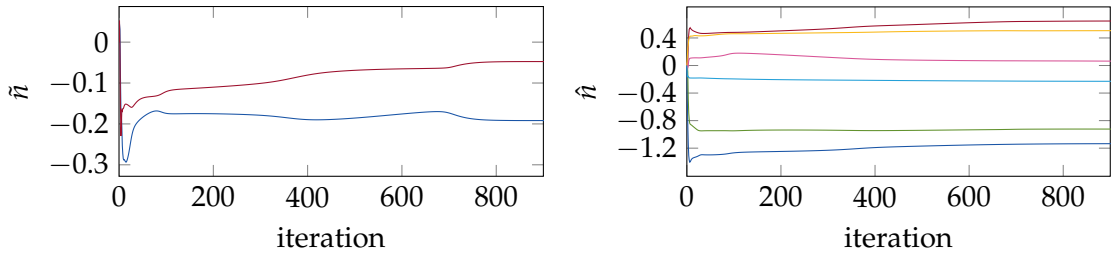
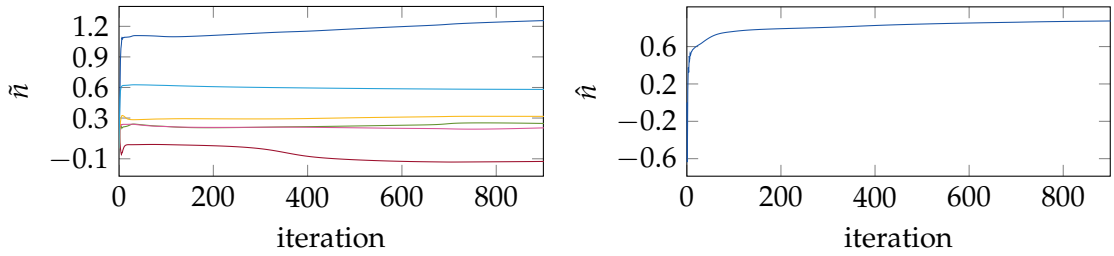


Figure 5.2.: Test (—) and training loss (····) for training a two-layer ANN for selected optimizers learning a multivariate second-degree polynomial of two variables.



(a) Parameters in the first layer.



(b) Parameters in the second layer.

Figure 5.3.: Progression of parameter n in an ANN learning a multivariate polynomial. The parameters change most in the very beginning and than stabilize quickly.

the learning rate is again reduced upon no further improvement, different learning rates are not part of the search. The initial learning rate is set to 10^{-2} and all hyper-parameter combinations use a batchsize of 100 samples.

Table 5.3 shows the settings for the five best performing models for a net using the real and complex fractional exponential function. Although gradient descent performs reasonably well for a small ANN, adaptive optimization methods generally yield better results. Fig. 5.2 additionally displays the progression of test and training loss for selected optimizers. Progression of parameter n in case of the best model with gradient descent is shown in Fig. 5.3

5.2.1. Generalization in Multinomial Regression

Closer investigation of the multivariate polynomial used in the previous section reveals that it can be learned even by an additive net with the same number of hidden units. The employed coefficients yield an almost bilinear function. Additionally, with 8 000 training samples in the two-dimensional unit box, it is likely that for each test point

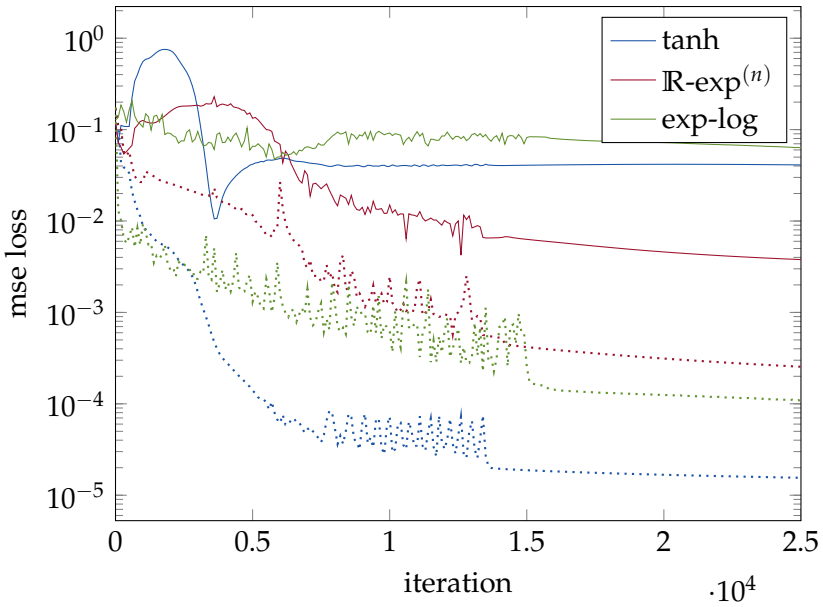
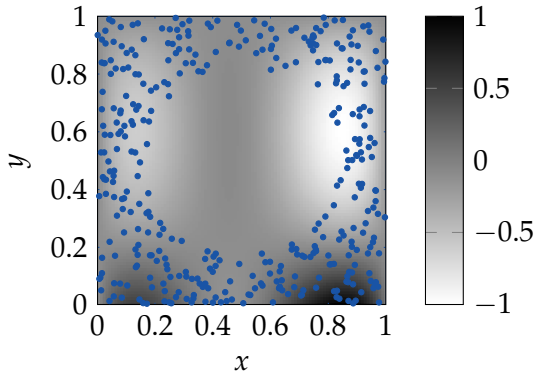


Figure 5.4.: Test (—) and training loss (····) loss for approximation of a multivariate polynomial for different net structures.

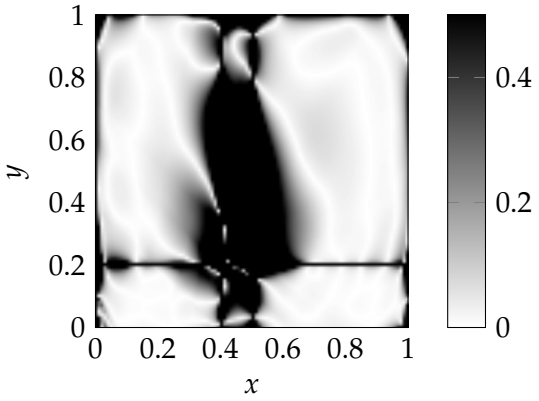
there exists a training sample close to it. Therefore, in order to analyze long-range generalization performance, we use a more complex polynomial of fourth degree with two inputs and split test and training set in a non-random way. Input values are sampled from the interval $[0, 1]$. However, all points within a circle around $(0.5, 0.5)$ are used as the test set and all points outside that circle make up the training set.

We train three different three-layer net structures with the Adam-optimizer, where the final layer is additive with no additional transfer function in all cases. The first neural network is purely additive with tanh as transfer function for the first two layers. The second network uses the real fractional exponential function in the first two layers. The third and final network uses a multiplicative first layer, i.e. (1.2) is used as the transfer function. The progression of training and test loss for all three structures is displayed in Fig. 5.4.

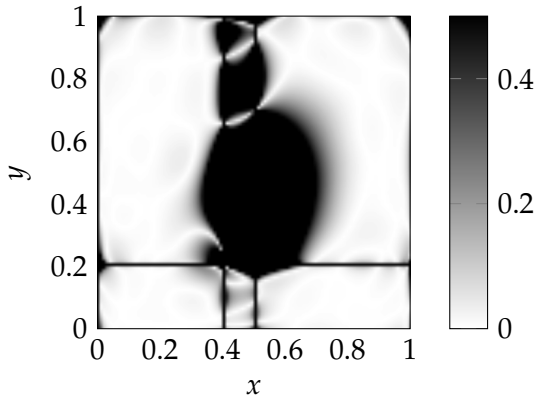
Here, our proposed transfer function generalizes best. This can also be seen when reviewing the relative error of the approximation (see Fig. 5.5) computed by comparing function value and approximation for points on a regular grid. Parts of the circle, where the network has no training points are approximated well. It remains to be examined why the network containing the multiplicative layer, thus resembling the model that actually generated the data, performs worst.



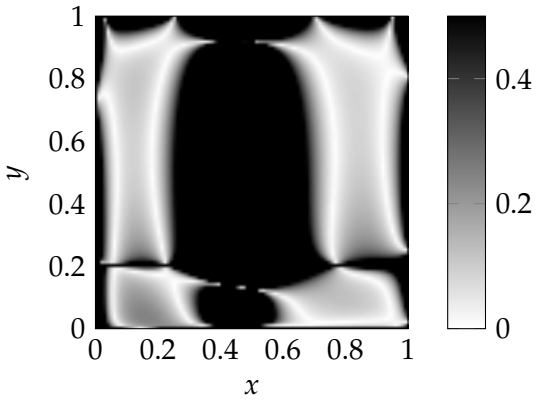
(a) Data to be learned. Training points are colored in blue.



(b) Relative error for approximation for ANN with $\mathbb{R}$-exp$^{(n)}$ as the used transfer function.



(c) Relative error for approximation for ANN with tanh as the used transfer function.



(d) Relative error for approximation for ANN with exp-log as the used transfer function.

Figure 5.5.: Relative error for approximation of a multivariate polynomial for the three tested ANN architectures.

5.3. Double Pendulum Regression

Consider a double pendulum as shown in Fig. 5.6. Its kinematic state is determined by the angles θ_1, θ_2 and the corresponding angular momentums, in this case parameterized by the corresponding angular velocities $w_1 = d\theta_1/dt$, $w_2 = d\theta_2/dt$. Using Lagrangian mechanics the angular accelerations $a_1 = dw_1/dt$, $a_2 = dw_2/dt$ with only gravity acting as an external force on the system are determined to

$$\begin{aligned} a_1 = [&m_2 l_1 w_1^2 \sin(\delta_\theta)\cos(\delta_\theta) + \\ &m_2 g \sin(\theta_2)\cos(\delta_\theta) + \\ &m_2 l_2 w_2^2 \sin(\delta_\theta) - \\ &(m_1 + m_2) g \sin(\theta_1)] \;/\; D_1 \end{aligned} \tag{5.3a}$$

$$\begin{aligned} a_2 = [&-m_2 l_2 w_2^2 \sin(\delta_\theta)\cos(\delta_\theta) + \\ &(m_1 + m_2) g \sin(\theta_1)\cos(\delta_\theta) - \\ &(m_1 + m_2) l_1 w_1^2 \sin(\delta_\theta) - \\ &(m_1 + m_2) g \sin(\theta_2)] \;/\; D_2 \end{aligned} \tag{5.3b}$$

with

$$\begin{aligned} \delta_\theta &= \theta_2 - \theta_1 \\ D_1 &= (m_1 + m_2) l_1 - m_2 l_1 \cos(\delta_\theta)\cos(\delta_\theta) \\ D_2 &= (m_1 + m_2) l_2 - m_2 l_2 \cos(\delta_\theta)\cos(\delta_\theta) \end{aligned} \tag{5.3c}$$

where $g = 9.81$ is standard gravity, l_1 and l_2 are the lengths of the pendulums and m_1, m_2 are their respective masses. Given an initial state $\theta_1(t = 0)$, $\theta_2(t = 0)$, $w_1(t = 0)$, $w_2(t = 0)$ the trajectory of the double pendulum can be obtained by numerical integration of (5.3a) and (5.3b) (Levien and Tan, 1993).

We train an ANN to perform regression of the accelerations a_1, a_2 given the state of the double pendulum. We sampled 1800 states from a uniform distribution; 1600 samples were used for training and the rest as an independent test set. The employed ANN has three hidden layers using the real fractional exponential and a sigmoid as an additional non-linear transfer function.

Training was performed using the following procedure. After random weight initialization using a zero-mean normal distribution, standard loss minimization training using the Adam-optimizer was performed while keeping $\tilde{n}$ and $\hat{n}$ constant at zero for a fixed number of iterations. Then, the constraint on $\tilde{n}$ and $\hat{n}$ was lifted and training was continued.

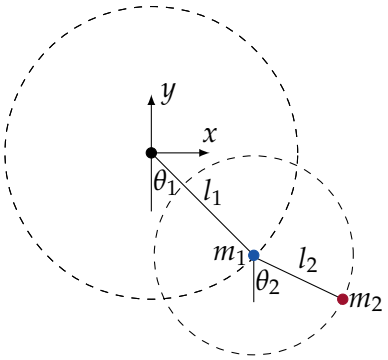


Figure 5.6.: Double pendulum where l_1 and l_2 are the lengths of the pendulums, m_1, m_2 are their respective masses, and θ_1, θ_2 corresponds to the corresponding angular momentums.

	net structure	training loss	test loss
1.	$\exp^{(n)}$ with free n after 200 000 iterations	0.134	0.497
2.	$\exp^{(n)}$ with $n = 0$	0.117	0.549
3.	sigmoid	0.105	0.361

Table 5.4.: Results for learning accelerations a_1, a_2 of a double pendulum with different net structures after 2 000 000 iterations.

For comparison a copy of the neural net without lifting the constraints and a neural network using the standard sigmoid transfer function were also trained for the same number of iterations.

While the first network with free n outperforms the second network in generalization performance on the test set, both are exceeded by the standard sigmoid network (see Table 5.4). Since $\exp^{(n)}$ is the identity function for $n = 0$, networks 2 and 3 should have equal performance; nevertheless the bounds on ψ^{-1} in Algorithm 2 limit the identity function to the interval $[-10, 10]$ as well. This poses an implementation trade-off that seems to have a negative effect on performance for at least this regression problem.

6. Conclusion and Future Work

The objective of this thesis was to efficiently implement and evaluate a novel transfer function enabling smooth interpolation between summation and multiplication with fractional iterates of the exponential function for the use in ANNs.

While real-valued fractional iterates of the exponential function can be efficiently evaluated with exact computations, the complex-valued version requires an approximation method for training ANNs in reasonable time. We have tested bi- and trilinear interpolation of $\exp^{(n)}$. With similar results in terms of accuracy, interpolation in three dimensions is significantly faster than interpolation for two dimensions and therefore has been used in subsequent experiments. However, this method requires almost 500 MB of memory on the GPU and can only be used on GPUs with sufficient space designated as texture memory. Future work therefore includes the examination of other approximation methods. One possibility is the use of interpolation for a certain number of iterations of the exponential function thus avoiding both inaccuracy and efficiency issues for a high number of iterations. Better memory utilization in interpolation of χ^{-1} could possibly be achieved by exploiting the fact that the range of $c^n\chi$ covers roughly the shape of an ellipse with an irregular interpolation grid.

Experiments on different regression datasets have revealed that usage of the proposed transfer function achieves comparable or sometimes slightly better results compared to conventional additive ANNs. Overall, adaptive optimizers have been shown to be superior to regular stochastic gradient descent in learning the operation of individual neurons. Specifically, Adam-optimization method outperforms other optimizers for classification of handwritten digits and approximation of multinomials. Many of our observations thus far lack an explanation, suggesting further directions for future work. This includes the investigation of learning data with negative inputs, especially with real fractional iterates of the exponential function. In this scope, the choice of cut-off parameters ψ_{min} and ψ_{max} (see Algorithm 2) has to be reevaluated. Furthermore, we noticed that parameter n sometimes grows beyond the interval of $[-1, 1]$. It should be tested if this behavior can be remedied with by either adding regularization for parameter n to the loss currently used or restricting the change of n during learning through capping the update step.

Finally, the usage of fractional functional iterates could be investigated for other functions used in ANNs. We believe that with a continuously differentiable solution for either Abel's or Schröder's functional equation, e.g. $f(z) = \tanh(z)$, an ANN could learn the order of non-linearity for a neuron.

In conclusion, we think that while further investigation of the proposed transfer function is necessary to issue a definitive judgement on its usefulness, this method has the potential to advance the field of deep learning and therefore to contribute to future steps towards more sophisticated machine learning and general-purpose artificial intelligence, which in turn will have a significant impact (Rudin and Wagstaff, 2013) on both science and society.

List of Figures

List of Tables

Bibliography

Agostinelli, F., M. Hoffman, P. Sadowski, and P. Baldi (2014). "Learning Activation Functions to Improve Deep Neural Networks." In: eprint: arXiv:1412.6830[cs.NE].

Bastien, F., P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio (2012). "Theano: New Features and Speed Improvements." In: eprint: arXiv:1211.5590[cs.SC].

Baydin, A. G., B. A. Pearlmutter, and A. A. Radul (2015). "Automatic Differentiation in Machine Learning: A Survey." In: eprint: arXiv:1502.05767[cs.SC].

Bayer, J. (2013). *climin: optimization, straight-forward*. URL: http://climin.readthedocs.org/en/latest/ (visited on 12/14/2015).

Bergstra, J., O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio (2010). "Theano: a CPU and GPU Math Expression Compiler." In: *Proceedings of the 9th Python for Scientific Computing Conference (SciPy)*. Ed. by S. van der Walt and J. Millman, pp. 3–10.

Bishop, C. M. (2013). *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer.

Cheney, E. W. and D. R. Kincaid (2012). *Numerical Mathematics and Computing*. 7th ed. Brooks Cole.

Droniou, A. and O. Sigaud (2013). "Gated Autoencoders with Tied Input Weights." In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013*. Vol. 28. JMLR Proceedings, pp. 154–162.

Durbin, R. and D. E. Rumelhart (1989). "Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks." In: *Neural Computation* 1.1, pp. 133–142.

Engelbrecht, A. P. and A. Ismail (1999). "Training Product Unit Neural Networks." In: *Stability and Control: Theory and Applications* 2, pp. 59–74.

Freitag, E. and R. Busam (2009). *Complex Analysis*. 2nd ed. Universitext. Springer.

Frigge, M., D. C. Hoaglin, and B. Iglewicz (1989). "Some Implementations of the Boxplot." In: *The American Statistician* 43.1, pp. 50–54.

Gabbiani, F., H. G. Krapp, N. Hatsopoulos, C.-H. Mo, C. Koch, and G. Laurent (2004). "Multiplication and Stimulus Invariance in a Looming-Sensitive Neuron." In: *Journal of Physiology-Paris* 98.1, pp. 19–34.

Glorot, X. and Y. Bengio (2010). "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In: *International Conference on Artificial Intelligence and Statistics*, pp. 249–256.

Griewank, A. (2003). "A Mathematical View of Automatic Differentiation." In: *Acta Numerica* 12, pp. 321–398.

He, K., X. Zhang, S. Ren, and J. Sun (2015). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In: *The IEEE International Conference on Computer Vision (ICCV)*.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics.

Hirose, A. (2012). *Complex-Valued Neural Networks*. 2nd ed. Studies in Computational Intelligence 400. Springer.

Hornik, K., M. B. Stinchcombe, and H. White (1989). "Multilayer Feedforward Networks are Universal Approximators." In: *Neural Networks* 2.5, pp. 359–366.

Janson, D. J. and J. F. Frenzel (1993). "Training Product Unit Neural Networks with Genetic Algorithms." In: *IEEE Expert* 8, pp. 26–33.

Kingma, D. and J. Ba (2015). "Adam: A Method for Stochastic Optimization." In: eprint: `arXiv:1412.6980[cs.LG]`.

Kneser, H. (1950). "Reelle analytische Lösungen der Gleichung ... und verwandter Funktionalgleichungen." In: *Journal für die reine und angewandte Mathematik* 187, pp. 56–67.

Kuczma, M., B. Choczewski, and R. Ger (1990). *Iterative Functional Equations*. Encyclopedia of Mathematics and its Applications. Cambridge University Press.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998). "Gradient-Based Learning Applied to Document Recognition." In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Levien, R. and S. Tan (1993). "Double Pendulum: An Experiment in Chaos." In: *American Journal of Physics* 61, pp. 1038–1038.

Nvidia Cooperation (2015a). *CUDA C Programming Guide v7.5*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 09/01/2015).

– (2015b). *Parallel Thread Execution ISA v4.3*. URL: http://docs.nvidia.com/cuda/parallel-thread-execution/ (visited on 09/01/2015).

Philip R. Bevington, D. K. R. (2003). *Data Reduction and Error Analysis for the Physical Sciences*. 3rd ed. McGraw-Hill.

Rudin, C. and K. L. Wagstaff (2013). "Machine Learning for Science and Society." In: *Machine Learning* 95.1, pp. 1–9.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). "Learning Representations by Back-Propagating Errors." In: *Nature* 6088.323, pp. 533–536.

Schmidhuber, J. (2015). "Deep Learning in Neural Networks: An Overview." In: *Neural Networks* 61, pp. 85–117.

Schmitt, M. (2002). "On the Complexity of Computing and Learning with Multiplicative Neural Networks." In: *Neural Computation* 14.2, pp. 241–301.

Schnupp, J. W. H. and A. J. King (2001). "Neural Processing: The Logic of Multiplication in Single Neurons." In: *Current Biology* 11.16, R640–R642.

Sutskever, I., J. Martens, and G. E. Hinton (2011). "Generating Text with Recurrent Neural Networks." In: *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*. Ed. by L. Getoor and T. Scheffer. Omnipress, pp. 1017–1024.

Tang, P. T. P. (1991). "Table-Lookup Algorithms for Elementary Functions and their Error Analysis." In: *IEEE Symposium on Computer Arithmetic*, pp. 232–236.

Tieleman, T. and G. Hinton (2012). *Lecture 6.5 - rmsprop: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning.

Turner, A. J. and J. F. Miller (2014). "NeuroEvolution: Evolving Heterogeneous Artificial Neural Networks." In: *Evolutionary Intelligence* 7.3, pp. 135–154.

Urban, S. and P. van der Smagt (2015). "A Neural Transfer Function for a Smooth and Differentiable Transition Between Additive and Multiplicative Interactions." In: eprint: arXiv:1503.05724[stat.ML].

Wegert, E. (2012). *Visual Complex Functions: An Introduction with Phase Portraits*. Springer.

Yao, X. (1999). "Evolving Artificial Neural Networks." In: *Proceedings of the IEEE* 87.9, pp. 1423–1447.

Zeiler, M. D. (2012). "ADADELTA: An Adaptive Learning Rate Method." In: eprint: arXiv:1212.5701[cs.LG].

A. Plot Techniques

A.1. Domain Coloring

Domain coloring refers to the visualization of complex-valued functions (Wegert, 2012, Ch. 2.5) based on the HSB-color model. HSB stands for *hue*, *saturation* and *brightness* and each value can vary between 0 and 1. The hue is used to represent the phase ϕ of $z = re^{i\phi}$, the saturation encodes the magnitude r and for brightness real and imaginary part are combined. The exact computation for the latter two involves taking the sin of either z or its real or imaginary part, leading to reparative patterns.

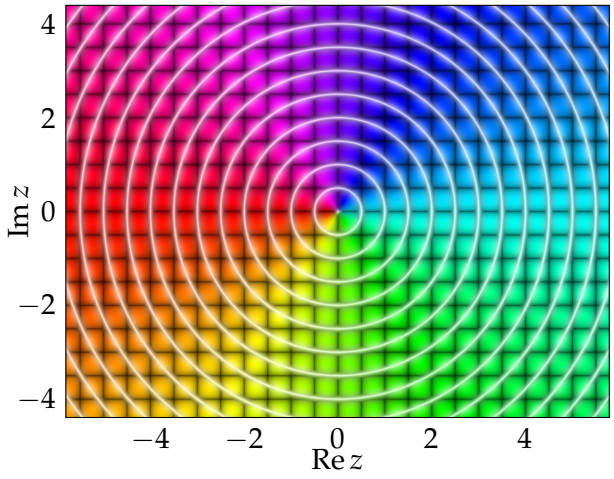
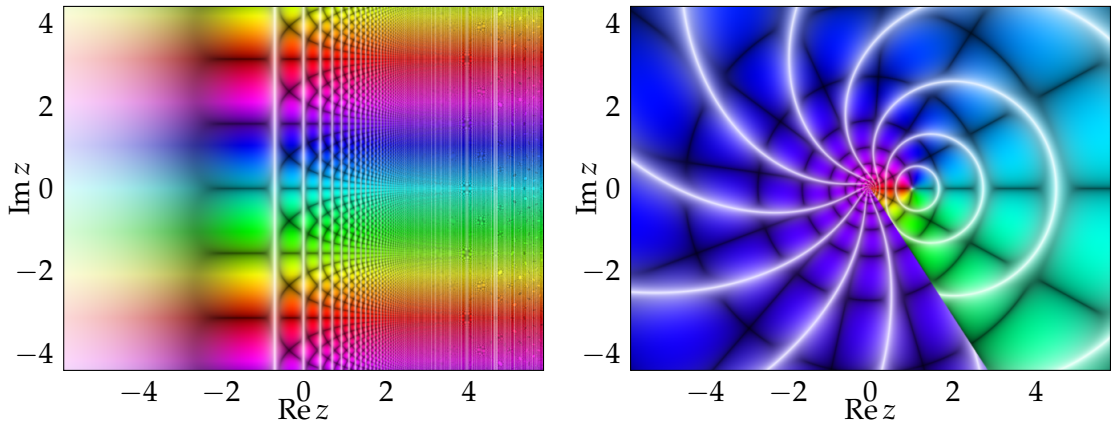


Figure A.1.: Domain coloring plot of $f(z) = z$.

This becomes more clear when looking at the domain plot for the identity function $f(z) = z$ (see Fig. A.1). One can see the color depending on the angle as well as circles of varying saturation around the origin. The grid-like structure originates from the brightness being computed in terms of real and imaginary part as well as total magnitude. More examples are displayed in Fig. A.2.



(a) Domain coloring plot of $f(z) = \exp(z)$.

(b) Domain coloring plot of $f(z) = \log(z)$ with $\log : \mathbb{C} \to \{z \in \mathbb{C} : -1 \leq \operatorname{Im} z \leq -1 + 2\pi\}$.

Figure A.2.: More examples for domain coloring plots.

A.2. Box-and-Whisker Plot

Box-and-whisker plots (Frigge, Hoaglin, and Iglewicz, 1989) are used to display some statistical measures of one-dimensional data, such as the relative error in Fig. 4.5. Among those statistical measures are the median, the first quartile Q_1, and the third quartile Q_3 (see Fig. A.3). Quartiles mark the boundary of the box and are the result of partitioning the data into four sets of equal size. The whiskers are computed with the interquartile range (IQR).

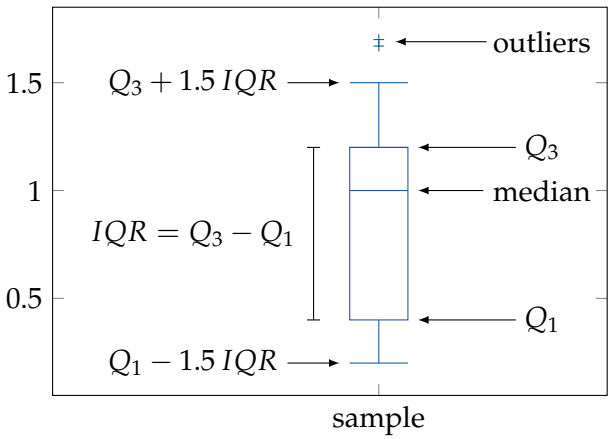


Figure A.3.: An examplary box-and-whisker plot.