Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Deep neural networks with multiplication layers

Diploma thesis

2023

Bc. Slavomír Holenda

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Deep neural networks with multiplication layers

Diploma thesis

Study Programme: Aplikovaná Informatika
Field of Study: 2511 aplikovaná Informatika
Department: Katedra aplikovanej informatiky
Supervisor: RNDr. Kristína Malinovská, PhD.
Consultant: Mgr. Ľudovít Malinovský

Bratislava, 2023
Bc. Slavomír Holenda

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Slavomír Holenda
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Deep neural networks with multiplication layers
Hlboké neurónové siete s multiplikatívnymi vrstvami

Anotácia: Nová neurónová sieť s produktovými neurónmi QuasiNet, ktorá sa učí metódou gradientového zostupu, elegantne rieši problém vzájomne vylučujúcich sa situácii, ako je napríklad problémy XOR a parita a má veľký potenciál v oblasti neurónových sietí. Pre pokračovanie existujúceho výskumu je potrebné urobiť systematické experimenty s hlbokou verziou tejto neurónovej siete a dôsledne overiť úspešnosť tohto nového modelu pre ťažké úlohy ako problém dvoch špirál, dvoch banánov a separáciu šachovnice. Ďalej je potrebné preskúmať možnosti tohto modelu pre použitie v populárnej paradigme transfer learning v hlbokom učení pre klasifikačné úlohy, kde môže tento model nahradiť viacvrstvový perceptrón na posledných vrstvách hlbokej neurónovej siete.

Cieľ: 1. Naštudovať teóriu o hlbokých neurónových sieťach.
2. Implementovať hlbokú verziu modelu NS s multiplikatívnymi vrstvami.
3. Implementovať a vyhodnotiť experimenty s touto NS.
4. Implementovať multiplikatívnu hlbokú sieť pre klasifikáciu pomocou transfer learning, implementovať a vyhodnotiť experimenty s vybraným benchmark datasetom.

Literatúra:

[1] Goodfellow, I., Bengio, Y. and Courville, A., (2016). Deep learning. MIT press.
[2] Ghosh, J. & Shin, Y. (1995). Efficient Higher-order Neural Networks for Classification and Function Approximation. In: International Journal of Neural Systems. 3. 10.1142/S0129065792000255.
[3] Malinovský, Ľ., Malinovská, K. (2022): Neurónová sieť s násobiacou vrstvou. In Šejnová, G., Vavrečka, M., Hvorecký, J. (eds.), Kognice a umělý život XX, České vysoké učení technické v Praze. 79-83.

Vedúci: RNDr. Kristína Malinovská, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 30.11.2022

Dátum schválenia: 01.12.2022

prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

Thanks: podakovanie

Abstrakt

TODOTODO

Kľúčové slová: TODOTODO

Abstract

TODOTODO

Keywords: TODODO

Contents

0.1 Overview and methods . . . 3
  0.1.1 Neural Networks . . . 3
  0.1.2 Basic model - MLP . . . 3
  0.1.3 Adaptation of weights by gradient descent method . . . 4
  0.1.4 Activation functions . . . 5
  0.1.5 CNN . . . 6
  0.1.6 Pooling . . . 7
  0.1.7 Deep learning . . . 8
  0.1.8 Transfer Learning . . . 8
  0.1.9 Multiplication in neurons . . . 9
0.2 State of the art . . . 11
0.3 Aim of the work/Motivation . . . 11
0.4 QuasiNet . . . 11
  0.4.1 Fully learning model . . . 12
  0.4.2 Activation . . . 12
  0.4.3 Learning . . . 13
0.5 Methods of research . . . 13
  0.5.1 Convergence . . . 13
  0.5.2 Comparison . . . 13
0.6 Research . . . 14
  0.6.1 XOR and parity problems with hidden 1 layer . . . 14
  0.6.2 2 hidden layers on parity-7 . . . 16
  0.6.3 Multiple layers and 2 spirals . . . 17
0.7 Discussion . . . 18

List of Figures

1 Complete bipartite graph 3
2 MLP visualisation 4
3 Graph of sigmoid (left) and sigmoid derivation (right). Taken from [5] . 5
4 Graph of tanh (left) and graph of tahn derivation(right). Taken from [5]. 6
5 An example of 2-D convolution from Goodfellow-et-al-2016 7
6 An example of Max Pooling 7
7 Transfer learning scheme 8
8 Sigma pi architecture 9
9 Architecture of our model with 1 hidden layer adapted from [2]. 11
10 Results from XOR experiments with varying hidden layer size (max. 500 epochs). Number of converged nets (left) and average number of training epochs (right, non-converging runs included) 15
11 Results from parity 8 experiments with varying hidden layer size (100 nets, max. 5000 epochs). Convergence (left) and average training epochs to converg (right, including non-converging runs). 16
12 Results from n-parity experiments: minimum size of the hidden layer **h** for maximum number of converging networks. 16
13 The 2 spirals dataset (2000 pts) split into 80% training and 20% testing data. 18
14 Results from training of the 2 spirals problem: mean and standard deviation over 10 nets trained for 10000 epochs, only every 10th epoch is shown. 18

List of Tables

1 Results from n-parity experiments: minimum size of the hidden layer **h** for maximum number of converging networks. For MLP baseline we report results we have achieved given by the computational limits, very large hidden layer size would lead to a slightly better performance, but not full convergence. 15
2 Results from 7-parity on networks with 2 hidden layers. T = hyperbolic tangent. S = sigmoid function. Q = our (Quasi) product layer. [-,-] means that no nets converged to a solution. 17

Introduction

TODO [1] [2]

0.1 Overview and methods

0.1.1 Neural Networks

Very first idea of artificial neural networks is based on imitation of real biological neurons. They are able to learn, sometimes incomprehensible, correlations from given data and then generalize over new unseen inputs. The networks can be made of arbitrary number of layers or neurons. Artificial neurons, sum up the values from the neurons from previous layer, which are multiplied by weight. Additionally, the activation function is evaluated over the sum. This is done, because more complex problems are not linearly separable, so the non-linear functions has to be used. In fully connected neural networks, we have weight for every pair of neurons from two subsequent layers. The learning ofthe network is done by tweaking these weights in specific manner.

0.1.2 Basic model - MLP

In 1986, Rumelhart, Hinton, and Williams published a new perspective on artificial neurons. They were not considered as logic switches, but rather as analog elements that contained a continuous input-output function [3].

The Multi-Layer Perceptron (MLP) is capable of learning from examples and then generalizing new inputs. It can consist of any number of hidden layers, which can have different numbers of neurons. We compute the values for each layer similarly to a simple perceptron, but we must always start at the input layer and continue iteratively to get to the result.

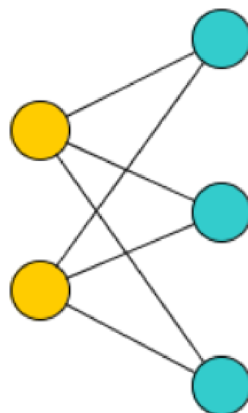

Figure 1: Complete bipartite graph

So far, we have only defined weights for a perceptron that contains only one neuron on the output layer. If our network consisted of n input neurons and m output neurons,

the weights could be visualized using a bipartite graph where the sets are the individual layers. In practice, we will represent them by a matrix of size $m.n$.

For learning the weights, a back-propagation rule was introduced to efficiently adjust the weights, called *error back-propagation* (often referred to as *backprop* for short) [3]. The algorithm is not yet sufficient to learn to solve an arbitrary problem, but the model can already handle many problems that cannot be solved by linear separation. However, this method has limitations. It is only applicable to perceptrons that have three or more layers. Also, these neurons cannot be connected "each to each", but only forward to the next layer.

Let us define an MLP with n dimensional input and one hidden layer which has m neurons represented by the vector $h = (h_1, h_2, ..., h_m)$, an output layer which has 1 neuron. The weights W, consist of a matrix w^{hid} between the input and hidden layers and a vector w^{out} between the hidden and output layers. We will also add bias at the input layer. We consider that the neurons on the hidden layer have a certain activation function f_{hid} and the neurons on the output layer have a certain activation function f_{out}. Then :

$$h_i = f_{hid}(\sum_{j=1}^{n} w_{ij}^{hid} \cdot x_j) \tag{1}$$

$$y = f_{out}(\sum_{i=1}^{m} w_i^{out} \cdot h_i) \tag{2}$$

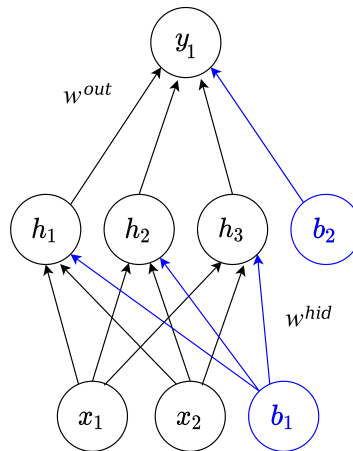

Figure 2: MLP visualisation

0.1.3 Adaptation of weights by gradient descent method

Weight adaptation, or network learning, is the search for weights such that there is the smallest possible difference between the desired value and the value computed by our network for a particular input for any training dataset [4].

Consider that $A_{train} = (x^1, d^1), (x^2, d^2), ..., (x^p, d^p), ..., (x^P, d^P)$ where x^p is the input vector and d^p is the desired output. Let us define an error function for MLP:

$$E = \frac{1}{2} \sum_{i=0}^{P} (d^p - g(w.x^p))^2, \tag{3}$$

where g is the activation function.

Next, we continue with stochastic gradient descent (SGD). This method adjusts the weights so that the error E reaches its minimum.

0.1.4 Activation functions

Sigmoid

Sigmoid function displays any real number on the interval $(0, 1)$. It is used in models whose output is a probability estimate, or also in binary classification. It is S-shaped, so it does not contain any jumps or drastic changes in values. Its drawback is that it is not symmetric with respect to 0, so its derivative takes very small values outside the interval $(-3, 3)$, which makes it difficult to train the network and makes it unstable. This phenomenon is called the "vanishing gradient problem" [5] .

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

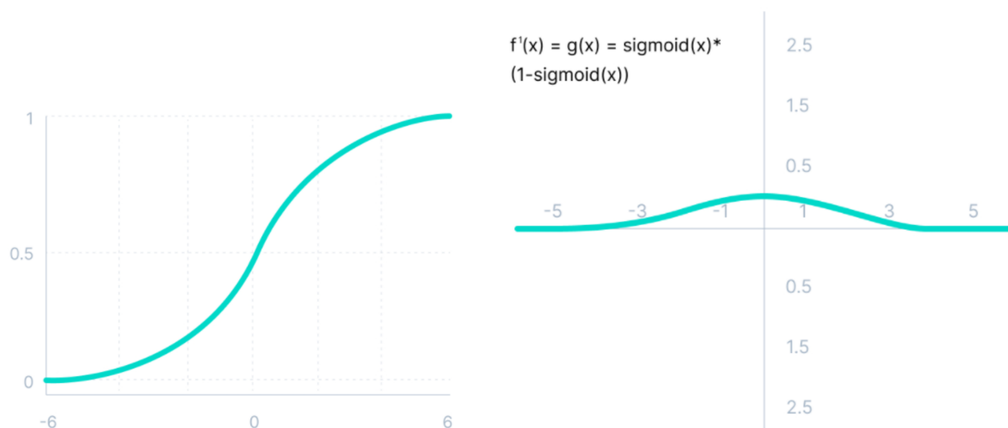

Figure 3: Graph of sigmoid (left) and sigmoid derivation (right). Taken from [5]

Tanh

Hyperbolic tangent (Tanh) - is a function very similar to sigmoid. The difference is that its range of values is on the interval $(-1, 1)$, and it is symmetric under 0. We can use it to tell whether the result is negative, neutral or positive. It is usually used on hidden neurons because it maps the data around 0, making it easier to learn at the next layer. The downside is that, like sigmoid, it suffers from the vanishing gradient problem [5].

$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \tag{5}$$

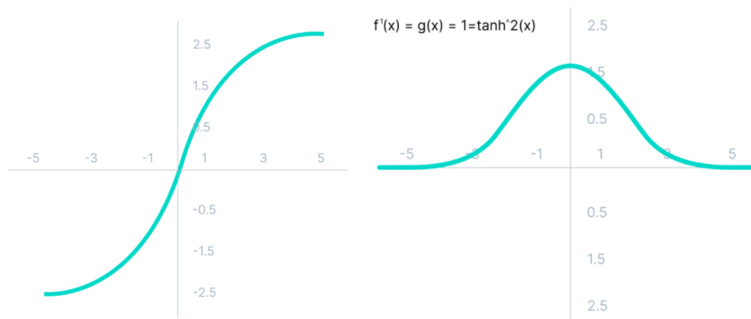

Figure 4: Graph of tanh (left) and graph of tahn derivation(right). Taken from [5].

0.1.5 CNN

Convolutional neural networks (CNNs) are a specific kind of neural network designed for processing data with a known grid-like topology. This includes time-series data, which can be considered as a 1-D grid with regular time intervals as samples, and image data, which can be viewed as a 2-D grid of pixels. CNNs have proven highly effective in practical applications. The term “convolutional neural network” suggests that the network utilizes a mathematical operation known as convolution, which is a unique form of linear operation. In at least one of its layers, convolutional networks are just neural networks that substitute general matrix multiplication with convolution. This mathematical operation is performed on two real-valued functions. The first function, known as the input, will undergo convolution with the second function, referred to as the kernel/feature detector/filter. To enhance clarity, we will refer to these functions as the input and kernel throughout this document. The output is commonly known as the feature map. Convolution employs three essential concepts that can enhance a machine learning system: sparse interactions, parameter sharing, and equivariant representations. Furthermore, convolution offers a method of operating with inputs of varying dimensions.

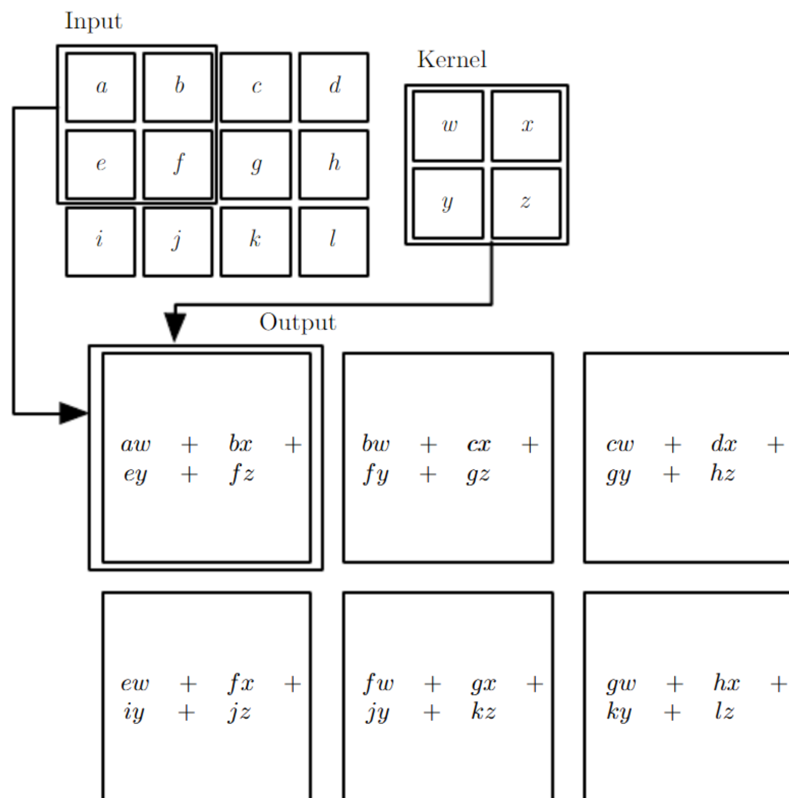

Figure 5: An example of 2-D convolution from Goodfellow-et-al-2016

0.1.6 Pooling

Pooling aswell as convolution operates with sliding a two-dimensional filter, but instead of convolution returns 1 value for the whole window. There are several types of pooling. For example max pooling return the maximum value from the filter. Pooling layers are used to reduce the dimensions, which reduces the number of parameters to be learned, so we need less computational power. Also, it summarizes the features found produced by a convolution layer. This step is crucial, because we can do further operations on the summarized features rather at once. By this the model becomes more resilient to variations in feature position within the input image.

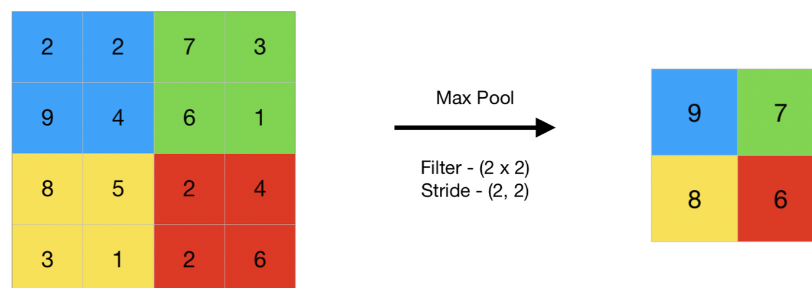

Figure 6: An example of Max Pooling

0.1.7 Deep learning

Deep learning are powerful model architectures for supervised learning. "Deep" means that architecture network has multiple hidden layers in the architecture. By adding layers and units to a network, it can represent increasingly complex functions. Deep learning models can recognize complex patterns in pictures, text, sounds, and other data to produce accurate insights and predictions. In deep learning, every layer learns features that are more and more composite and abstract. For example, if we want to use deep neural network for facial recognition, the first input would be a matrix of pixels, the first hidden layer would represent the pixels and encode the edges, the second layer would encode positioning of edges. In the third hidden layer would encode the nose and eyes, and the fourth layer would be fourth layer recognising that the image represents a face.

0.1.8 Transfer Learning

Usually, we need a lot of data to train a neural network from scratch but often it isn't available or it is too expensive. Transfer learning is the process where we reususe pre-existing model with already trained weights to solve a new problem. By this approach instead of starting the learning process from scratch, we reuse the whole model or tweak the weights in some layers, and by utilizing the gained knowledge from related task, we enhance generalization in another. For example if we have model already trained to recognize dishes, it could be applied when we try to recognize mugs. We cut the training time, need of large quantity of data while enhancing the performance.

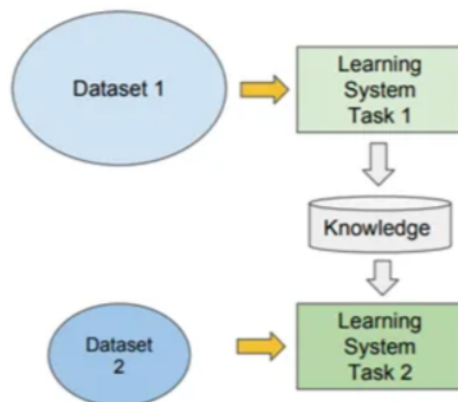

Figure 7: Transfer learning scheme

0.1.9 Multiplication in neurons

Although MLP can approximate nonlinear functions, the time required to train the model is much larger compared to a single-layer linear perceptron. Solution could occur in multiplication. Currently, multiplication is largely theoretically explored, but in actual practice it does not occur that often. In the past, multiplication has been popular so-called higher-order neurons [6]. For the most part, this model resembles the traditional perceptron, it can be said to be an extension of it. The main difference is in the input neurons. Instead of the traditional vector, which contained the input values or bias, the the vector is expanded with additional terms, which are calculated by multiplying and amplifying the original inputs. Each of these inputs was given its own weight. The model, however, without any constraints, it obtains exponential values on the polynomial scale.

Sigma-pi

One special model that is theoretically similar to higher-order neurons, are the so-called sigma-pi neurons. This model uses a weighted sum to calculate the output of the products of the inputs as well as the back propagation of errors. A suitable constraint on the maximum polynomial degree as well as selecting the right members, these networks are able to more quickly and more efficiently generalize and learn, compared to classical type of networks. A limitation of this approach consists in the fact that the performance of the model depended largely on the engineer himself [7].

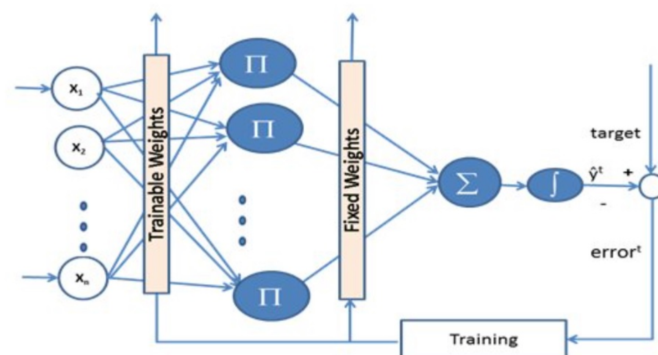

Figure 8: Sigma pi architecture

Generalisation attempt

Attempts to solve this problem have been proposed by models that learn the importance of individual polynomial terms [8, 9]. Models of this type started on a polynomial of minimum order, to which terms were gradually added. Throughout the process, the

current efficiency was evaluated, based on which the model progressed further. The type of these models were subsequently generalized by Durbin and colleagues [10]. In their model, the classical weights are not used to multiply individual inputs, as in the classical perceptron, but as their exponents, which are subsequently adjusted. However, in the calculations we arrive at complex numbers. A proposed solution to this problem is to neglect the imaginary part and use only the real part. Although this type of networks are effective in solving parity and symmetry problems, their application contains yet further complications that have not been explored.

Pi-sigma

A different perspective on the multiplication problem in NS was brought by Ghosh and Shin [2]. In contrast to previous models, they bring the nonlinearity up to the output layer. Their main task is to achieve an efficient mapping similar to a higher-order network, but with the proviso that they are not computationally intensive. This category of models can be described as feedforward neural networks, which first perform a linear combination over the inputs, which they multiply without without using any activation function. Only at the output layer are they then used non-linear activation function. Since this type of models uses the product of sums, the exact opposite of sigma-pi, which use the sum of products, are called inverse, or pi-sigma. Their result can be analyzed using polynomials. Learning is achieved by modifying weights on the linear layer, using the gradient method. The weights of the multiplicative layer are not learned and are set to a fixed value of 1. As the models mentioned above, this type of models also achieves a significant increase in the success rate on a variety of problems, as well as the aforementioned parity compared to MLP. Fast learning is guaranteed by the fact that it takes place on only one layer, but this principle has its rawbacks. By not using an activation function on the linear layer, this model essentially collapses into a simpler polynomial that has pre-selected terms. There is also the disadvantage that the authors did not attempt to find a way of learning weights on the multiplicative layer, which could lead to even greater efficiency of the overall model [2].

Multiplication in biological neurons

Multiplication is also found in biological neural networks. Our body uses multiplication, for example, in sensations such as vision [11] or hearing [12]. Several experts claim, that if synapses are too close together, the individual stimuli serve as individual agents product. Conversely, synapses that are not so close together use the sum excitations, where excitations play the role of classical adders [13]. B. Mel argues also the occurrence of splitting in dendritic plexuses [14], as well as other operations that could represent sigma-pi models [15].

0.2 State of the art

Testing of several types of multiplicative models was done by M. Schmitt, in which he also derived constraints for each type [16]. The result of his research suggests that multiplication is a suitable method that guarantees an increase in computational power without drastically increase the nonlinear interaction as in other methods. As such, multiplication has also found application in deep neural networks, where it is used in a limited at selected locations. [17, 18, 19, 20]

0.3 Aim of the work/Motivation

Our model is inspired by pi-sigma networks. As with pi-sigma networks, multiplication is performed at the output layer and the model contains only one hidden layer, thus preserving learning speed. However, it takes the essence of these models one step further by including, instead of linear or sigmoidal neurons on the hidden layer, the nowadays more widely used neurons that have a hyperbolic tangent as an activation function. As described in section **??**, this function determines values on the neuron in the range $(-1, 1)$. The aim of the work is to test this new network on various datasets. These will include problems as Parity, 2 spirals and more convetional as backpack dataset, CIFAR10, CIFAR100,(ImageNet if gpus). We will also test transfer learning, where we will swap traditional some fully connected layers with Quasi in architectures like VGG.

0.4 QuasiNet

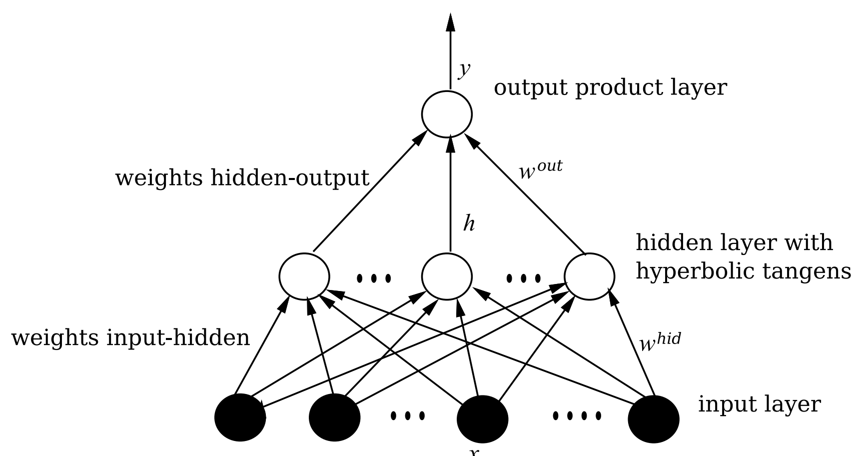

Figure 9: Architecture of our model with 1 hidden layer adapted from [2].

0.4.1 Fully learning model

The newer model shares a similar structure with the prototype. However, it is different from it in the approach to multiplicative weights as well as the activation function. Learning at the multiplicative layer is implemented by replacement of exponentiation by a polynomial. [21]

The weights no longer act as exponents, but rather as indicators of the extent to which the factor to which they belong will be used. Also, the compressed weights can be understood as a percentage of the importance of a given value in multiplication. This substitution removes the occurrence of complex numbers, and also allows the model to adjust the importance of individual hidden neurons in calculating the desired output by itself.

0.4.2 Activation

We compute the values of individual neurons on the hidden layer as for the classical perceptron, using the activation function tahn.

$$h_i = \tanh(\sum_j w_{ij}^{\text{hid}} \cdot x_j) \tag{6}$$

We compress the exponent weights on the resulting layer using sigmoids. The weights themselves can take values from the real number range, but after the function is applied, they will be in the range $(0, 1)$.

For activation we use the following polynomial function:

$$f(h, \sigma(w)) = 1 - \sigma(w)(1 - h) \tag{7}$$

$$h^1 = f(h, \sigma(w)) = h \tag{8}$$

$$h^1 = f(h, \sigma(w)) = h \tag{9}$$

This approach preserves the original properties of the exponentiation, but is continuous and continuously derivable for all zero values, so we can use gradient methods without any restrictions or appearance of complex numbers. In the extremes, the function behaves as follows

$$y_i = \prod_j \left(1 - \sigma(w_{ij}^{\text{out}})(1 - h_j)\right) \tag{10}$$

0.4.3 Learning

Due to the properties described in the sections above, there is no need for constraining or further limiting the model. We can implement model learning using the gradient descent method as follows:

$$\frac{\partial E}{\partial w_{ij}^{\mathrm{n}}} = (d_i - y_i)\left(\prod_{k \neq j} 1 - \sigma(w_{ik}^{\mathrm{n}})(1 - h_k)\right) \\ (h_j - 1)\sigma(w_{ij}^{\mathrm{n}})(1 - \sigma(w_{ij}^{\mathrm{n}})) \tag{11}$$

spätné šírenie chyby na predošlú vrstvu

$$\frac{\partial E}{\partial h_i} = \sum_k (d_k - y_k)\left(\prod_{l \neq i} 1 - \sigma(w_{kl}^{\mathrm{n}})(1 - h_l)\right)\sigma(w_{ki}^{\mathrm{n}}) \tag{12}$$

The multiplicative layer is not constrained by its position, so it can be arbitrarily employed into network architectures. It is possible to use multiple multiplicative layers in a row or to alternate between classical and summation layers.

0.5 Methods of research

0.5.1 Convergence

Using convergence, we can measure the effectiveness of the model, whether it can learn the problem at all, or how many epochs on average it takes to do so. The algorithm for measuring this is to create n models with the same parameters chosen by us, which we then learn on the same data. When one instance of the network learns a given problem, we say it converges. The main result in the case of XOR and parity is to calculate what percentage of network instances were able to converge. We will be able to determine the computational power of the model against a given problem based on the percentage of convergence. In Python, we created a function that receives as parameters the type of network, its settings, the value of how many networks to test and the input data. After creating and training a given number of networks, the function returns the number of networks that managed to converge, the ending epoch number of each network, the time as the computation test ran. We will use this function in the experiments.

0.5.2 Comparison

Comparison of models is possible based on their convergence. In order to be able to compare the two models objectively, we will try to have them as similarly implemented as possible. We will evaluate them on the same problem, in most cases with the same parameters. The model with the higher percentage of convergence is more efficient.

We will also compare the average number of terminal epochs of the networks. We also count failed networks in the average; their terminal epoch count will be the maximum set by us. In datasets other than parity it will be hard to train bigger models on our hardware. We will investigate and compare their performance on chosen datasets from other research papers.

0.6 Research

0.6.1 XOR and parity problems with hidden 1 layer

The parity problem can be even or odd, we will consider the even variant. The problem is that we have an N-bit string to which we assign one bit. We determine this by whether the number of ones in the string is even but not so that their final count is even even even with the assigned bit. Using convergence, we measure the effectiveness of the model, whether it can learn a given problem at all or how many epochs it needs on average to do so. The algorithm for measuring this is to build n models with the same parameters chosen by us, which are then learned on the same data. When one instance of the network learns a given problem and predict all inputs right for set number of consecutive epochs, then we say it converges. Simply put we measure how many networks out of 100 reach a stable solution. For the baseline comparison, we used a basic multilayer perceptron (MLP) without any regularization. For both types of networks, we use a uniform learning rate $\alpha = 0.9$ and a Gaussian initialization of the weights with distribution $\mathcal{N}(0; 1.0)$. These hyperparameters were chosen based on previous experimentation.

From Fig. 10 we can see the superior performance of our model over all degrees of parity. Also, we could not find a neural network model, where 100% of nets would converge on XOR problem with only 2 neurons in hidden layer. In Table 1 we can see detailed result for parity 2-7. The table does not include results for parity of higher degree, because we could not find such hyperparameters, where MLP converged under contrained number of epochs. In parity-7 we even increased the number of epochs to 10 thousand, and enlarged the size of hidden layer significantly, but we still achieved lower convergence. Quasi net in the best case converges in less then 100 epochs.

Next we wanted to test the optimal size of hidden layer in architecture with Quasi layer. We chose the parity of number 8 and set number of max epochs to 5000. Then gradually increasing the number of hidden neurons, we observed that larger size does not lead to better performance. It seems that the optimal size for Quasi architecture is close to the number of input features.

Figure 12 shows performance of Quasi nets on parity of $2 \leq n \leq 13$ degree. For parity of higher degree we changed the initial weight distribution to $\mathcal{N}(0; 0.5)$ and

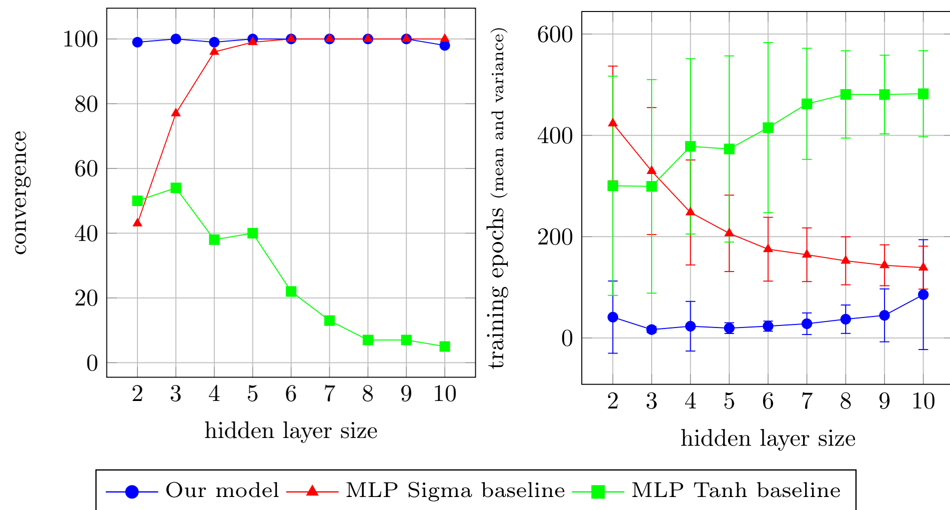

Figure 10: Results from XOR experiments with varying hidden layer size (max. 500 epochs). Number of converged nets (left) and average number of training epochs (right, non-converging runs included)

increased number of epochs to 1 thousand.

	QuasiNet		**MLP baseline**	
n-parity	**h**	convergence	**h**	convergence
2	2	100	4	100
3	4	100	9	100
4	6	100	12	91
5	7	100	50	44
6	12	100	45	69
7	15	100	45	33

Table 1: Results from n-parity experiments: minimum size of the hidden layer **h** for maximum number of converging networks. For MLP baseline we report results we have achieved given by the computational limits, very large hidden layer size would lead to a slightly better performance, but not full convergence.

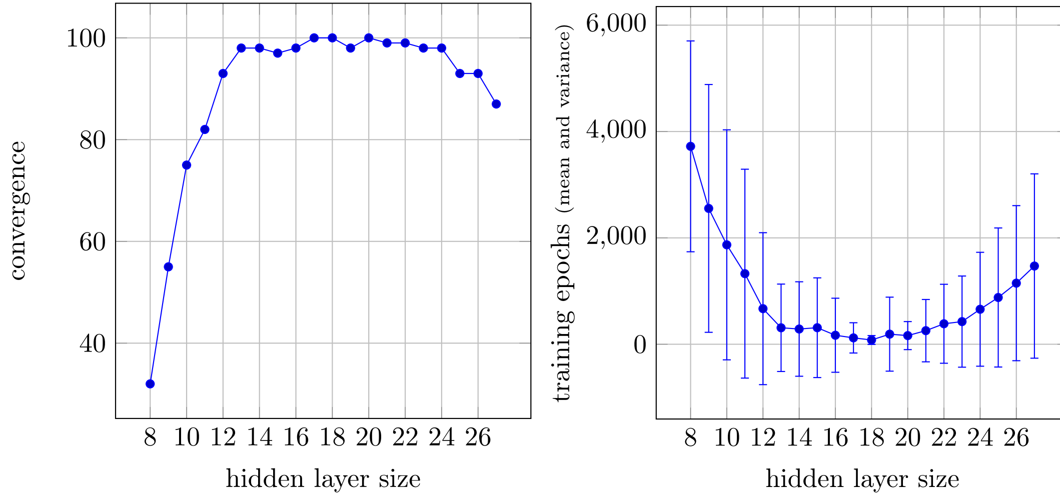

Figure 11: Results from parity 8 experiments with varying hidden layer size (100 nets, max. 5000 epochs). Convergence (left) and average training epochs to converg (right, including non-converging runs).

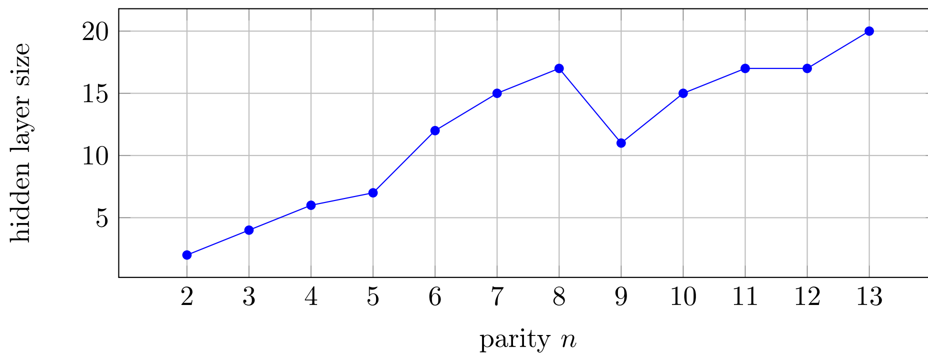

Figure 12: Results from n-parity experiments: minimum size of the hidden layer **h** for maximum number of converging networks.

0.6.2 2 hidden layers on parity-7

In this experiment we investigated the performance of Quasi layers in different architectures. We also tried net architectures with only classical summation layers for comparison. We wanted the dataset to be a little bit harder to learn while we still want the nets to converge in reasonable time, so we chose the number of parity to be 7. To save the time, architectures were tested only on 20 instances. Alse we constrained the number of epochs to 350 for Quasi architectures and 1000 for MLP architectures. After increasing the number of epochs to 5000, we noticed that if the instances did not converg after reaching previous limits(350 and 1000), also did not converge until reaching 5000.

In 2 we can see that result for every possible architectures made from Quasi layer

Act. f.	h	conv.	epochs
T, Q, T	[15, 5]	18	100.7
T, T, Q	[20,20]	11	332.5
Q, T, T	[5,10]	14	148.5
Q, Q, T	[10,15]	20	82.5
Q, Q, T	[5,10]	19	14
Q, T, Q	[5,5]	20	22.15
T, Q, Q	[-,-]	0	-
Q, Q, Q	[2,2]	20	20.55
S, S, S	[100,100]	20	710
T, T, T	[-,-]	0	-

Table 2: Results from 7-parity on networks with 2 hidden layers. T = hyperbolic tangent. S = sigmoid function. Q = our (Quasi) product layer. [-,-] means that no nets converged to a solution.

and Tanh layer. We can see that the performance between architectures vary. Some could converge under

0.6.3 Multiple layers and 2 spirals

The famous 2 spiral problem is commonly used for testing new neural network models. Similarly to parity it poses a problem with mutually exclusive situations, i.e the point belongs to one spiral or to another. We display the 2 spirals dataset with 2 thousand points we have used and its distribution into training and testing data in Fig. 13. Note, that the spiral coordinates are transformed to the interval $(-1, 1)$ similarly to inputs in parity problem. In our preliminary experiments we have observed that this problem requires more hidden layers for a satisfying performance level.

In Fig 14 we display training progress of QuasiNet with four hidden layers of neurons, where the hyperbolic tangent and product layers are combined one after another. Namely, the architecture used was: 2 input neurons, 10 tanh neurons, 80 product neurons, 5 tanh neurons and finally 1 output product neuron. Other hyperparameters used were learning rate $\alpha = 0.01$ and a Gaussian initialization of the weights with distribution $\mathcal{N}(0; 0.5)$. The networks were trained for 10 thousand epochs. The mean

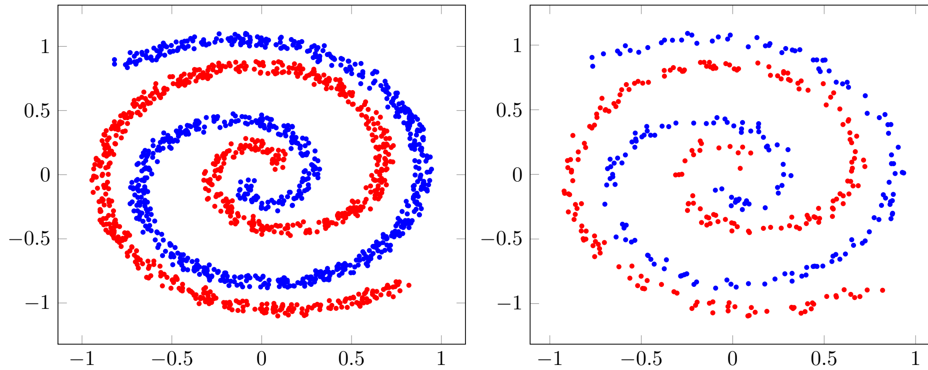

Figure 13: The 2 spirals dataset (2000 pts) split into 80% training and 20% testing data.

accuracy for the testing data set was 98.275% and 4 out of 10 networks achieved full convergence, which we deem very successful.

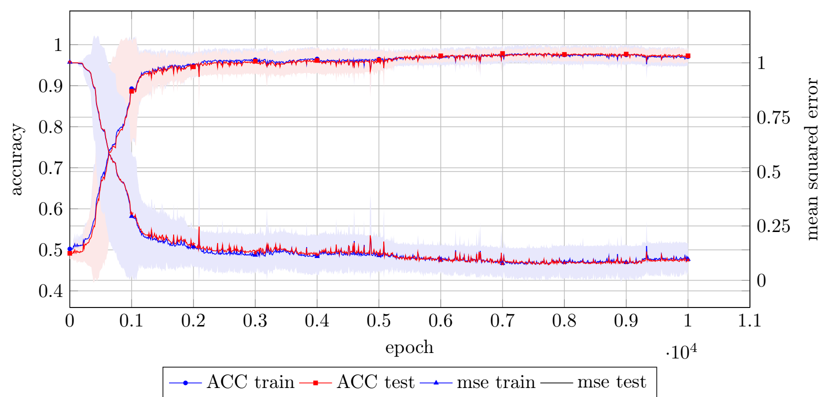

Figure 14: Results from training of the 2 spirals problem: mean and standard deviation over 10 nets trained for 10000 epochs, only every 10th epoch is shown.

0.7 Discussion

Bibliography

[1] Malinovský, Ľ., Malinovská, K. "Neurónová sieť s násobiacou vrstvou". In: *Kognice a umělý život XX*. Ed. by Hvorecký J. Šejnová G. Vavrečka M. 2022, pp. 79–83.

[2] Joydeep Ghosh and Yoan Shin. "Efficient higher-order neural networks for classification and function approximation". In: *International Journal of Neural Systems* 3.04 (1992), pp. 323–350.

[3] David E Rumelhart, Geoffrey E Hinton, and James L McClelland. "A general framework for parallel distributed processing". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Vol. 1. Cambridge, MA: MIT Press, 1986, p. 26.

[4] V. Kvasnička et al. *Úvod do teórie neurónových sietí.* Iris: Bratislava, 1997.

[5] Pragati Baheti. *12 Types of Neural Network Activation Functions: How to Choose?* `https://www.v7labs.com/blog/neural-networks-activation-functions`. [Online; accessed 25-March-2022].

[6] Nils J Nilsson. *Learning machines.* McGrawHill New York, 1965.

[7] C Lee Giles and Tom Maxwell. "Learning, invariance, and generalization in high-order neural networks". In: *Applied Optics* 26.23 (1987), pp. 4972–4978.

[8] Malcolm Heywood and Peter Noakes. "A framework for improved training of Sigma-Pi networks". In: *IEEE transactions on Neural Networks* 6.4 (1995), pp. 893–903.

[9] Nicholas J Redding, Adam Kowalczyk, and Tom Downs. "Constructive higher-order network that is polynomial time". In: *Neural Networks* 6.7 (1993), pp. 997–1010.

[10] Richard Durbin and David E Rumelhart. "Product units: A computationally powerful and biologically plausible extension to backpropagation networks". In: *Neural Computation* 1.1 (1989), pp. 133–142.

[11] Richard A Andersen, Greg K Essick, and Ralph M Siegel. "Encoding of spatial location by posterior parietal neurons". In: *Science* 230.4724 (1985), pp. 456–458.

[12] N Suga, JF Olsen, and JA Butman. “Specialized subsystems for processing biologically important complex sounds: Cross-correlation analysis for ranging in the bat’s brain”. In: *Cold Spring Harbor symposia on quantitative biology*. Vol. 55. Cold Spring Harbor Laboratory Press. 1990, pp. 585–597.

[13] Guido Bugmann. “Summation and multiplication: two distinct operation domains of leaky integrate-and-fire neurons”. In: *Network: Computation in Neural Systems* 2.4 (1991), pp. 489–509.

[14] Bartlett W Mel. “Information processing in dendritic trees”. In: *Neural Computation* 6.6 (1994), pp. 1031–1085.

[15] Bartlett Mel and Christof Koch. “Sigma-pi learning: On radial basis functions and cortical associative learning”. In: *Advances in Neural Information Processing Systems* 2 (1989).

[16] Michael Schmitt. “On the complexity of computing and learning with multiplicative neural networks”. In: *Neural Computation* 14.2 (2002), pp. 241–301.

[17] Dumitru Erhan et al. “Visualizing higher-layer features of a deep network”. In: *University of Montreal* 1341.3 (2009), p. 1.

[18] Jianqing Zhu et al. “Joint feature and similarity deep learning for vehicle re-identification”. In: *IEEE Access* 6 (2018), pp. 43724–43731.

[19] Ali Diba, Vivek Sharma, and Luc Van Gool. “Deep temporal linear encoding networks”. In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2017, pp. 2329–2338.

[20] Connor Schenck and Dieter Fox. “Spnets: Differentiable fluid dynamics for deep neural networks”. In: *Conference on Robot Learning*. PMLR. 2018, pp. 317–335.

[21] K. Malinovská and Ľ Malinovský. “Neuronová sieť s násobiacou vrstvou”. In: *Kognice a umělý život XX*. Ed. by Vařečka et al. Praha: České vysoké učení technické v Praze., 2022, pp. 79–83.