Neural Networks and Analog Circuits: An Overview

Christoph Schröter

Abstract

probleme: viele implementierungsmöglichkeiten für analoge nns -> verschiedene vor/nachteile -> vorstellen und vergleichen

- vorteile nachteile neuronale netze analog digital
- implementierungen die neuronale netze analog verwenden finden und beschreiben - vergleichen, metriken für auswahl für implementierung finden

1 Introduction

Neuronal networks (NNs) are becoming increasingly relevant for industry and research. Their power stems from being able to approximate an arbitrary function from just input and output values through training and backpropagation. Therefore they are heavily used already today, for example in image recognition which can be used in medical applications or for autonomous systems such as automotives or roboters. Even in manufacturing they can be used productively, let it be for product design or quality inspection.

In recent research, analog NNs are occurring more and more frequently, as further improvement in gemeral purpose processors slows down, while the demand for powerful NNs increases, slowly forcing research away from the traditional digital ones.

Even though over 30 years ago research has been conducted already in this topic [3, 5, 8, 11], new developments and improvements are still being made with great success. Therefore, this paper presents some recent work put into analog NNs and compares their architectures against each other. As a result, recommendations can be given on which architecture to use based on metrics of a problem or an already developed neural network which should be transferred to the analog design space.

2 Neural Network Structure

Since for implementing a NN whether its digital or analog the structure is crucial, this section provides a brief summary about basic NN components. Because NNs try to recreate the structure of a brain, there are similarities between the two, however, these are not relevant for implementation and therefore not explained in this paper.

2.1 Neuron

The smallest piece of a neural network is a *neuron* (also called *perceptron*), whose structure is shown in Figure 1.

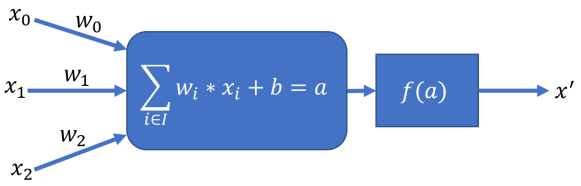

Figure 1. Structure of a neuron. The inputs x_i are multiplied with their corresponding w_i, after that, the bias b is added. Lastly an activation function is applied to determine the output x'.

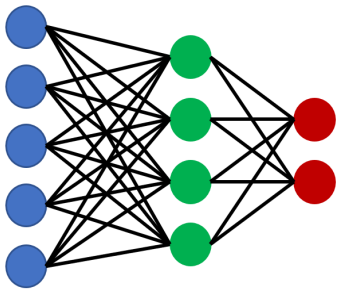

Figure 2. Visualization of a neural network with only fully connected layers. Each input (blue) and output (red) is connected to every neuron in the hidden layer (green).

It sums up an arbitrary number of input values x_i multiplied with their individual weights w_i and adds a bias b to it. The resulting value is called the *activation* a and gets passed to the next neuron (or the output) after applying the *activation function* f. This function plays a huge role in the networks performance, can be selected almost arbitrary and is a research topic on its own. However, simpler activation functions tend to outperform more complex ones, presumably because of a more difficult training process (see section 3). Currently the most widespread function is the *Rectified Linear Unit* which is defined as $ReLU(a) = max(0, a)$. [7]

2.2 Layer

The NN itself consists out of multiple *layers* which in turn consist of the neurons described in subsection 2.1. Figure 2 shows a basic network with three layers, one of which is considered a *hidden layer*, which means that it is neither input nor output. Some common layer types are explained in the following subsections.

2.2.1 Fully-connected Layer. A *fully-connected layer* denotes a layer, where each neuron consumes input from each neuron of the preceding layer. Having a lot of fully connected layers can lead to higher processing time and electrical power requirements due to a high number of trainable parameters, even though all the inputs might not be required in most of the layers [2].

2.2.2 Convolutional Layer. *Convolutional layers* are used to exract features from their input. Therefore, they use a *kernel*, which stands for an array of weights which is smaller than the input. This kernel can then be applied to multiple positions of the input (by "shifting" over it) with a convolution. This leads to a so-called *feature map*. Since the kernel is smaller than the input, convolutional layers are good at detecting for example features of an image (e.g. eyes of a human) regardless of their position. If the output is not supposed to be smaller than the input, *padding* can be used to extend the input, for example by using extrapolation. Typically multiple convolutional layers will be used in parallel to extract multiple features. As this layer type is the core technique, it is name giving for *convolutional neural networks* (CNNs). Layers which perform the inverse of convolutional ones are called *de-convolutional layers*. [4]

2.2.3 Pooling Layer. *Pooling layers* reduce the size of their input by performing a simple pooling operation, such as selecting the maximum (*max pooling*) or average (*average pooling*) value, on different regions of the input. These layers typically follow convolutional ones, as they can be used to extract the existance of a feature in a feature map independent of its position and therefore add robustness for example when rotating or translating an input image. [4]

2.2.4 Normalization Layer. If activation functions such as ReLU (see subsection 2.1) are used in a neural network, inputs into the following layers can take on very different values. Thus, normalizing data inside the network in *normalization layers* can have significant benefits. It allows an easier analysis by the model and speeds up training, as the layers after the normalization one can adapt to a single distribution of inputs. [2]

2.3 Classification

As already explained in subsubsection 2.2.2, CNNs with their convolutional layers are highly performant when extracting features is required. This makes them especially applicable in topics such as computer vision, image classification, video processing or speech recognition [1]. *Deep neural networks* stand for NNs with many hidden layers, while *recurrent neural networks* (sometimes also called *feed-backward NNs*) correspond to NNs, which unlike the up to this point only discussed *feed-forward NNs* contain feedback connections to previous layers or inbetween them. Having feedback inside the network allows better processing of event sequences or time-based data, thus fitting those NNs better to applications such as language learning or adaptive processes in autonomous systems [6]. As classification of NNs is not the main scope, it needs to be noticed that there are many other types of NNs, which are not further discussed in this paper.

3 Neural Network Training

As an important part of creating a (feed-forward) NN is its training, it is briefly explained in this section. To begin with, sufficient training data needs to be available, in that there must exist a set of inputs to the network with the corresponding correct output for those inputs. Moreover, since only training by feeding data forward through the network does not lead to acceptable results, *backpropagation* still generally is the main method for training NNs. As an example, Wilamoski et al. [10] proposed a method which was way quicker in the training process, but did not yield satisfactory outcomes. The results could be improved by increasing the number of neurons, however, this lead to the network being *overtrained*, which means that the network performed well on data it had trained on, but could not handle new inputs (which were not part of the training) correctly.

3.1 Backpropagation

Since backpropagation itself is just a way to calculate derivatives, there are other applications than NNs. In most cases a simpler version of backpropagation is used for NNs, as proposed by Werbos et al. [9], as it would require a lot of manual mathematics which can not be translated to executable code easily. Moreover, the derivatives would be highly dependent on the application itself and therefore could not be reused.
The general idea is to propagate the test input through the network to receive an output, which is then compared with the correct output of the training case. This comparison is done via a so-called *loss function*. Just like with the activation function (see subsection 2.1), there exist numerous different versions. Once the loss has been determined, its derivative with respect to the parameter in the network (e.g. weight, bias) calculated. For this to work, every activation function, as well as the loss function has to be differentiable, however there are workarounds for non-differentiable functions such as ReLU. With the derivatives availabe, *gradient descent* can take place. The gradient of the function is calculated and scaled by a (predetermined) *step-size*, before it gets subtracted from the parameter corresponding to the specific entry of the gradient. This process can be executed iteratively which leads to minimizing the function, in

the case of NNs the loss function, thereby adjusting the parameters such that the output is closer to the correct output when feeding the network with the same training cases again.

4 Analog Implementations

This section provides an overview of selected analog implementations of NNs in past and recent research. After that advantages and disadvantages of this implementation method can be summarized.

4.1 Overview

4.1.1 Early Research. As stated in section 1 already, research on analog implementations of neural networks has been conducted many years ago. Graf et al. (1989) [3] as one of those early works found some interesting aspects. First, the sum of products in a neuron is easily implemented with Kirchhoff's law computing the sum, while the product can be performed by a simple resistor. Second, interconnected (fully-connected) layers were difficult to create because of the high number of required connections, which would have led to a lot of space on the chip used up just for those layers. This point is not as prevalent today, as circuitry has decreased extremely in physical size since then. Another problem was the precision of the analog circuits, which was not as high back then. Therefore Graf et al. recommended to train the NN digitally before its analog implementation and to use a learning algorithm which can tolerate low precision network connections. Flexibility in terms of reprogrammable interconnections could be achieved by using a storage cell to contain the weight which then is applied to the actual connection element (e.g. resistor), but would use up a lot of space on the chip too. They concluded that designing an analog chip for a NN is a tradeoff between functionality and network size. If for example training should take place on the chip too or high resolution interconnections are required, the network must be smaller than without those features. Another conclusion was that neural networks in analog circuits can compute results way faster than their digital counterparts, but were not as flexible nor precise. An aspect not mentioned by Graf et al. is the electrical power consumption, which presumably was not as important at that time, because networks were not close to being as big as they are today.

Another example of pioneer work in this area is Zurada et al. [11], who picked up on the problems of that time three years later and tried to combine the advantages of digital and analog implementations by adding a digital-to-analog interface to the chip which allowed the parameters and interconnection of the network to be programmable. Moreover, they implemented on-chip unsupervised (Hebbian[1]) learning as they assessed on-chip learning as being essential for most appliations. Even though their paper was a huge step forwars in terms of design of analog NNs, they correctly conclude that future designs will become way more complex as the networks get more complicated. Also, desirable properties for future NN chips are listed, such as being affordable, being reprogrammable and not taking up a lot of space. In contrast to Graf et al. power consumption as an important criteria is listed too.

4.1.2 Recent Research.

5 Conclusion

References

[1] Dulari Bhatt, Chirag Patel, Hardik Talsania, Jigar Patel, Rasmika Vaghela, Sharnil Pandya, Kirit Modi, and Hemant Ghayvat. Cnn variants for computer vision: History, architecture, application, challenges and future scope. *Electronics*, 10 (20), 2021. ISSN 2079-9292. doi: 10.3390/electronics10202470. URL https://www.mdpi.com/2079-9292/10/20/2470.

[2] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead. *IEEE Access*, 8:225134–225180, 2020. doi: 10.1109/ACCESS.2020.3039858.

[3] H.P. Graf and L.D. Jackel. Analog electronic neural network circuits. *IEEE Circuits and Devices Magazine*, 5(4):44–49, 1989. doi: 10.1109/101.29902.

[4] Tianmei Guo, Jiwen Dong, Henjian Li, and Yunxing Gao. Simple convolutional neural network on image classification. In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*, pages 721–724, 2017. doi: 10.1109/ICBDA.2017.8078730.

[5] H. Harrer, J.A. Nossek, and R. Stelzl. An analog implementation of discrete-time cellular neural networks. *IEEE Transactions on Neural Networks*, 3(3):466–476, 1992. doi: 10.1109/72.129419.

[6] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.

[7] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017. URL https://arxiv.org/abs/1710.05941.

[8] E.A. Vittoz. Analog vlsi implementation of neural networks. In *IEEE International Symposium on Circuits and Systems*, pages 2524–2527 vol.4, 1990. doi: 10.1109/ISCAS.1990.112524.

[9] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. doi: 10.1109/5.58337.

[10] Bogdan M. Wilamowski and Hao Yu. Neural network learning without backpropagation. *IEEE Transactions on Neural Networks*, 21(11):1793–1803, 2010. doi: 10.1109/TNN.2010.2073482.

[11] J.M. Zurada. Analog implementation of neural networks. *IEEE Circuits and Devices Magazine*, 8(5):36–41, 1992. doi: 10.1109/101.158511.

[1] https://julien-vitay.net/lecturenotes-neurocomputing/4-neurocomputing/5-Hebbian.html