Going Deeper: Infinite Deep Neural Networks

Benjamin Meier
Zurich University of Applied Sciences, Winterthur, Switzerland
meierbe8@students.zhaw.ch

Abstract

In this paper-like document a meta-layer for neural networks is described. This meta-layer contains an infinite amount of sub-layers, where the network can decide at the training time how many of these layers should be used. The complete network may be optimized with gradient descent based methods. Some simple experiments are conducted to demonstrate how the meta-layer might be used. The required code to reproduce the results is online available. This document is very minimal and does not cover as much content as, e.g., a scientific paper.

I. Introduction

Neural networks are very powerful function approximators [1], especially Recurrent Neural Networks (RNNs) [2] [3] are very powerful. It is possible to show they are turing-complete [4]. The structure of neural networks is relatively static and their depth is in general fixed before the training. RNNs allow deeper architectures (through recurrent connections), but they reuse their weights. Finally, the amount of weights for the network has to be known before the training starts. This is often the limiting factor, because more weights allow in general more powerful function approximations, but before the training starts it is often not that easy to guess how many weights are required. Nevertheless, current neural network architectures solve very complex problem with a high accuracy.

Often it is not that easy to decide how many convolutional layers, fully connected layers etc. should be used. Why not let the learning algorithm decide the amount of layers? Every new layer contains new weights, but adding weights to a neural network is intuitively not differentiable and, therefore, the network could not be trained with gradient descent based methods.

In this paper, a new meta-layer is proposed that contains infinite many sub-layers. The network decides how many of these sub-layers are used. To avoid the allocation of new weights, this meta-layer has infinite many weights, but at every timestep only the weights of the first n sub-layers are used (where n may change after every update of the weights). This allows to use gradient descent based methods to optimize the network.

This meta-layers allows a network to adjust the layer-count. It may decide to use 1 layer or 10 layers. The layer count may first increase and then decrease due to the optimization of the weights.

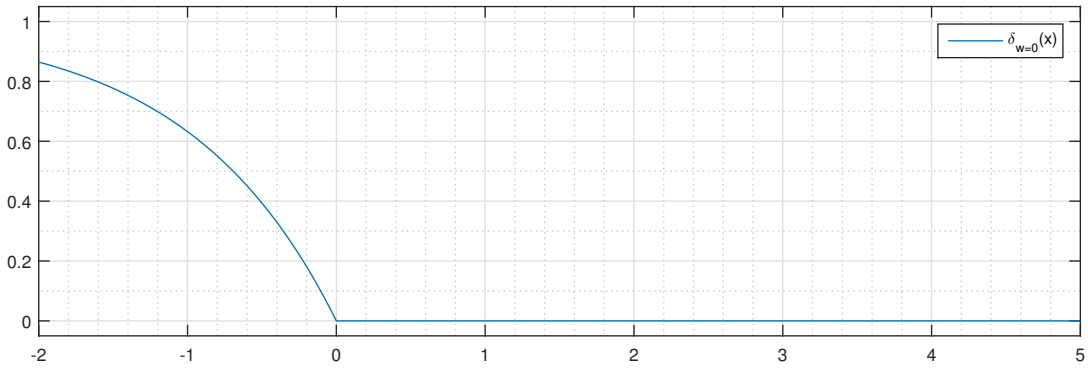

Figure 1: $\delta_{w=0}(x)$

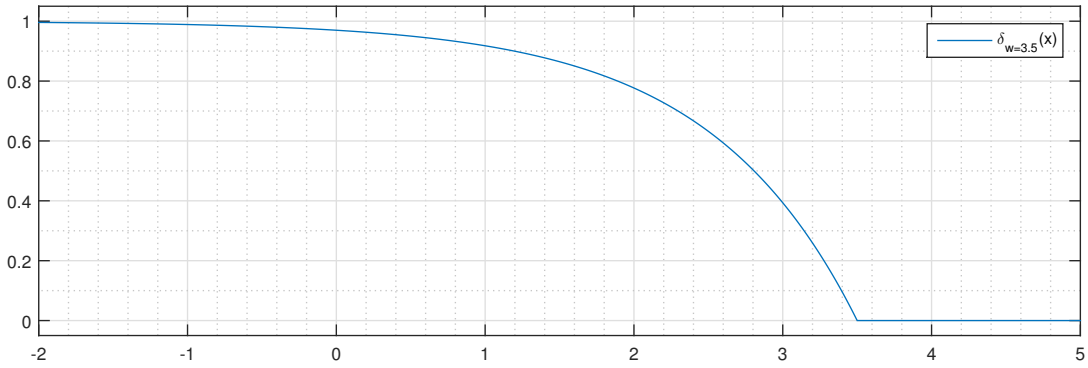

Figure 2: $\delta_{w=3.5}(x)$

II. Basics

This section explains some prerequisites and introduces some required functions.

A. Neural Network Layers

Neural networks are built with small function blocks. Such a function block often contains a linear function (e.g. a convolution) and a non-linear function to post-process the output. A well-known non-linearity is ReLU [5]. A fully connected layer may then look like this:

$$\begin{aligned} \mathrm{fc}_i(x) &= \mathrm{ReLU}(xW_i + b_i) \\ x &\in \mathbb{R}^N \\ W_i &\in \mathbb{R}^{N \times M} \\ b_i &\in \mathbb{R}^M \end{aligned}$$

In this example W_i and b_i are trainable weights.

B. The $\delta_w(x)$-Function

In this section the $\delta_w(x)$-function is introduced. It is defined by the following expression:

$$\delta_w(x) = \mathrm{ReLU}(1 - \exp(x - w))$$

This function is 0 for all $x \geq w$ and grows fast to 1 for $x < w$. This function is (non-strictly) decreasing and, therefore, $\forall x_0, x_1 \in \mathbb{R} : x_0 < x_1 \Rightarrow \delta_w(x_1) \leq \delta_w(x_0)$. Figure (1) and (2) visualize the $\delta_w(x)$-function for different w-values. The w value is the only parameter of this function; if the function is used in a neural network, it may be a trainable parameter. The function is everywhere, except for $x = w$, differentiable.

III. Approach

A. The specific $g_\infty(x)$-Layer

A fully connected neural network layer may be modified to the following expression:

$$\mathrm{fc}_i(x) = \mathrm{ReLU}(x(I + W_i) + b_i)$$

This layer is able to learn the same set of functions as the previous defined fully connected layer. It may be re-written again to obtain the following expression:

$$\mathrm{fc}_i(x) = \mathrm{ReLU}(x + xW_i + b_i)$$

The function is again modified: The $\delta_w(x)$-function is included to control how much of the linearity should be used. The $\delta_w(x)$-function returns a scalar value which is multiplied with the resulting vector of the linearity:

$$\mathrm{fc}_i(x) = \mathrm{ReLU}(x + (xW_i + b_i)\delta_w(i))$$

If $\delta_w(i) > 0$, then the function may still learn any mapping that a fully connected layer can learn. If $\delta_w(i) = 0$, then the function is reduced to $\mathrm{ReLU}(x)$.

The $g_\infty(x)$-layer is now introduced. An important restriction of it is that its output dimension is equal to its input dimension. It is defined by an infinite composition of functions:

$$\begin{aligned} g_0(x) &= x \\ g_{i+1}(x) &= \mathrm{ReLU}(g_i(x) + (g_i(x)W_i + b_i)\delta_w(i)) \\ g_\infty(x) &= \lim_{n \to \infty} g_i(x) \\ x &\in \mathbb{R}^N \\ W_i &\in \mathbb{R}^{N \times N} \\ b_i &\in \mathbb{R}^N \end{aligned}$$

The result of the layer is the output of $g_\infty(x)$. To obtain this result, the layer is analyzed more detailed. First we count the weights: Assume x is an N-dimensional vector. For every $i \in \mathbb{N}_0$ every $g_{i+1}(x)$-function contains the weights W_i and b_i and, therefore, $N \times N + N$ weights. This results obviously in

(countable) infinite weights. An additional single weight is the w-parameter of the $\delta_w(x)$ function. As already shown, the $\delta_w(x)$-function is always equal to 0 for all $x \geq w$, therefore, only the weights of the functions $g_{i<w}(x)$ have any effects to the calculations. This actually means that only $\lceil w \rceil$ linearities are used inside $g_\infty(x)$. All $g_{i \geq w}(x)$ functions are just equal to $\mathrm{ReLU}(x)$. Given the identity

$$\mathrm{ReLU}(\mathrm{ReLU}(x)) = \mathrm{ReLU}(x)$$

it can be seen that $g_\infty(x) = \mathrm{ReLU}(g_{\max(0,\lceil w \rceil)}(x))$. Therefore, the result of $g_\infty(x)$ can be obtained, even if there are infinite many sub-layers, because effectively only $\lceil w \rceil$ sub-layers are used. Now only $\max(0, \lceil w \rceil)N(N+1)+1$ weights are used to calculate this value. All other weights do not have an effect to the result. This means the gradient of the result with respect to these weights is equal to 0. The weights may not even exist in an implementation before w grows up to the value when they are used.

This new defined $g_\infty(x)$-layer just may be used in a neural network like any other layer. The optimization still works, because in every step only a finite amount of weights is optimized. The amount of optimized weights is controlled by the w-parameter of the $\delta_w(x)$-function. This parameter is also optimized and, therefore, the network can control the amount of parameters that are used and optimized. It defines the depth of the meta-layer $g_\infty(x)$. This layer may be included in a neural network as any other layer. For implementations, it may be required to check the value of w after every minibatch and add some layers to the model if necessary.

B. The generalized $g_{f_i;h;\infty}(x)$-Layer

The previously defined $g_\infty(x)$-layer may be generalized. Instead of $\mathrm{ReLU}(x)$ any other function with the identity

$$h(h(x)) = h(x)$$

may be used. E.g. $h(x) = x$, $h(x) = \mathrm{BatchNorm}_{\gamma=1,\beta=0 \text{ fixed}}(x)$ and $h(x) = \mathrm{ReLU}(x)$ are suitable candidates.

It is also possible to replace the $xW_i + b_i$ expression by any other function that has the same input-shape like its output-shape. This function is called $f_i(x)$. E.g. a K-dimensional convolutional-layer, given the correct padding, stride and feature count is defined, may be used or an LSTM-layer. Also compositions like $\mathrm{Dropout}_{p=0.5}(\mathrm{Conv2D}(x))$ work quite well. The used function also may vary from layer to layer. There is no restriction to always use the same function, therefore, one may, e.g., define $f_{2k}(x) = \sigma(xW_i + b_i)$ and $f_{2k+1}(x) = \mathrm{BatchNorm}(\mathrm{Dropout}_{p=0.5}(xW_i + b_i))$.

Therefore, $g_{f_i;h;\infty}(x)$ can be defined as follows:

$$\begin{aligned} g_{f_i;h;0}(x) &= x \\ g_{f_i;h;i+1}(x) &= h(g_i(x) + f_i(x)\delta_w(i)) \\ g_{f_i;h;\infty}(x) &= \lim_{n\to\infty} g_{f_i;h;n}(x) = h(g_{\max(0,\lceil w \rceil)}(x)) \end{aligned}$$

In this notation the previously defined specialized version of the function may be written as $g_{xW_i+b_i;\mathrm{ReLU}(x);\infty}(x)$. A visualization of this meta-layer may be seen on Figure (3). As can be seen this meta-layer is similar to a ResNet architecture [6].

It is important that $f_i(x)$ is able to return positive and negative values. If all values have the same sign, the sum might diverge and no useful training is possible.

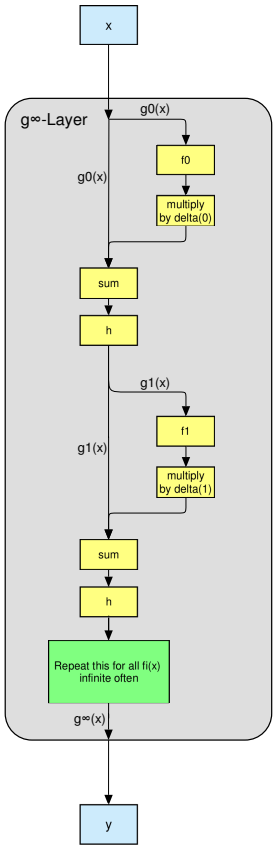

Figure 3: A visualization of the internal structure of a $g_{f_i;h;\infty}(x)$-layer.

C. Optimization

There are several issues in the optimization process that have to be handled carefully. Often there is only a finite amount of training data given, therefore, there is always the chance that a neural network overfits to this finite dataset. If the neural network has the chance to increase its parameter count to better fit the data, then it potentially just starts to overfit. To avoid this, a problem-specific ℓ_2-regularization is used for the w-parameter of the $\delta_w(x)$-function. Another problem occurs for a fixed i for the $f_i(x)$ function if $\delta_w(i)$ is small: The neural network should then decide if this new layer is useful and increase $\delta_w(i)$ (which is equal to increase w). As defined, this operation is not free, because of the ℓ_2-regularization of w. Therefore, if the network has the chance just to upscale $f_i(x)$ by a factor (e.g. $\delta_w(i)^{-1}$), then it does this. For this reason, if the output range of $f_i(x)$ is unbound, at least the last layer inside $f_i(x)$ should be regularized. If the last layer with parameters of $f_i(x)$ is a $\mathrm{BatchNormalization}(x)$-layer, then the ℓ_2-regularization is applied to regularize γ. For fully connected layers an ℓ_2-regularization may be used for the weights.

Different applications, also depending on the complexity of the chosen layers, may require different regularization factors.

$f_i(x)$ produces now for many layer types regularizations that increase the loss. Because there are infinite many $f_i(x)$ sub-layers, the loss would grow up to ∞. This obviously is hard to handle and optimize. To handle this case, the regularization of the function $f_i(x)$ is a conditional ℓ_2-regularization. It is multiplied by $\delta_w(i+\varepsilon)$, which has the effect that if $\delta_w(i)$ is equal to 0, then the regularization

loss for $f_i(x)$ is ignored. The small ε shift is used to guarantee if $w = n$ for $n \in \mathbb{N}_0$ that the weight for the regularization and the derivative of this weight factor is 0. Without the ε, the gradient would be undefined, but $w = n$ means that the nth layer is inactive (with n starting at 0), therefore the gradient should be 0. The regularization is given by the following expression, where i is the sub-layer index, x the weights and ε a small number (e.g. 10^{-8}):

$$\delta_w(i + \varepsilon)\ell_2(x)$$

The described meta-layer is everywhere continuous and has infinite many sub-layers, but at every timestep the maximum amount of used layers is exactly $\lceil w \rceil$. Therefore, just a standard optimizer may be used. A problem is if new layers are accessed, then the network has to add them. This may be implemented by using a new model that just has a new layer appended; of course, this new model uses the weights of the previous model. Many optimizers have an internal state [7] [8] which also has to be initialized according the previous training state. Creating every time when a layer is added a new model is quite slow for complex models, therefore, the implementation may always add 5 or even 10 layers if a new layer is accessed that is not contained in the current model.

In general one has to be very careful about the chosen regularizations, especially if $f_i(x)$ contains many fully connected layers. If the regularizations are too low, then the model tries to overfit to the training data.

IV. Experiments

The proposed meta-layer is implemented in a small library that is based on Keras [9]. This simple implementation allows it to create feed-forward neural networks with this new layer-type. It only was tested with the TensorFlow [10] backend. The complete source code, including all experiments, is available on GitHub: https://github.com/kutoga/going_deeper. Every time a new layer that does not yet exist is accessed, there are 5 new layers created. This minimizes the time that is required to always rebuild the model. If the count of the newly added layers per time is larger, the training process may be slower, but this highly depends on the total amount of required layers. The internal state of the used optimizer [8] is not transfered if a new layer is added. The optimization still works, but in our examples, there are not more than 5 layers used.

A. XOR-Network

The XOR-network experiment uses 8 binary inputs and calculates the XOR-operation over them, Therefore, the problem is a binary classification problem. To control the complexity of the network, only some of the inputs are really used to calculate the XOR-results. Such tests are done from 0 up to 8 active inputs. For 0 active inputs it is assumed that the output is always 0. For 1 activate input, the output is equal to the input. Always the first n inputs are relevant for the output, all other inputs contain random binary values. The network architecture contains an initial randomly initialized and not trainable fully connected layer to create an internal representation size of 24, then a

$$g_{\mathrm{Dropout}_{p=0.1}(\mathrm{BatchNormalization}(xW_i+b_i));\mathrm{ReLU}(x);\infty}(x)$$

-layer after the input and finally a not trainable, random initialized fully connected layer with 1 unit and a sigmoid-activation. Therefore, all trainable-weights are in the $g_\infty(x)$-layer included. The $\mathrm{BatchNormalization}$ inside the $g_\infty(x)$-layer uses a ℓ_2-regularization for the γ-parameter with a factor

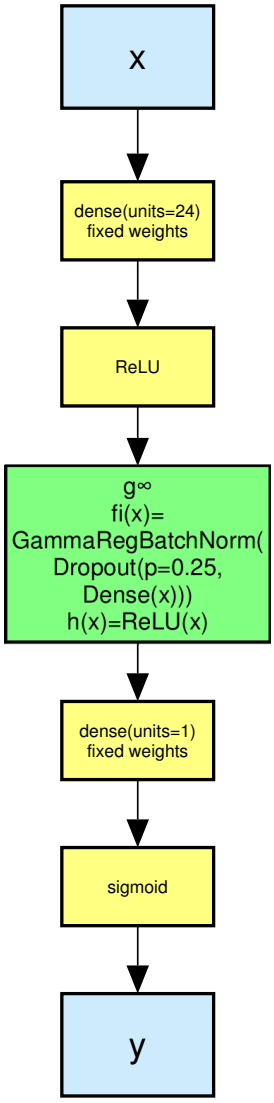

Figure 4: XOR-Experiment Architecture

of 10^{-8}. On the other side, the w-parameter uses an ℓ_2-regularization with a factor of 10^{-5}. Different regularization values were tested, but these two seem to be a reasonable choice. The internal representation always has a size of 24 values. The architecture is visualized in Figure (4).

These experiments are conducted on a Intel® Core™i7-3630QM CPU. All networks in the experiments quite fast reach an accuracy of 100% as can be seen on Figure (5). Also the loss decreases very fast (see Figure (6)). It may be observed that the w-values first increase and then they start to decrease. This means the networks first use too many sub-layer and then start to decrease the number of used sub-layers. This may happen due to the optimization of the already allocated layers. This effect is visualized on Figure (7). It can also be seen that networks with more input values for the XOR-function use more sub-layers. Therefore, it can be concluded that more complex tasks allocate more layers. All used data records for the experiment are just generated at runtime. Finally, it can be seen that the network handles this task quite well.

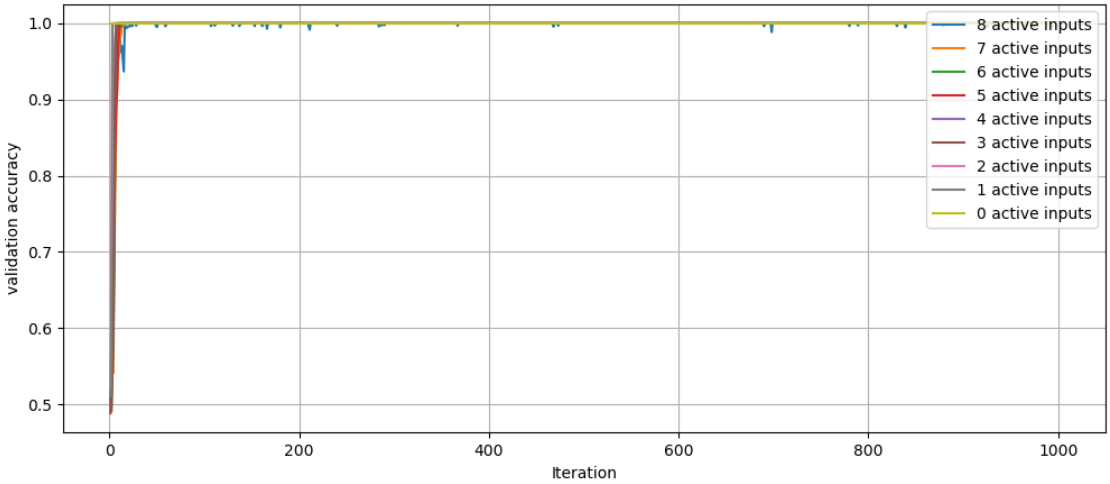

Figure 5: XOR-Network: Validation Accuracy

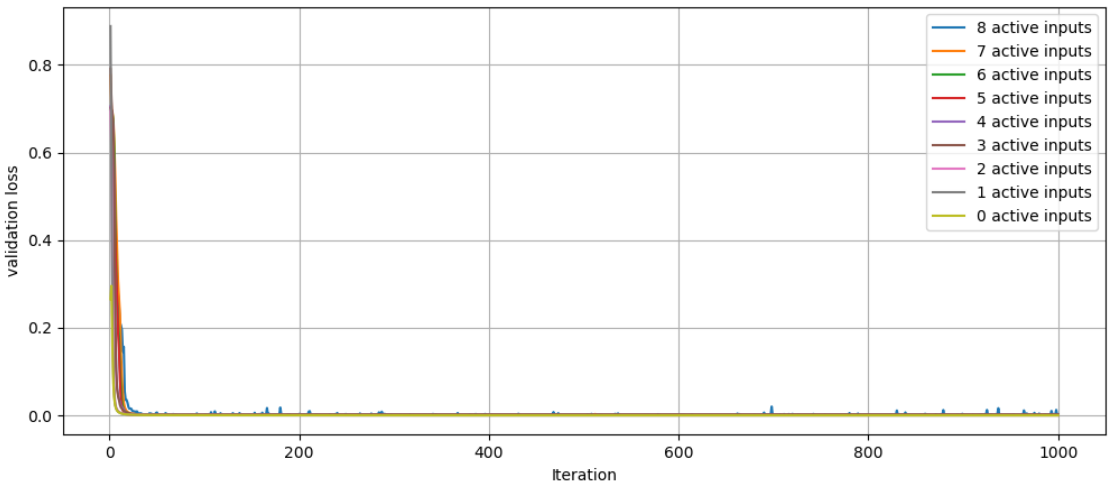

Figure 6: XOR-Network: Validation Loss

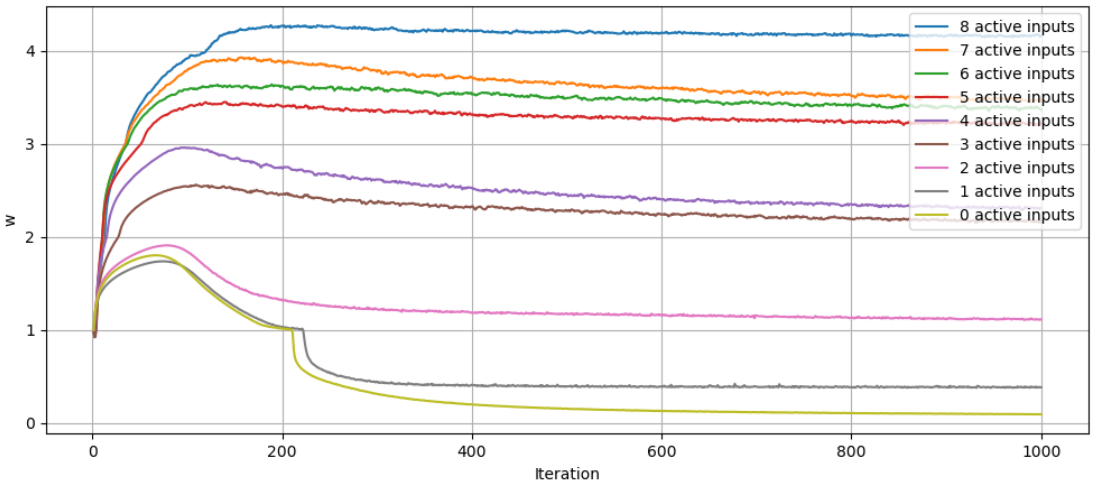

Figure 7: XOR-Networks: w-Values

B. MNIST-Network

MNIST [11] is a well-known digit classification dataset with 10 classes. In this experiment, a CNN is used to classify these digits. All trainable weights are inside of three $g_\infty(x)$-layers. The network is trained on the pre-defined training set with $50'000$ images and tests are done on the test set with $10'000$ images.

The first layer of the network is a random initialized convolutional layers that has fixed weights. It is used to reshape the input for the following $g_\infty(x)$-layer. This convolutional layer uses a kernel of the size 3×3, a stride of 1, padding of 1 and 30 filters. After this layer, there is a

$$g_{\mathrm{Dropout}_{p=0.1}(\mathrm{BatchNormalization}(\mathrm{Conv2D}_{3\times 3}(x)));\mathrm{ReLU}(x);\infty}(x)$$

-layer. The stride and the padding of the used convolutional sub-layer are 1. After this layer, there is a $\mathrm{MaxPooling}$-layer to reduce the dimension of the current representation. It uses a kernel and stride of 2×2. Then, again, there is a random initialized convolutional layer with fixed weights. It is also used to reshape the internal representation for the next $g_\infty(x)$-layer. It uses a kernel of the size 3×3, a stride of 1, padding of 1 and 60 filters. After this layer, there is again a

$$g_{\mathrm{Dropout}_{p=0.1}(\mathrm{BatchNormalization}(\mathrm{Conv2D}_{3\times 3}(x)));\mathrm{ReLU}(x);\infty}(x)$$

-layer with a stride and padding of 1. After this layer, a random initialized fully connected not trainable layer with 256 units is used to reduce the dimension of the current representation. With a

$$g_{\mathrm{Dropout}_{p=0.1}(\mathrm{BatchNormalization}(xW_i+b_i));\mathrm{ReLU}(x);\infty}(x)$$

-layer the network has again the chance to learn something. Finally, there is a fully connected layer with 10 units and a $\mathrm{softmax}$-activation. The first two convolutional $g_\infty(x)$-layers use an ℓ_2-regularization for the w-parameter with a factor of $5 * 10^{-5}$ and an ℓ_2-regularization for the γ-parameter of the $\mathrm{BatchNormalization}$-layer with a factor of 10^{-8}. The fully connected $g_\infty(x)$-layer uses also an ℓ_2-regularization for the w-parameter, but with a factor of 10^{-3} and an ℓ_2-regularization for the γ-parameter of the $\mathrm{BatchNormalization}$-layer with a factor of 10^{-8}

The model architecture is quite standard and simple, probably it could be improved. The used loss-function to train the network is the categorical crossentropy. The network was trained for $5'000$ epochs on a single NVIDIA Titan X Pascal GPU.

Figure (9) shows that the network loss is quite fast very low and Figure (10) shows that the accuracy for the test set reaches quite fast 99.5%. The more interesting result is shown on Figure (11). The visualization shows the depths of the different $g_\infty(x)$-layers. As can be seen, the depth of the fully connected $g_\infty(x)$-layer at the end increases quite fast, but early stops and then it decreases. Finally the network uses the following w-parameters:

- $w_{\mathrm{cnn0}} = 0.339... \Rightarrow$ There are $\lceil w_{\mathrm{cnn0}} \rceil = 1$ sub-layer(s) used for the first convolutional $g_\infty(x)$-layer.
- $w_{\mathrm{cnn1}} = 1.928... \Rightarrow$ There are $\lceil w_{\mathrm{cnn1}} \rceil = 3$ sub-layer(s) used for the second convolutional $g_\infty(x)$-layer.
- $w_{\mathrm{fc0}} = 0.406... \Rightarrow$ There are $\lceil w_{\mathrm{fc0}} \rceil = 1$ sub-layer(s) used for the first fully connected $g_\infty(x)$-layer.

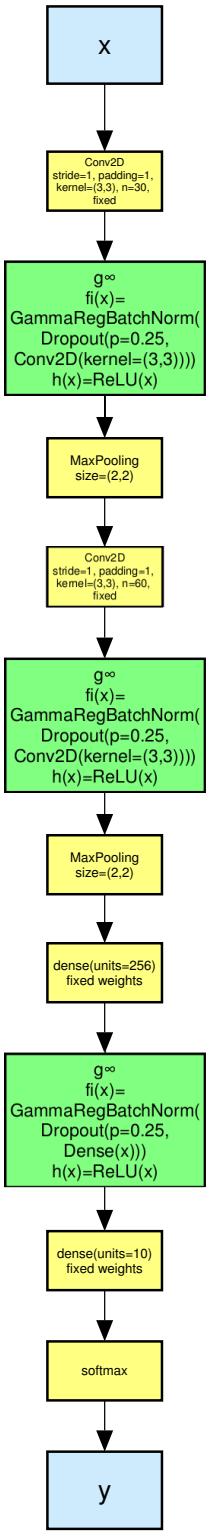

Figure 8: MNSIT-experiment architecture

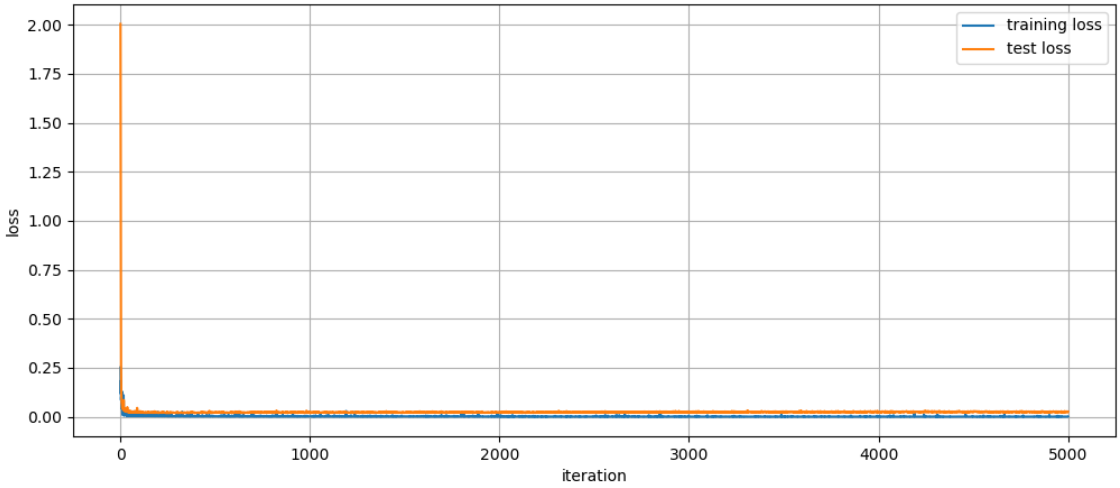

Figure 9: MNSIT-Experiment Loss

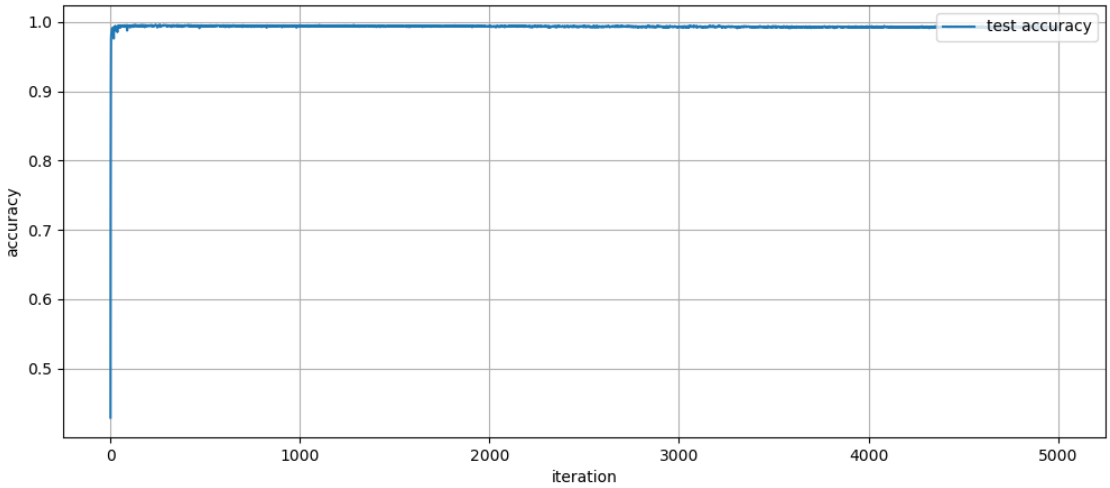

Figure 10: MNSIT-Experiment Accuracy

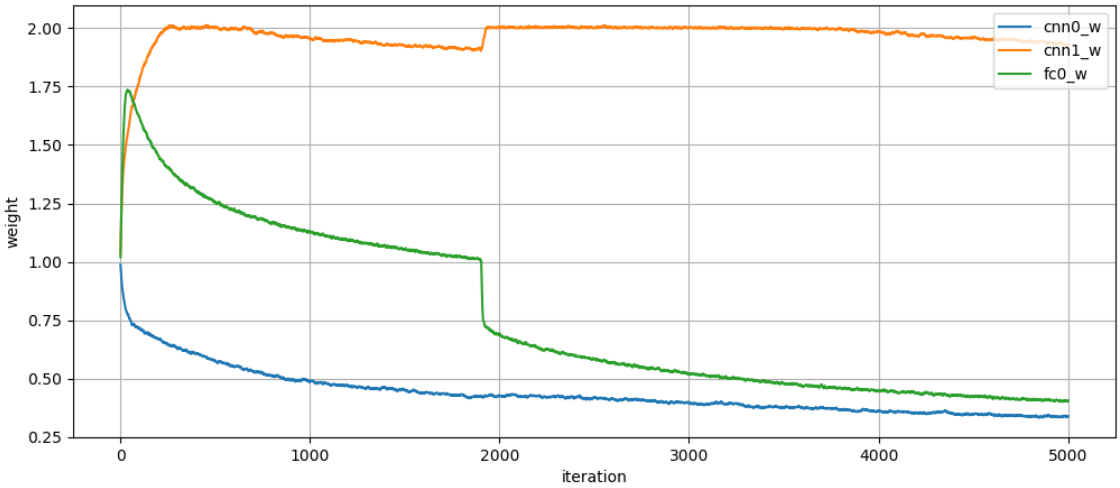

Figure 11: MNSIT-Experiment Weights

V. Future Work

This paper just might be seen as another try for infinite deep neural network architectures. The core idea may be extended in many directions. E.g., one could think about not growing the network in depth, but in width. This can be done very similar, only the merging process may become more difficult. One could still use addition, but also using an LSTM might be a good choice; even if the use of LSTM might be a bit tricky. Growing in width might be less problematic according to overfitting. It also could be possible to pyramid the described meta-layer. Some more ideas are described in the following sub-sections.

The described meta-layer could be used in different other architectures, e.g. in Generative Adversarial Networks (GANs) [12].

A. Merging-Function

In this document the merging function is defined before $h(x)$ as the sum of of $\delta_w(i)f_i(x)$ and $g_i(x)$, but other functions would be possible. E.g., a merging function based on the geometric mean:

$$g_{i+1}(x) = g_i(x)f_i(x)^{\delta_w(i)}$$

This function requires strictly positive values for $f_i(x)$. To avoid this problem it may be changed to the following expression.

$$g_{i+1}(x) = g_i(x)\sqrt{(f_i(x)^2 + \varepsilon)^{\delta_w(i)}}$$

It is assumed that the ResNet-like [6] architecture, based on the addition, works better, because already ResNet works very well compared to other network architectures. This example just should show that this merging function may be replaced by another expression. The only important aspect is that the expression is equal to $g_i(x)$ if $\delta_w(i)$ is equal to 0.

B. w-Regularization

The current used regularization for w does not depend on the parameter-count of the current available sub-layers or the parameter-count per new sub-layer. If a neural network uses multiple of these dynamic layers it may always prefer to increase w for the meta-layer where it gets the most new parameters. A solution to this problem would be to use a regularization that depends on the parameter-count.

Additionally, currently it is assumed that the ℓ_2-regularization is suitable for the w-parameter. It may be the case that other regularization functions work much better for this parameter.

Instead of using the factor $\delta_w(i+\varepsilon)$ for the regularization of the weights of $f_i(x)$, one could also use a step-function to enable the regularization as soon as the layer is used. Therefore, the regularization would be modified to the following expression:

$$s(\delta_w(x))\ell_2$$

A candidate would be a hard-step function:

$$s(x) = \begin{cases} 0 & x \leq 0 \\ 1 & \text{else} \end{cases}$$

$$\frac{\mathrm{d}}{\mathrm{d}x}s(x) = 0$$

It is important that the derivate at $x = 0$ is equal to 0, otherwise weights of not used layers may be modified; there are infinite many such weights, therefore, this is not an option. A disadvantage of this hard-step function is, that the neural network (viewed as a function) is no longer continuous.

Therefore, the hard-step function $s(x)$ may not be that nice, because it produces functions that are not everywhere continuous. One could replace this function with a soft-step function $\widetilde{s}(x)$ that has the following properties:

$$\begin{aligned} \widetilde{s}(0) &= 0 \\ \frac{\mathrm{d}}{\mathrm{d}x}\widetilde{s}(x)\Big|_{x=0} &= 0 \\ \widetilde{s}(\varepsilon > 0) &\widetilde{=} 1 \end{aligned}$$

The first property is required to do not count all the regularizations of the infinite available layers. The second property makes the derivative for the regularizations of weights of the infinite many unused sub-layers equal to 0. The third property is more a soft-constraint: It just implements the idea that weights of layers which are currently used should get a high penalty to their weights.

C. The $\delta_w(x)$-Function

It could be asked if the $\delta_w(x)$-function is a good choice or if it should be replaced by another function $f_w(x)$ with the following properties:

$$\begin{aligned} f_w(x \geq w) &= 0 \\ f_w(x < w) &\widetilde{=} 1 \\ f_w(x) &\in [0, 1] \\ f_w(x + c) &\leq f_w(x), c \in \mathbb{R}_+ \end{aligned}$$

The first property is used to make most of the infinite many layers inactive. This is very important for the optimization process. The second property implies that if a layer is used it should be very active; which means $f_w(x)$ should produce a value near to 1. This constraint may be seen as a soft-constraint. The third property defines that the function $f_w(x)$ should only produce values in $[0, 1]$. This constraint also may be seen as a soft-constraint. The last property defines that the function is (non-strictly) decreasing. This property is helpful for the optimization process, because it implies that if a layer should be more active, then the function should be shifted to the right. But also this property might be seen as a soft-constraint.

One could, e.g., think about using the function $\delta_{w;\alpha} = \mathrm{ReLU}(1 - \exp(\alpha(x - w)))$ as a good alternative. This function contains a new parameter α that can be optimized by the neural network. It allows the network to stretch the previous defined $\delta_w(x)$ function. Of course, there are many other possible functions.

VI. Conclusions

As can be seen, there are many things that might be tested. This paper just shows the core idea. One could also show that this architecture does not make sense at all. Hopefully, another more powerful architecture could then be recommended.

VII. Will the author do the future work?

Unfortunately, the author of this (just-for-fun) paper-like document currently has no time to investigate more on this model. This is also the main reason why this report was written. Later, probably in a few months, he will continue experimenting with models like this. Maybe this report and some of its ideas may be used by someone else.

Acknowledgment

You (finally) reached the end of the document. I like to thank you for reading it ◡̈!

References

[1] B. C. Csáji, "Approximation with artificial neural networks," *Faculty of Sciences, Etvs Lornd University, Hungary*, vol. 24, p. 48, 2001.

[2] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019*, 2015.

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[4] H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets," *Journal of computer and system sciences*, vol. 50, no. 1, pp. 132–150, 1995.

[5] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[7] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[8] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[9] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[11] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[12] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.