Training neural network with zero weight initialization

Schicho Christopher

February 2023

1 Introduction

Research and practical experiments over the years demonstrate the importance of parameter initialization of neural networks (NNs). In this work, we refer to parameters as the set of weights and biases of a NN. The major difficulty in weight initialization is finding the right scale and distribution of values. Small values may lead to vanishing gradients in the backward pass, and thus to a slow convergence of the NN. On the other hand, large values may cause exploding gradients, which may lead to instability during the training.

There is no universal remedy for the initialization of NN parameters. Therefore, a variety of approaches have been explored in literature [15][5][2][9][13]. One group of these approaches is random weight initializers. This is the most commonly used category of weight initializers [15], which usually draws the values from a normal or uniform distribution in combination with a certain heuristic nowadays. Besides random initialization, there is a second category of weight initializers, zeros and ones initialization [15] or constant initialization in general.

Zero initialization is the process of setting all weights of a NN to 0. The sole use of zero initialization leads to equal activations across each layer, as we show in Section 2.3. Thus, during backpropagation each weight in a layer receives the same update, this makes layers symmetric. Eventually, the capability of such NNs is reduced tremendously, and they behave like a linear model [15].

In this work, we suggest a random initialization of the bias of NNs initialized with zero initialization to overcome the problem of symmetrically evolving neurons. To investigate the issue, we performed a comprehensive set of experiments for a range of NN architectures and different datasets. The results provide evidence that by initializing the bias, we can break the symmetry and thus overcome the shortcoming of zero initialization.

The report is organized as follows. We briefly review main concepts related to our NN parameter initialization in Section 2. In Section 3 we present the results of our experiments on NN parameter initialization for the MNIST [3] and CIFAR-10 [1] datasets. In the following Section 4 we discuss the main results and findings of our work. Finally, we draw the conclusions in Section 5.

2 Background

2.1 Parameter initialization

The initialization of NN parameters has a significant influence on the effectiveness of the training process [16]. Therefore, there exist many different methods and approaches [15][5][2][9][13] on how to effectively initialize those parameters.

The LeCun initialization [5] is a commonly used initialization method. This approach exists for normal and uniform distribution, in this work we focus on the one which draws its values from a uniform distribution. Using this approach would result in an initialization which draws its values from the uniform distribution $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = 3/n$ and n is the number of inputs to the particular layer.

PyTorch, the machine learning framework we used in our practical experiments, uses a scaled version of this initialization approach. For example, PyTorch draws its random values for the parameters of the linear NN layers by default from a uniform distribution $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = 1/n$ and n is again the number of inputs to the particular layer. Thus, this approach has a lower standard deviation compared to LeCun initialization [5]. The background for this has not been recorded in literature to our knowledge. We call this initialization approach *PyTorch initialization* in this work.

Thus, we get the following distributions for random parameter initialization using the PyTorch initialization for the layers of our NNs:

- Linear layer:
  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = 1/H_{in}$ and H_{in} is the number of input features of the linear layer.

- Convolutional layer:
  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = 1/(C_{in} * k_w + k_h)$, C_{in} are the input channels, k_w is the kernel width and k_h is the kernel height of the convolutional layer.

In our proposed initialization approach, we use zero initialization combined with random bias initialization. As our goal is to show that this approach can break the symmetry introduced by zero initialization, we sampled the value of the biases in the experiments either from a standard normal distribution $\mathcal{N}(0, 1)$ or a uniform distribution on the interval $[0, 1)$ for simplicity reasons.

We introduced the well known LeCun initialization and the scaled version which is used by PyTorch, which we use in the rolled out experiments to compare our initialization approach against. In this section, we also briefly described our approach and the distributions we used in our experiments.

2.2 Effect of activation function

The choice of the right activation function for the hidden layers of a NN is essential to reach good results. Despite the popularity of the Rectified Linear

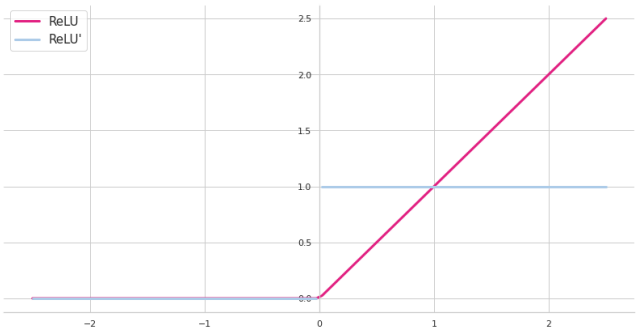

Figure 1: ReLU function and ReLU derivative

Unit (ReLU) activation function [4], it has an important drawback, which we address in this chapter.

One of the drawbacks of ReLU, $\max\{x, 0\}$, is what is commonly referred to as 'dying ReLU' or 'dead neuron'. A 'dying ReLU' occurs when the activation is 0, this happens when the input x of the activation function is ≤ 0. Once the activation of a neuron is zero, the resulting gradient of the backpropagation is zero as well, due to the gradient of the ReLU as shown in Figure 1. Thus, the affected neuron receives no update during the backpropagation, and we consider it as 'dead'. The worst case would be an entire layer of 'dead neurons', this would cut off the gradient flow to previous layers, thus those layers would receive no updates.

The selection of a suitable activation function is of critical importance for our proposed initialization approach. As we initialize the network weights with zero, only the bias controls the sign of the pre-activation in the initial phase. As an example, this would result in approximately 50% of the NN neurons 'dying', when initializing the bias with random values drawn from a standard normal distribution $\mathcal{N}(0, 1)$. This would limit the capability of the neural network dramatically.

The problem of 'dead neurons' can be addressed by using activation function which are not constant in the negative region. Several activation function with this characteristic have been proposed by researchers, some of them are e.g. Leaky ReLU [12], Parametric ReLU [10], SELU [14] and ELU [7].

In this section, we highlighted the necessity of selecting the right activation function to avoid the problem of 'dead neurons'. Furthermore, we showed the importance of this problem for our initialization approach and gave some examples of alternative activation functions to avoid this problem.

2.3 Symmetry in neural networks

In this section, we briefly explain the problem of symmetry in NNs, and we show how zero initialization cannot break this symmetry. Furthermore, we also

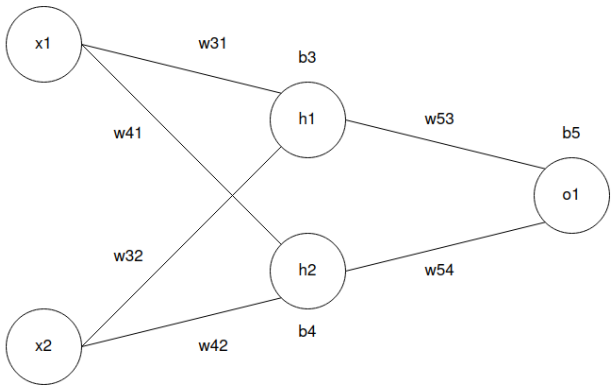

Figure 2: Neural network with one hidden layer

explain why our proposed initialization approach is able to break this symmetry introduced by zero initialization.

In the context of NNs, symmetry refers to the property that multiple neurons or sets of neurons are structural identical and have the same outputs. Due to backpropagation, neurons in the same layer, having the same output value, receive the same update. Thus, such neurons cannot learn different features. Depending on the number of symmetrical neurons this can be seen as a natural dropout, however, having too many such neurons in a NN leads to reduced capabilities of the network. In the worst case, where each neuron is symmetric to all the other neurons in a NN, this can make the network behave like a linear model [15]. A source of such symmetry is initializing a NN with a constant value, such as zero.

For the following example, consider the NN from Figure 2. Let θ be the set of weights $w_{31}, w_{32}, w_{41}, w_{42}, w_{53}, w_{54}$ and biases b_3, b_4, b_5 of the NN g. Furthermore, let $f_a(x)$ be any activation function and L be any loss function. The data is denoted by $x = (x_1, x_2)$ for the input and y for the real label of the input.

For demonstrating that zero initialization cannot overcome symmetry, assume we use zero initialization for the weights of the NN from Figure 2, and we initialize the bias with zero as well. When forward propagating the input x, the preactivation s_i and activations a_i of the hidden layer are calculated as follows.

$$s_3 = w_{31}x_1 + w_{32}x_2 + b_3 \tag{1}$$

$$a_3 = f_a(s_3) \tag{2}$$

$$s_4 = w_{41}x_1 + w_{42}x_2 + b_4 \tag{3}$$

$$a_4 = f_a(s_4) \tag{4}$$

Using the results of the hidden layer activations, we can compute the output of the NN as follows.

$$s_5 = w_{53}a_3 + w_{54}a_4 + b_5 \tag{5}$$

$$g(x;\theta) = f_a(s_5) \tag{6}$$

Because all weights and biases are initialized with zero, it follows $s_3 = s_4$ regardless of the input. Thus, this implies $a_3 = a_4$. Therefore, the neurons in each individual layer have identical contribution to the loss of the output, what we show in the following.

We can rewrite the gradient needed for the update step as the product of the delta-error δ_i and the activation a_j.

$$\frac{\partial}{\partial w_{ij}} L(y, g(x;\theta)) = \delta_i a_j \tag{7}$$

$$w_{ij}^{new} = w_{ij}^{old} - \eta \delta_i a_j \tag{8}$$

We need to distinguish between the error of the output layer and the errors of the hidden layers. The delta-error of the output layer is calculated by using Equation 9. For the delta-error of the hidden layers we need to consider the layer above the layer for which we calculate the error, this is done by using Equation 10.

$$\delta_k = \frac{\partial}{\partial a_k} L(y, g(x;\theta)) f_a'(s_k) \tag{9}$$

$$\delta_j = f_a'(s_j) \sum_i \delta_i w_{ij} \tag{10}$$

Using Equation 9 and 10 we can calculate the updates for the weights w_{31} and w_{32} as follows.

$$w_{53}^{new} = w_{53}^{old} - \eta \delta_5 a_3 \tag{11}$$

$$w_{54}^{new} = w_{54}^{old} - \eta \delta_5 a_4 \tag{12}$$

We need to note here, that because $a_3 = a_4$ we have $\eta\delta_5 a_3 = \eta\delta_5 a_4$. Thus, both weights receive the same update in each iteration. This shows the symmetry introduced by zero initialization and because all the neurons in a layer receive the same update during backpropagation, the neurons in each layer cannot learn different features, instead all of them learn the same. Thus, their activation during the forward pass is again the same in the next iteration. Due to this repetition, zero initialized NN cannot break this symmetry. We omit the calculations of the remaining weights, as this follow the same structure and would not add any benefit.

We propose a combination of zero initialization and randomly initialized biases in order to break the symmetry introduced by zero initialization. For this assume the example from above, however, now let b_3, b_4, b_5 be values drawn from a random distribution.

We can now see, that by using Equation 1-5 we get that in general $a_3 \neq a_4$ because of $s_3 \neq s_4$ which is a result of $b_3 \neq b_4$, as these are randomly drawn values. Thus, each neuron in a layer has a different contribution to the loss of the output of the NN. Therefore, by using Equation 7-10 it follows that also the calculated gradient during the backpropagation is different and thus also the update the weights receive. Thus, this shows that the random asymmetry introduced by the initialization of the bias breaks the symmetry initially introduced by the zero initialization.

In this section, we explained what symmetry in a NN is and how zero initialization cannot break it. Furthermore, we scratched the theoretical background of our proposed initialization approach and showed why this approach can break the symmetry.

3 Experiments

This section presents the practical results for the initialization approaches described in Section 2.1 for a range of NN architectures. The goal of those experiments is to show the problem of symmetry introduced by using zero initialization, and to show that our proposed approach can break this symmetry.

The experiments have been performed on two different datasets: MNIST [3] and CIFAR-10 [1]. We ran the MNIST experiments for feedforward neural networks (FNNs) and a convolutional neural networks (CNNs) with a wide variety of initialization approaches to investigate the issue of symmetry in NNs. The CIFAR-10 experiments were additionally run for a ResNet-50 model [8] to confirm our conclusions on a more complex task.

All experiments, which use our proposed initialization approach, use zero initialization combined with random bias initialization. As our goal is to show that this approach can break the symmetry introduced by zero initialization, we sampled the value of the biases in the experiments either from a standard normal distribution $\mathcal{N}(0, 1)$ or a uniform distribution on the interval $[0, 1)$ for simplicity reasons.

For all the results, the reported accuracy is measured on the test samples of the datasets and is the mean over 5 runs of training to account for randomness. The error bars in the figures show a confidence level of 95%. All networks were trained until they did not significantly improve anymore, the number of epochs is reflected in the corresponding figures.

We omitted data augmentation for the FNNs and CNNs, for the reason of isolating the effect of our changes to the initialization approach. For the ResNet-50 model we used a data augmentation for the CIFAR-10 input images, the used methods are discussed in Section 3.2.

We based the optimization on adam optimization [6] with a learning rate of 0.001. The batch size was 128 samples across all the experiments. We decided to not optimize these hyperparameters with the purpose of isolating the effect of changes in our initialization approach.

We used cross entropy as criteria to optimize the performed experiments.

MNIST - NNs	
FNN	CNN
	conv 5x5x16, padding 2
	conv 5x5x32, padding 2
dense 784, 500	
dense 500, 10	dense 25088, 10

Table 1: MNIST NNs architecture. Description of the layers used for the NN architectures. Each layer is followed by an ELU activation, the final layer has no activation

Also, we used a logistic regression as baseline model for the performed experiments. This model was used as a "sanity" check for our experiments.

As activation function, we used ELU [7] across all the networks. Firstly because of the problem of 'dead neurons' mentioned in Section 2.2 and secondly compared to Leaky ReLU [12] and Parametric ReLU [10], ELU [7] saturates to a negative value, which leads to smaller variations of the signals in the forward propagation.

The whole of hyperparameters and network architectures are selected by our experience and a conducted hyperparameter search. As our goal was not to achieve the highest possible accuracy, we selected networks which achieve good results, however, there might be better ones in terms of achievable performance.

All details of the implementations used in the experiments and the code used to perform all the experiments can be found in our repository on GitHub: https://github.com/c-schicho/ZeroInitialization

3.1 Experiments on the MNIST dataset

This section presents the results of the experiments performed on the MNIST dataset [3]. Furthermore, we compare our results to zero initialization and PyTorch initialization [5].

The MNIST dataset [3] is a collection of grayscale images of handwritten digits with the values from zero to nine. In total, it contains 70,000 images which are divided into 60,000 training and 10,000 test images, each with a resolution of 28x28 pixels.

For our first experiments, we used a FNN with two dense layers as described in Table 1. We trained this architecture for 10 epochs, as the accuracy did not improve significantly, when we trained it longer.

Figure 3 shows the results of the experiments with the FNNs. In this, we can see, that our approach outperforms the baseline model and get close to the accuracy of the initialization approach discussed in Section 2.1.

We further investigated the influence of the bias and performed several experiments with different scaling of the bias values. In Figure 4 we display exemplary results of our experiments. The results in this figure are from experiments in which the biases values were up scaled with factor 100 and down scaled with

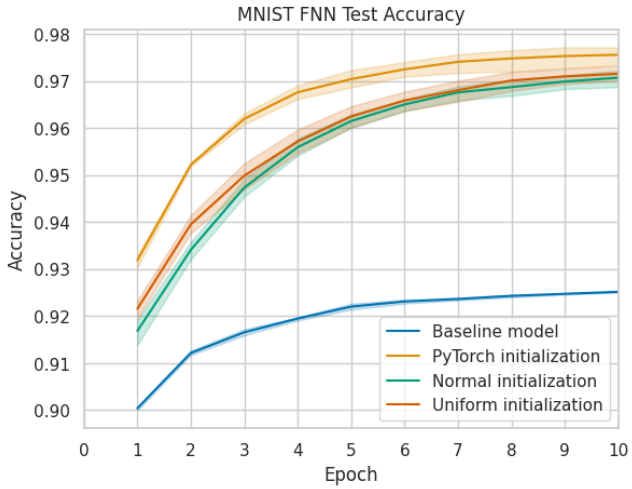

Figure 3: Test accuracy on the MNIST dataset of the baseline model and FNNs initialized with PyTorch initialization compared to our initialization approach using a normal and uniform distribution.

factor 0.01. We used this exemplary values, since they make the results more obvious. As we can see in Figure 4 a larger initial bias made the training in our experiments instable, whereas as small one had no negative impact.

In order to verify our findings also for CNNs, we trained a CNN with two convolutional and one dense layer as described in Table 1. We stopped training after 15 Epochs, as we did not observe significant improvements when training for longer.

In Figure 5 we display the results of the experiments with the CNNs. The figure shows that our approach also outperforms the baseline model using CNNs. Furthermore, the test accuracy of the CNNs initialized with our approach get close to the reference initialization from Section 2.1.

We also investigated the influence of initial bias scaling for the CNNs, the results are shown in Figure 6. We used the same values for scaling as for FNNs. In those experiments, upscaled biases had a larger impact. As Figure 6 shows, biases scaled by a factor of 100 made the network unable to learn.

Figure 7 shows the results of a FNN and a CNN initialized by zero initialization. In this figure, we can see that such networks were not able to outperform the baseline model in our experiments.

In this section, we presented the results of our experiments for the MNIST dataset [3]. We showed that our initialization approach was able to outperform the baseline model, whereas the networks initialized by zero initialization were not.

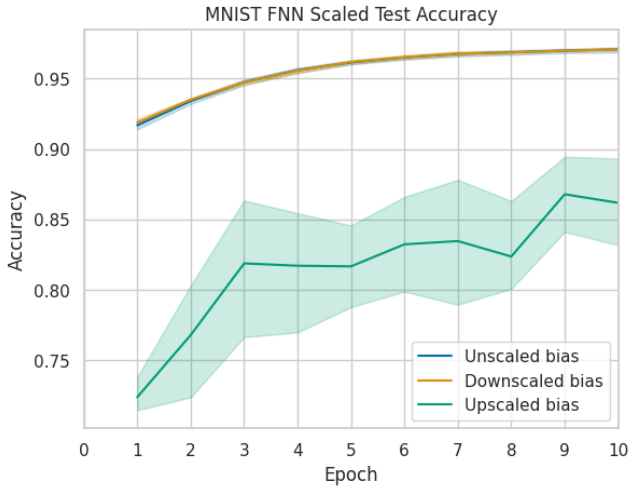

Figure 4: Test accuracy on the MNIST dataset of FNNS initialized by our approach with scaled and unscaled biases.

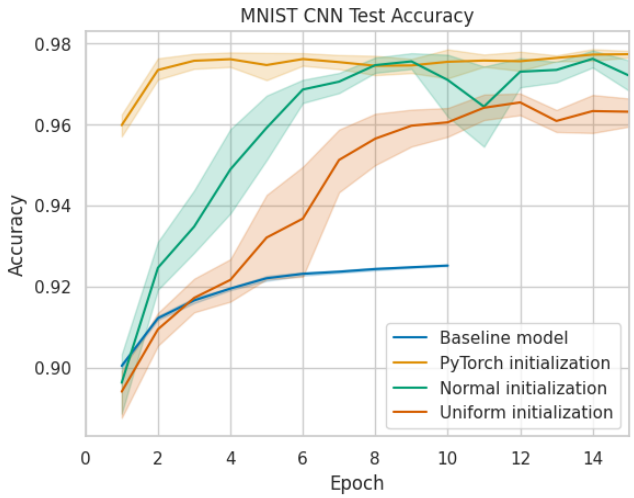

Figure 5: Test accuracy on the MNIST dataset of the baseline model and CNNs initialized with PyTorch initialization compared to our initialization approach using a normal and uniform distribution.

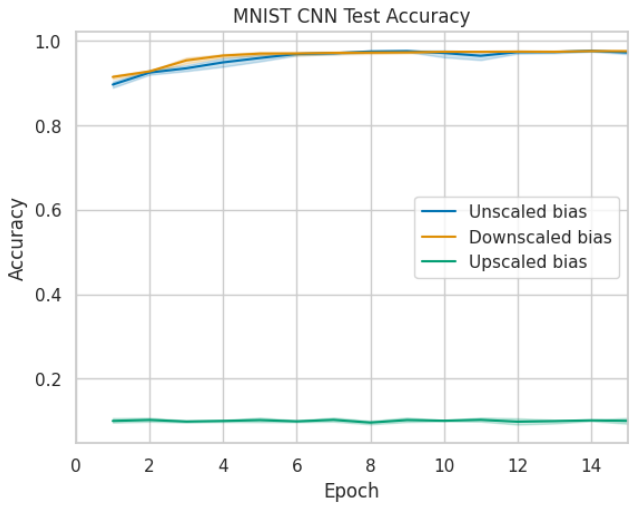

Figure 6: Test accuracy on the MNIST dataset of CNNs initialized by our approach with scaled and unscaled biases.

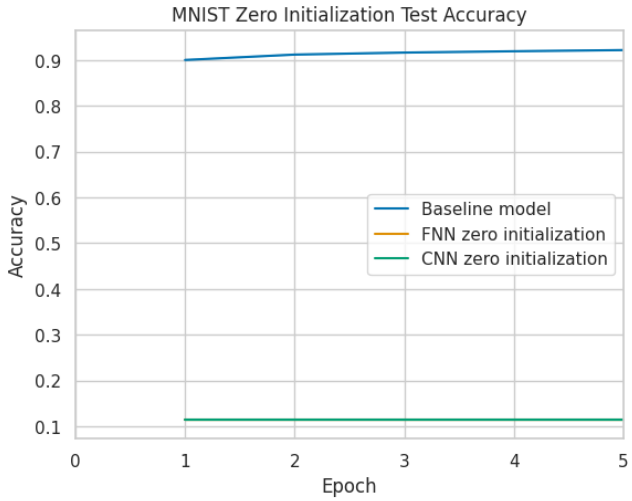

Figure 7: Test accuracy on the MNIST dataset of the baseline model and FNN and CNN initialized by zero initialization.

CIFAR-10 - NNs	
FNN	CNN
	conv 3x3x32
	conv 3x3x64
	max-pool 2x2, stride 2
	batch-norm
	conv 3x3x128
	conv 3x4x128
	max-pool 2x2, stride 2
	batch-norm
	conv 3x3x256
	conv 3x3x256
	max-pool 2x2, stride 2
	batch-norm
dense 3072, 512	
dense 512, 256	dense 4096, 1024
dense 256, 128	dense 1024, 512
dense 128, 10	dense 512, 10

Table 2: CIFAR-10 NNs architecture. Description of the layers used for the NN architectures. Each convolutional and dense layer is followed by an ELU activation, the final layer has no activation

3.2 Experiments on the CIFAR-10 dataset

This section presents the results of the experiments performed on the CIFAR-10 dataset [1]. Also, we compare our results to zero initialization and PyTorch initialization [5].

The CIFAR-10 dataset [1] is a collection of color images containing 10 different classes such as airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. In total, the dataset contains 60,000 images which are divided into 50,000 training and 10,000 test images, each with a resolution of 32x32 pixels.

For our first experiments for the CIFAR-10 dataset, we use a FNN with 4 dense layers as described in Table 2. We trained this model for 15 epochs.

The results in Figure 8 show that the FNN using our initialization approach get close to the reference initialization from Section 2.1 and thus outperforms the baseline model.

In the experiments with CNNs for the CIFAR-10 dataset, we firstly tried to add further layers beside dense and convolutional layers, for this we added max pooling and batch normalization [11] layers. The full architecture of this model is described in Table 2. We trained this architecture for 20 epochs in our experiments.

Figure 9 shows the results of the experiments for the CNN. The CNN outperforms the baseline model like the FNN. Thus, adding pooling and batch normalization layers did not prevent the model from learning with our approach.

Figure 10 shows the results of a FNN and a CNN initialized by zero initial-

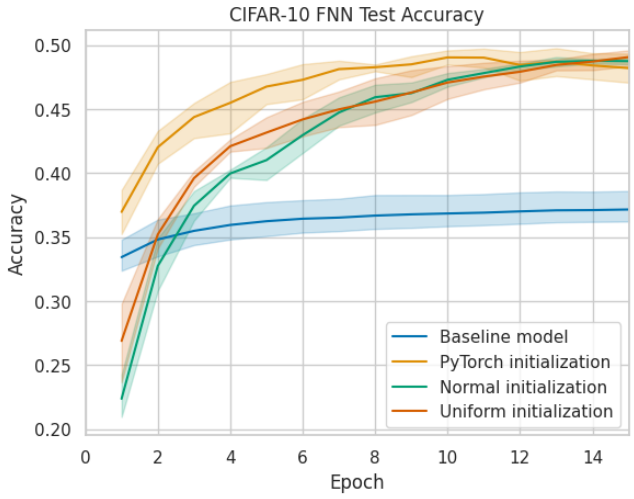

Figure 8: Test accuracy on the CIFAR-10 dataset of the baseline model and FNNs initialized with PyTorch initialization compared to our initialization approach using a normal and uniform distribution.

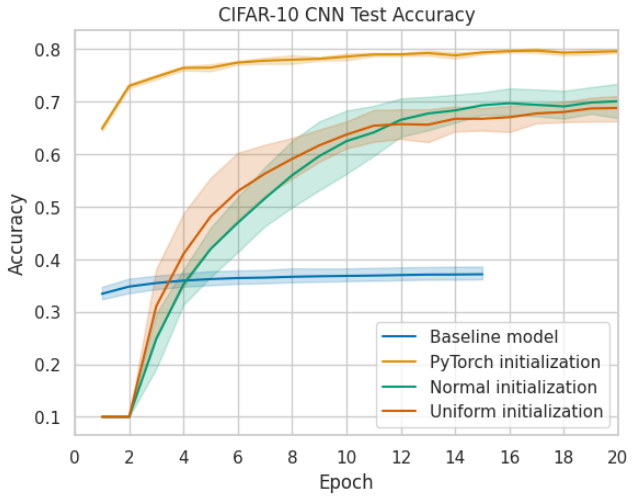

Figure 9: Test accuracy on the CIFAR-10 dataset of the baseline model and CNNs initialized with PyTorch initialization compared to our initialization approach using a normal and uniform distribution.

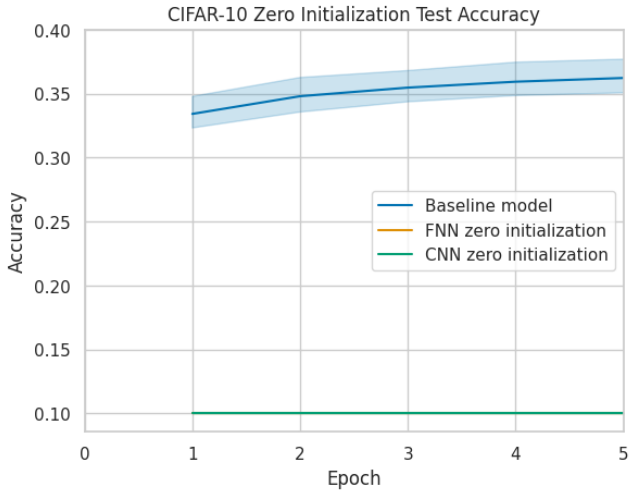

Figure 10: Test accuracy on the CIFAR-10 dataset of the baseline model and FNN and CNN initialized by zero initialization.

ization. Those networks were not able to outperform the baseline model in our experiments.

In order to verify our results for a more complex task, we tested our initialization approach for a ResNet-50 model [8]. For this, we took implementation of this model from [17] because in this work they tested the ResNet-50 architecture for the CIFAR-10 dataset already.

We needed to slightly adopt the taken ResNet-50 model in order to make it work with our initialization approach. Firstly, we added a learnable bias to each of the convolutional layers because these were not enabled in the source code we took. Finally, we also replaced the ReLU [4] activation functions in the implementation by the ELU activation [7] because of the mentioned limitations in Section 2.2. Furthermore, we removed the batch normalization layer after the first convolutional layer in the residual blocks because this layer caused a reduced accuracy with the setup of our experiments.

In the experiments with the ResNet-50 model, we also performed data preprocessing steps. For this, we kept the procedure from [17]. Thus, for training we normalized the values of the input images and randomly cropped a 32x32 with 4 pixel padding from the original images additionally performed random horizontal flips. The test data were only normalized before they were fed to the model.

The results of the experiments with the ResNet-50 model are displayed in Figure 11. Due to the limitation of computational resources, we trained this model for only 30 epochs. Nevertheless, we can see that our approach gets close to the performance of the reference initialization.

In this section, we presented the results of our experiments for the CIFAR-10

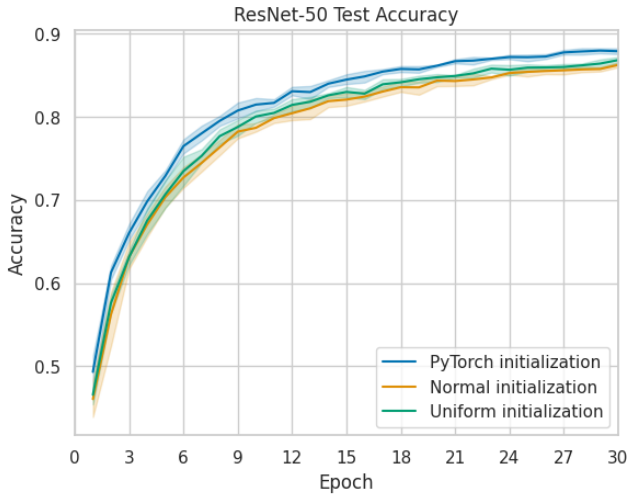

Figure 11: Test accuracy on the CIFAR-10 dataset of the ResNet-50 model initialized using PyTorch initialization compared to our initialization approach using a normal and uniform distribution.

dataset [1]. We showed that our initialization approach was able to outperform the baseline model, whereas the networks initialized by zero initialization were not. Additionally, we ran experiments for a ResNet-50 model [8] to show that our approach also works on a more complex task.

4 Discussion

The reported experiments have explored the ability to train NNs initialized by zero initialization for different datasets and architectures.

The performed experiments show that the symmetry introduced by zero initialization can be overcome by a bias initialization. Thus, this shows that our approach clearly outperforms zero initialization in our experiments.

Despite the fact that we were able to break the symmetry in zero initialized NNs, we still want to highlight that there is a gab in performance between our approach and the PyTorch initialization, which we used for comparison. We did not investigate in the cause of this gap or in how to close it, instead we keep it for future investigations.

Figure 4 and Figure 6 show that large initial biases caused instability during training, whereas small biases did not. This might be caused by an over amplification of the input signals during the forward pass. Thus, those signals might be heavily penalized in the backward pass. Further investigation in this behavior is kept for future research.

Furthermore, we noticed a change in the learning dynamics of models ini-

tialized with our approach compared to the PyTorch initialization in our rolled out experiments. The weights of the layers of the networks initialized with our approach are clearly updated from the output layer to the input layer in the initial phase. Thus, during this phase after each iteration step one layer more is updated, this can be seen in Figure 12. We explain this behavior with the cut-off gradient flow in the backpropagation while the weights are still zero. Thus, a layer can only be updated when the layer after it has been updated already.

5 Conclusion

We have presented an empirical work of an approach of how NNs can overcome the symmetry introduced by zero initialization and reviewed the cause of symmetrically evolving neuron in zero initialized NNs.

The presented results confirm that zero initialized NNs can break their symmetry by randomly initializing the biased. We also showed that altought we broke the symmetry of zero initialized NNs, there is still a gap in performance compared to the initialization method presented in Section 2.1.

The results also highlight the optimization difficulties associated with our approach and large bias initialization. We empirically showed that highly upscaled biased led to instable NNs.

Finally, the experimental results support our conclusion that the symmetry in zero initialized NNs can be broken by randomly initializing the biases of the networks.

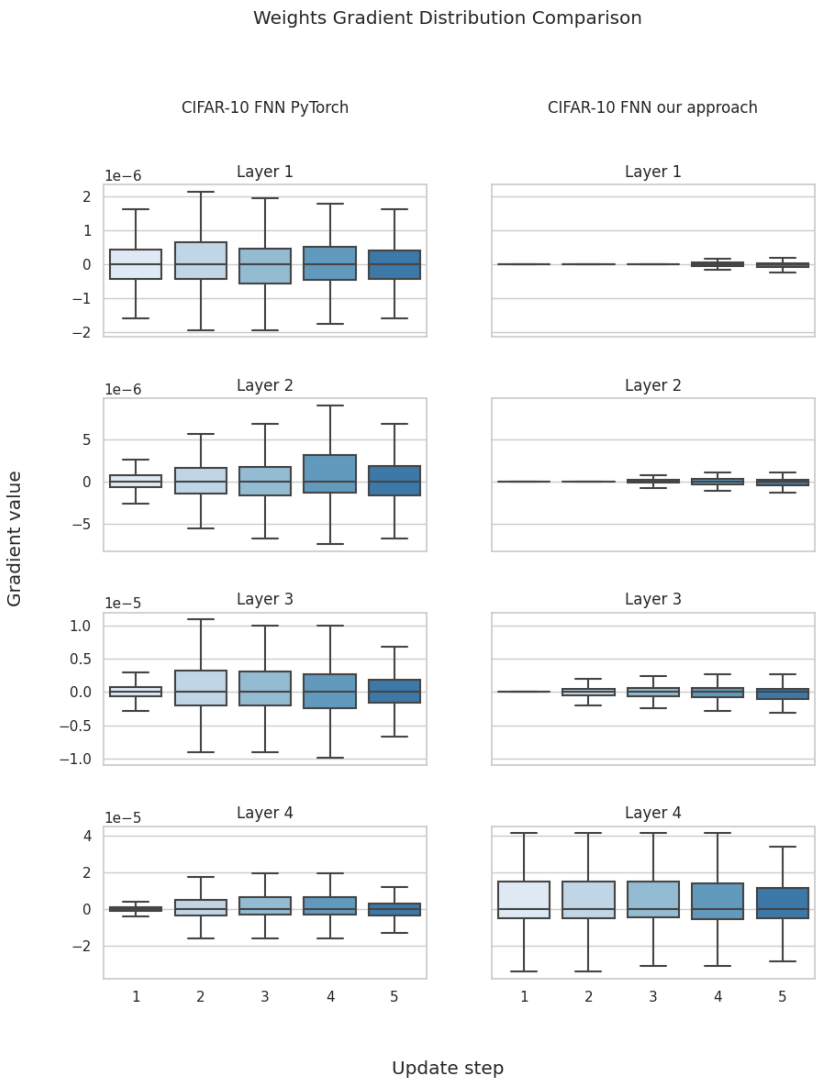

Figure 12: Comparison of the gradient distribution during the first 5 update steps between our CIFAR-10 FNN model initialized by PyTorch initialization and zero initialization with randomly bias values drawn from a standard normal distribution $\mathcal{N}(0, 1)$.

References

[1] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[2] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249–256, Jan. 2010.

[3] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[5] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," *Neural networks: Tricks of the trade: Second edition*, pp. 9–48, 2012.

[6] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. DOI: 10.48550/ARXIV.1412.6980. [Online]. Available: https://arxiv.org/abs/1412.6980.

[7] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, *Fast and accurate deep network learning by exponential linear units (elus)*, 2015. DOI: 10.48550/ARXIV.1511.07289. [Online]. Available: https://arxiv.org/abs/1511.07289.

[8] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. DOI: 10.48550/ARXIV.1512.03385. [Online]. Available: https://arxiv.org/abs/1512.03385.

[9] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, 2015. DOI: 10.48550/ARXIV.1502.01852. [Online]. Available: https://arxiv.org/abs/1502.01852.

[10] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, 2015. DOI: 10.48550/ARXIV.1502.01852. [Online]. Available: https://arxiv.org/abs/1502.01852.

[11] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. DOI: 10.48550/ARXIV.1502.03167. [Online]. Available: https://arxiv.org/abs/1502.03167.

[12] B. Xu, N. Wang, T. Chen, and M. Li, *Empirical evaluation of rectified activations in convolutional network*, 2015. DOI: 10.48550/ARXIV.1505.00853. [Online]. Available: https://arxiv.org/abs/1505.00853.

[13] B. Xu, R. Huang, and M. Li, *Revise saturated activation functions*, 2016. DOI: 10.48550/ARXIV.1602.05980. [Online]. Available: https://arxiv.org/abs/1602.05980.

[14] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," 2017. DOI: 10.48550/ARXIV.1706.02515. [Online]. Available: https://arxiv.org/abs/1706.02515.

[15] H. Li, M. Krček, and G. Perin, "A comparison of weight initializers in deep learning-based side-channel analysis," in *Applied Cryptography and Network Security Workshops*, J. Zhou, M. Conti, C. M. Ahmed, *et al.*, Eds., Cham: Springer International Publishing, 2020, pp. 126–143, ISBN: 978-3-030-61638-0.

[16] M. Skorski, A. Temperoni, and M. Theobald, *Revisiting initialization of neural networks*, 2020. DOI: 10.48550/ARXIV.2004.09506. [Online]. Available: https://arxiv.org/abs/2004.09506.

[17] [Online]. Available: https://github.com/kuangliu/pytorch-cifar.