Learning Dynamics: Zero Initialization of Neural Networks

Christopher Schicho K11943526

December 2022

1 Introduction

Initializing the parameters of a neural network can be critical to the model's ultimate performance. Initializing a neural network's parameters with 0 leads to symmetrically evolving neurons throughout training. Thus, such neurons will learn the same features during training. This limits the capability of a neural network dramatically. However, in the following report I show that one can still initialize the weights of a neural network with 0 and overcome the symmetry by initializing the bias of the network randomly. For this I trained different neural network architectures on the MNIST and CIFAR-10 datasets, to show that they get close to the models initialized by the methods used by PyTorch. Furthermore, this procedure reverts the dynamics in the initial training phase which results in high-level layers being updated first.

2 Background

Initializing the weights of a neural network with 0 requires some minor adoptions with respect to the used activation function since some might introduce some limitations when dealing such initialization. This initialization scheme has effects on the back propagation as well and, therefore, also on the learning dynamics of such neural networks. The following provides the theoretical foundation for the experiments mentioned in 3.

2.1 Parameter initialization

The initialization of network parameters has a significant influence on the performance and effectiveness of training. Therefore, there exist many different methods and approaches on how to effectively initialize those parameters. For simplicity the following experiments use none of them, instead I compare the results of the experiments to the standard initialization done by the framework PyTorch.

By default the parameters for linear layers are in PyTorch initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{H_{in}}$ and H_{in} = input features. The convolutional layers are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{in} * k_w * k_h}$, C_{in} = input channels, k_w = kernel width and k_h =kernel height.

The presented approach, however, initializes the weights of a neural network with 0 and initialized the bias randomly either from a normal distribution $\mathcal{N}(0, 1)$ or uniform distribution $[0, 1)$.

2.2 Effect of activation function

Choosing the right activation function for the hidden layers is essential when we want to reach good results with weights initialized with 0. Although ReLU (Rectified Linear Unit) is the default option nowadays, it might introduce some limitation in this setup. The drawback of ReLU is that they cannot learn on examples for which their activation is zero. This might be already the case for some neurons after the initialization described in 2.1 or by a parameter update step. Once the activation for a neuron is 0 during the forward propagation, this will make the gradient flow in the backward propagation through this neuron always be 0. Thus this neuron is considered to be a "dead neuron" and will never be updated again during training. In case a whole layer would suffer from this problem, this would even cut the gradient to previous layers. Thus, previous layers could not be trained anymore.

Due to the described limitations of the ReLU activation function I have decided to use the ELU (Exponential Linear Unit) activation function because the gradient of this activation function actually never reaches 0. Thus this resolved the mentioned drawback of ReLU.

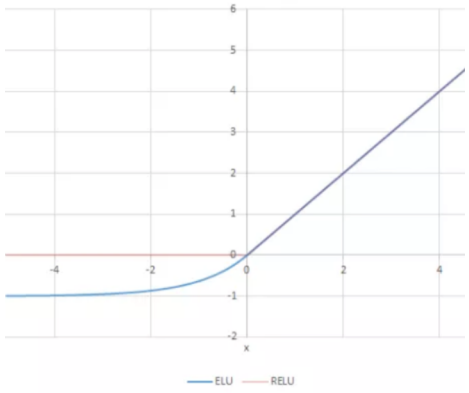


Figure 1: ReLU and ELU activation function

2.3 Effect on back propagation

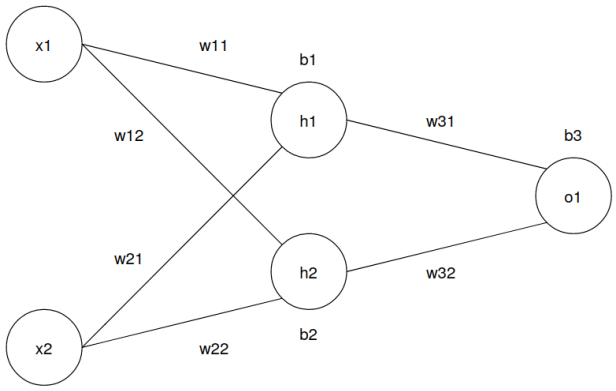


Figure 2: Neural network

Consider the neural network from 2. For the following examples assume w_{11}, w_{12}, $w_{21}, w_{22}, w_{31}, w_{32}$ are the weights and b_1, b_2, b_3 are the biases of the network and let $f_a(x)$ be the any activation function.

For demonstrating that randomly initialized bias can break the symmetry, introduced by initializing the weights of the neural network with 0 assume that we initialize the bias with 0, as well. When forward propagating the output (x_1, x_2) the output of each neuron of the hidden layer will be:

$$s_1 = w_{11}x_1 + w_{21}x_2 + b_1 \tag{1}$$

$$a_1 = f_a(s_1) \tag{2}$$

$$s_2 = w_{12}x_1 + w_{22}x_2 + b_2 \tag{3}$$

$$a_2 = f_a(s_2) \tag{4}$$

Since all weights and biases are initialized with with 0, $s_1 = s_2$ regardless of the input. This implies $a_1 = a_2$. The output will be calculated as follows:

$$f_a(w_{31}s_1 + w_{32}s_2 + b_3) \tag{5}$$

Thus, the neurons in each layer will have identical influence on the loss of the output. This leads to identical gradients, thus, throughout training those neurons will evolve symmetrically, which means that the individual neuron cannot learn different features.

In order to overcome this symmetry which prevents the network from using it's full potential, assume now that we still initialize the weights with 0 but

we initialize the bias with values drawn from a random distribution. Thus, let b_1, b_2, b_3 be arbitrary but fixed values from this random distribution. When forward propagating the output (x_1, x_2) with those initialized values we can use the same equations as above. However, note that in general $s_1 \neq s_2$ because since those values are sampled from a random distribution and, therefore, the chance of getting equal values is negligible. Thus, each neuron contributes differently to the loss of the output signal. This then leads to different gradients and thus eventually to different updates of the parameters of each neuron. Because each neuron will now get a different parameter update each neuron can learn a different feature compared to the other neurons in the same layer now. Therefore, this shows that randomly initialized bias can break the symmetry introduced by weight initialized with 0.

3 Experiments

To show that the theory presented in 2 also holds for real applications of neural networks this section will present the results of the experiments carried out.

3.1 Implementation Details

For the following experiments I used different network architectures and datasets. As Baseline I used a logistic regression model.

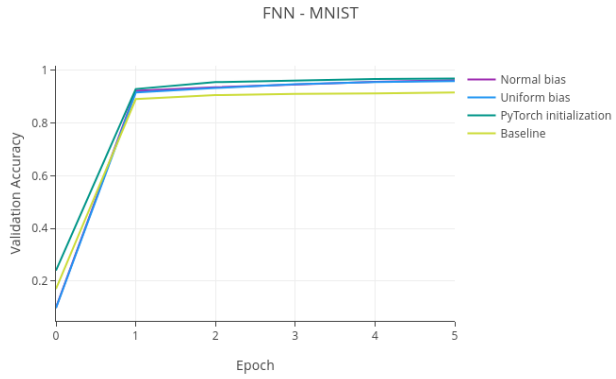
For the experiments on the MNIST dataset I trained a FNN (Feedforward Neural Network) with 1 hidden layers and a CNN (Convolutional Neural Network) with 2 convolutional layers and 1 hidden linear layers. In order to gain some intuition and insights in the change of learning behavior those models were trained with the PyTorch initialization, weights and bias initialized with 0 and weights initialized with 0 and bias sampled from a normal or uniform distribution. Furthermore, I experimented with down and up scaling the bias. Those models were trained for 5 epochs.

For the experiments on the CIFAR-10 dataset I trained a FNN with 3 hidden layers and a CNN with 6 convolutional layers and 4 hidden linear layers. Furthermore, in order to show that this initialization scheme also works outside of this experiment framework I trained a ResNet-50 model using the presented initialization. These models were trained using the PyTorch initialization and initializing weights with 0 and the bias sampled from a normal or uniform distribution. Those models were trained for 10 epochs and the ResNet-50 models were trained for 20 epochs.

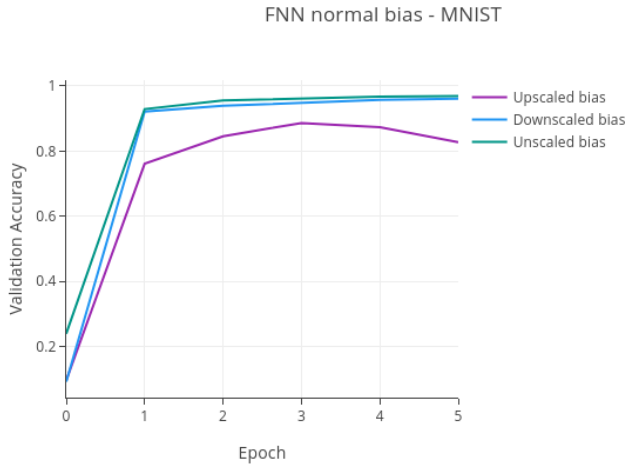
Each model was optimized with an Adam optimizer using a learning rate of 0.001 and the performance during training was monitored using a validation dataset in order to realize overfitting.

3.2 Experiments on the MNIST dataset

The following chart shows the results of the FNN network with bias randomly initialized from a normal and uniform distribution and the results of a network initialized by the initialization which is used by PyTorch internally. This plot shows that the performance between this networks is almost the same.

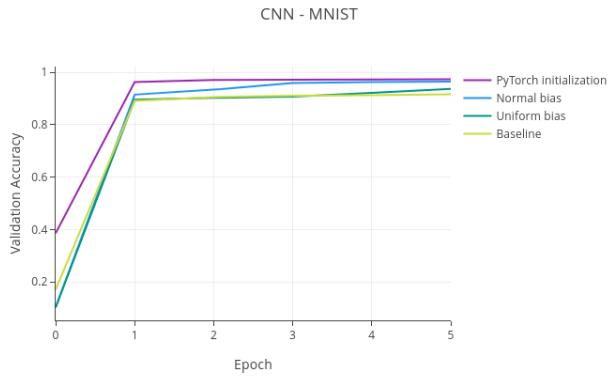


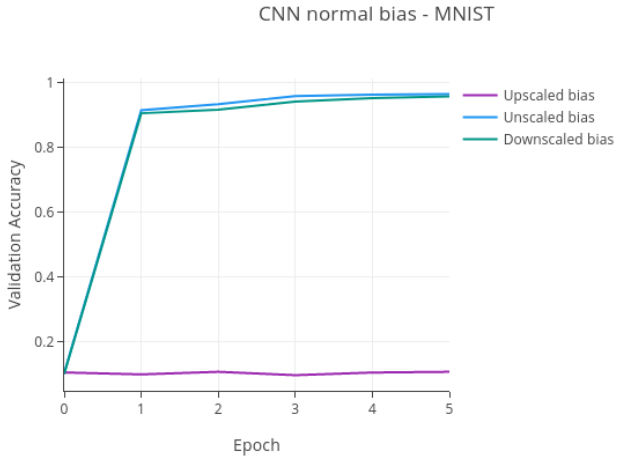
The next plot shows the result of the FNN network with scaled bias drawn from a normal distributions. This shows that the presented approach works best with small biases since the network with up scaled biases perform worse compared to the down scaled and unscaled ones. The used scaling factors are 0.01 for downscaling and 100 for upscaling. The results of the uniform distribution are omitted as those show the same picture.



The following two plots show the same experiments with a CNN network.

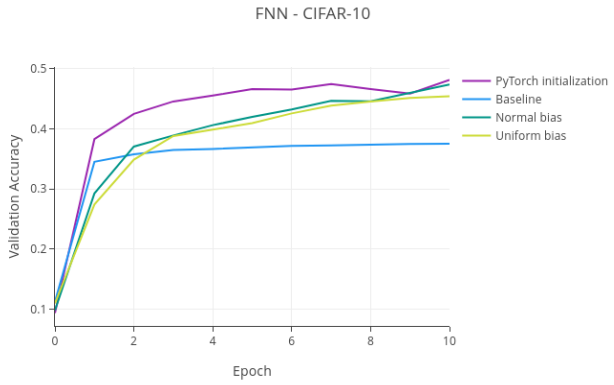
The observations are similar to the ones of the FNN, except those networks with up scaled biases are not able to learn within this framework.

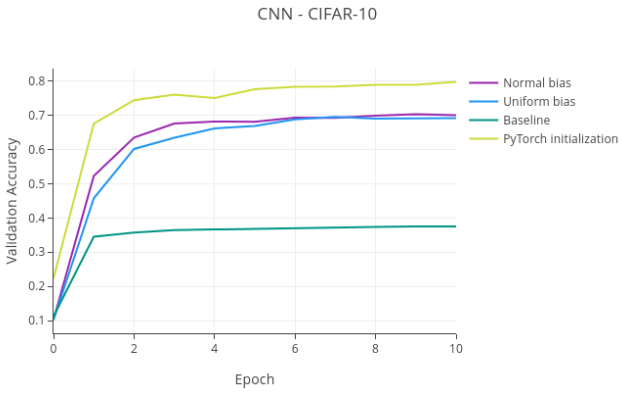




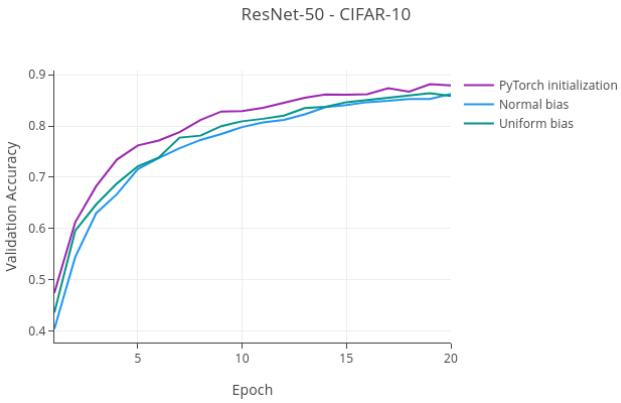
3.3 Experiments on the CIFAR-10 dataset

This section presents the results of more complex models trained on the CIFAR-10 dataset. It is notable that with this deeper networks and the more complex data there is a wider gap between the performance of the Pytorch initialization and the presented approach. The models with initial weights set to 0 perform slightly worse.



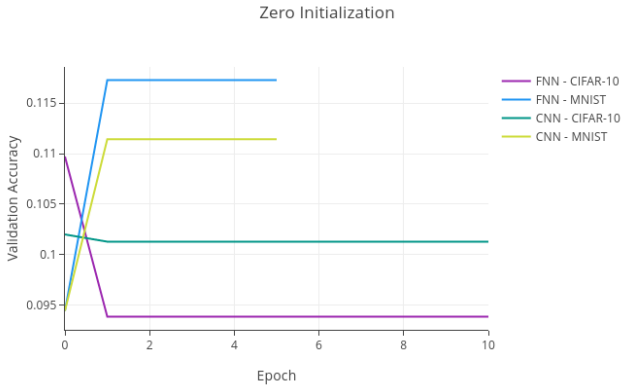


In order to show that this approach not only holds in an experimental setup the following plots show the results of a ResNet-50 model which shows that the results presented above also hold for more sophisticated models.

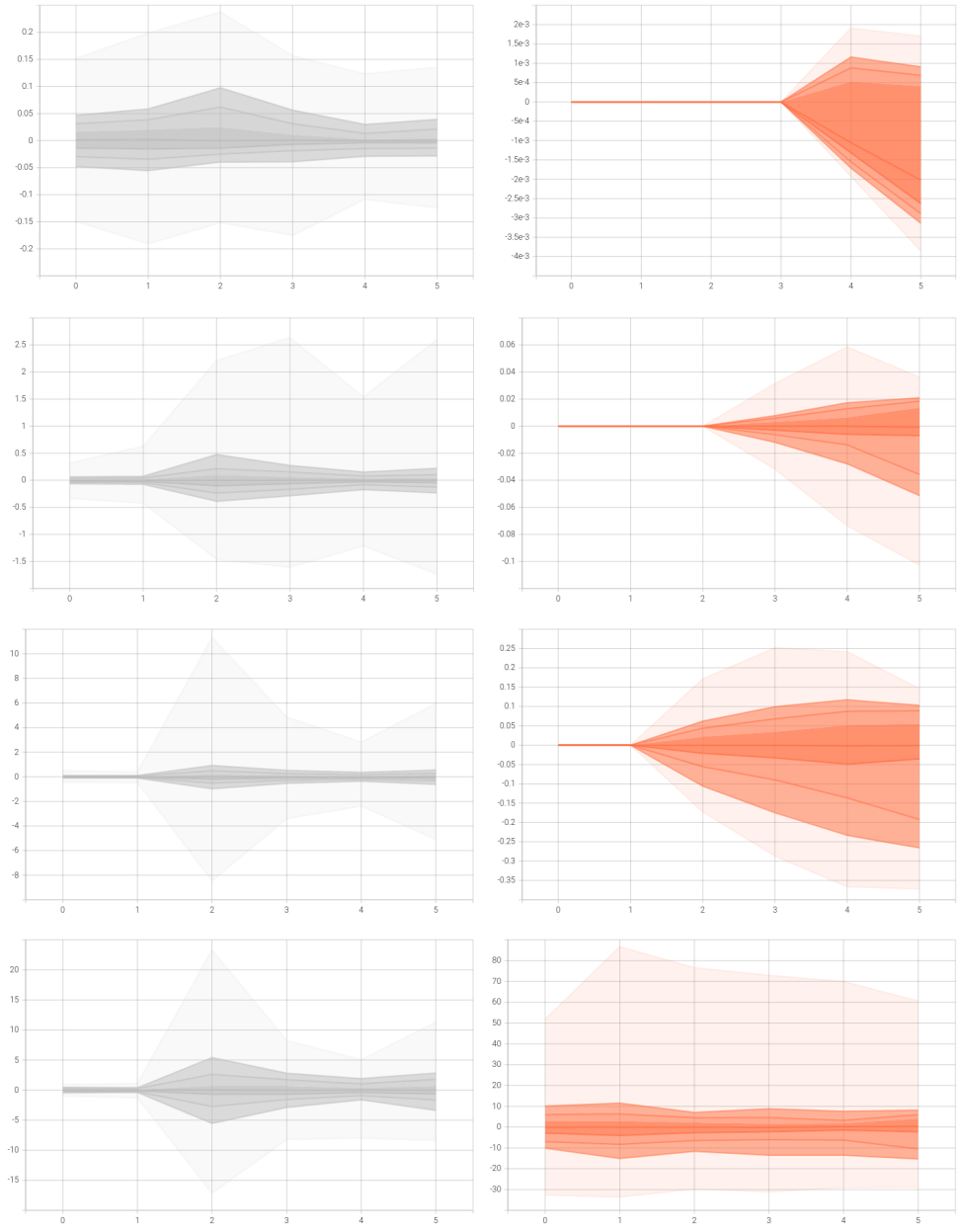


4 Discussion

The rolled-out experiments show that a randomly initialized bias can break the symmetry in neural networks introduced by initializing the weights with 0. Since neurons in layers of networks in which this symmetry described in 2.3 occurs cannot learn different features, their performance will not be significantly better than a random predictor which just predicts the most common class in the dataset as the following plot shows.



It is also notable that there is a change in the learning dynamics when initializing neural networks with the presented approach. Since the gradient flow during back propagation is cut off while the weights of the layer are still 0, this reverses the learning. Thus, the networks updates the layer from the highest level to the lowest label. This can be shown by checking the gradients during the first few update steps, which is shown in the following plot. In the plot below you can see the gradient distribution of the weights from the input layer to the output layer.



On the left one can see the gradients of a model initialized with the PyTorch initialization and on the right the gradients of a model initialized with the presented method. This plot clearly shows how the layers gradually get updated output to input layer.

5 Conclusion

As we can see in the results above I managed it to train neural networks with weights initialized by 0 by initializing the bias randomly. Those networks even perform relatively good, however, it is notable that there is still a little gap in performance between the PyTorch initialization of a network and the presented approach. The experiment with the ResNet-50 model on the CIFAR-10 dataset even shows that this holds true for more sophisticated models and real benchmarks. Furthermore, the analysis of the gradients during training shows that we are able to reverse the learning. Networks initialized with the presented scheme actually update high-level layers before low-level layers. So there is a changed in learning dynamics happening with this approach.

The further investigations of the change in the learning dynamics and the gap in performance are kept for future work.