Weight Rescaling: Applying Initialization Strategies During Training

Lukas Niehaus
Institute of Cognitive Science
University of Osnabrück
Osnabrück, Germany
luniehaus@uos.de

Ulf Krumnack
Institute of Cognitive Science
University of Osnabrück
Osnabrück, Germany
krumnack@uos.de

Gunther Heidemann
Institute of Cognitive Science
University of Osnabrück
Osnabrück, Germany
gheidema@uos.de

***Abstract*— The training success of deep learning is known to depend on the initial statistics of neural network parameters. Various strategies have been developed to determine suitable mean and standard deviation for weight distributions based on network architecture. However, during training, weights often diverge from their initial scale. This paper introduces the novel concept of *weight rescaling*, which enforces weights to remain within their initial regime throughout the training process. It is demonstrated that *weight rescaling* serves as an effective regularization method, reducing overfitting and stabilizing training while improving neural network performance. The approach rescales weight vector magnitudes to match the initialization methods' conditions without altering their direction. It exhibits minimal memory usage, is lightweight on computational resources and demonstrates comparable results to *weight decay*, but without introducing additional hyperparameters as it leverages architectural information. Empirical testing shows improved performance across various architectures, even when combined with additional regularization methods like *dropout* in AlexNet and *batch normalization* in ResNet-50. The effectiveness of *weight rescaling* is further supported by a thorough statistical evaluation.**

I. INTRODUCTION

Regularization is used in machine learning, particularly in training neural networks, to prevent overfitting and improve the generalizability of models. It employs the idea that adding some constraints or penalties on model parameters can help avoid overfitting and lead to better performance on unseen data. Some common regularization techniques include *weight decay* or specifically L2-regularization, which adds a penalty term proportional to the squared magnitude of the weights during the optimization process [5]. It encourages smaller weight values and helps prevent overfitting by limiting the influence of individual features in the model. Another method is *dropout*, where some neurons are randomly "dropped out" (set to zero) during training, forcing other neurons to learn more efficiently. This is applied at the layer level and can be used with any neural network architecture [20]. *Batch normalization* is another common regularization method, which helps stabilize training by making the inputs of each neuron have a similar distribution across different batches of data [10]. By doing so, it reduces the need for learning large weights and prevents overfitting. Early stopping follows the concept to stop training when the performance degrades. The method uses a validation set to detect overfitting. It helps prevent models from being trained too long and memorizing noise in the data rather than capturing useful patterns [15].

In this paper, we introduce a novel concept called *weight rescaling* that combines findings from initialization strategies and applies them during training to achieve regularization effects. The study begins by discussing various regularization methods, weight scaling approaches and initialization strategies, followed by an exploration of why activation variance changes in neural networks during the learning process. Subsequently, the concept of *weight rescaling* is introduced along with its application methodology. Following this, we detail our experimental setup and evaluation metrics used to assess the proposed technique's effectiveness. Then the empirical results of the experiments demonstrate the efficiency and potential benefits of incorporating *weight rescaling* into neural network training processes, backed up by a statistical analysis. The final section presents a conclusion of the findings and gives an outlook into further research questions.

The experiments encompass various architectures such as Multi Layer Perceptron (MLP) [8], LeNet-5 [13], AlexNet [12], and ResNet [6]. For the initialization methods we employ Xavier [4] and Kaiming [7] initialization. Furthermore, we compare the performance of *weight rescaling* to *weight decay*, assess its effectiveness on *relu* activations [2] and *tanh* activations [8], and examine the efficacy when other regularization methods are already applied, including *dropout* [20] and *batch normalization* [10].

II. WEIGHTS

A well-designed initialization method ensures that network weights are initialized in a manner preventing both exponential growth and shrinkage during training. In this context, utilizing *relu* as an activation function, we assume that increasing weights cause larger activations, which subsequently results in growing gradients, which then leads to self reinforcing effects of growing or shrinking variance of the weights, activations and gradients. As training progresses, neuron weights tend to develop Gaussian-like distributions with nontrivial correlations between components [17]. The majority of neurons approach zero values, while a few become significantly large. Concurrently, although the mean of weight distributions deviates around zero, the standard deviation increases throughout the learning process.

This behavior can be observed in Figure 1, where a three-layer MLP with the widths' 32, 32, and 10 neurons is trained on the CIFAR-10 dataset. The network was trained for 100 epochs and initialized by sampling the weights from a normal distribution with a standard deviation accoording to Kaiming [7] initialization. The results demonstrate that the standard deviation of weights in each layer increases throughout training, such as in the first layer where it grows more than five times its original size from 0.03 to 0.17.

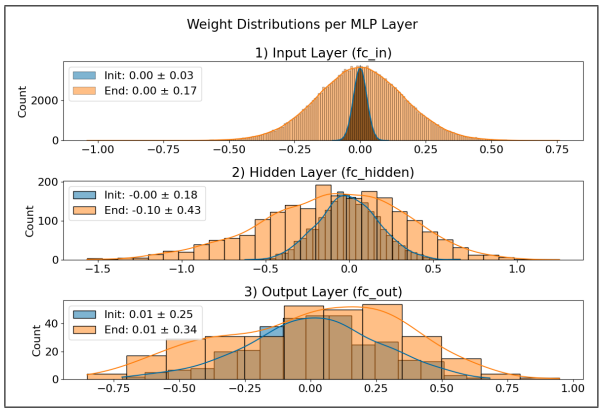

Fig. 1: Weight Distributions for each layer of an MLP at initialization in blue and after training for 100 epochs in orange.

This phenomenon can lead to high fluctuations of the activations in a neural network. Classical regularization techniques, such as *weight decay*, counteract this issue by incorporating weight size into the loss function. By penalizing large weights, these methods help prevent their development.

The question that arises is whether initialization strategies can be employed during training to avoid the fluctuations and create a regularizing effect. The experiments in this study show that repeatedly rescaling the weight size according to the distribution provided by an initialization method results in improved performance and generates a regularizing effect.

III. Related Work

A. Dropout

Dropout was introduced in [20] and is a regularization technique used to prevent overfitting in deep learning models. It involves randomly dropping out, or setting to zero, some of the neurons in the network during training. This effectively reduces the capacity of the neural network and forces it to learn redundant representations, subsequently reducing the risk of overfitting. This forces the remaining neurons to learn more robust and generalizable representations of the data, resulting in improved performance on unseen test data. *Dropout* has been shown to be effective in a variety of tasks, including image classification, natural language processing, and speech recognition. During testing or prediction, all neurons are active.

B. Batch Normalization

Batch normalization was published in [10] and is a now widely used technique for improving the stability and performance of deep neural networks by reducing internal covariate shift. The authors demonstrate that *batch normalization* accelerates training by several orders of magnitude, allowing deeper networks with more parameters to be trained effectively. Furthermore, they show that *batch normalization* improves generalization performance and robustness against overfitting. *Batch normalization* is implemented as a layer within a neural network, positioned between existing layers to modify the input data prior to further processing. A *batch normalization* layer introduces additional parameters to the model in the form of scale and shift factors, which are learned during the training process to optimize the transformation applied to each input mini-batch. Recent research has shown that the effectiveness of *batch normalization* might not primarily be due to mitigating internal covariate shift, but rather through its ability to smooth the optimization landscape [19] [3].

Batch normalization relies on mini-batches as its foundation for calculating the mean and standard deviation of input data, which effectiveness is highly dependent on the size of the batches as a larger batch size leads to more accurate statistical calculations. Large batches quickly exceed the available memory capacity, especially in applications that depend on high dimensional data like Computer Vision tasks with high resolution images. On top of the growing memory requirements associated with mini-batches and *batch normalization*, there is also an extra computational overhead due to the need to calculate the mean and standard deviation for each input channel or feature map separately. The additional trainable parameters increases the number of operations required to compute the activations of each layer. Overall, while *batch normalization* can improve the performance of neural networks, it is important to consider the memory and computational requirements associated with this technique when designing and training deep learning models.

C. Weight Decay (WD)

Weight decay is a widely adopted regularization technique employed within the domain of machine learning, particularly in neural network models. This method aims to prevent overfitting by penalizing model parameters with higher magnitudes during the training process. By imposing this constraint on the weights, the model learns more generalizable and robust features while minimizing the risk of overfitting to specific data points or patterns. Consequently, *weight decay* enhances the overall performance and generalization capabilities of machine learning models in various applications [5]. In the following, θ denotes the parameters of the neural network, which include the weights w and biases b. $\mathcal{L}_{\text{data}}(\theta)$ denotes the loss function used to optimize the network, like cross-entropy or mean squared error. $R(\theta)$ is the regularization term and the final loss function $\mathcal{L}(\theta)$ is

obtained by combining these as follows:

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{data}}(\theta) + \lambda * R(\theta) \tag{1}$$

λ is a hyperparameter that has to be choosen carefully when applying *weight decay*. A λ that is too small does not prevent overfitting and a λ that is too big leads to underfitting. For this paper, we use the *weight decay* implementation of the stochastic gradient descend optimizer from the deep learning framework Pytorch, which utilizes a form of L2-regularization. One disadvantage of *weight decay* is the need for a λ hyperparameter, which is chosen by the user.

D. Generalization error

The generalization error is closely related to overfitting or underfitting, which refers to the difference between the performance of a neural network on the training data and its actual performance on unseen test data. Overfitting occurs when a neural network is too complex or there is not enough data to represent the underlying data dirstribution. This results in the model learning the noise and idiosyncrasies present in the training set rather than capturing the underlying patterns that are useful for making predictions on new, yet unseen test data. In this case, the model's performance on the training dataset is high but fails to generalize well when applied to new input. Underfitting, on the other hand, occurs when a neural network is too simple or has not been trained enough. This results in the model failing to capture essential patterns present in the data and leads to poor performance both during training and testing phases. The amount of overfitting can be tested by cross-validation, where the whole dataset $\mathcal{D}$ is split into three nonoverlapping subsets $\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{val}}$, $\mathcal{D}_{\text{test}}$, where $\mathcal{D}_{\text{train}}$ is used for training, $\mathcal{D}_{\text{val}}$ for model validation and $\mathcal{D}_{\text{test}}$ for testing [17].

The generalization error is represented by $\mathcal{E}$. The loss is represented by $\mathcal{L}$ and is calculated for both subsets $\mathcal{D}_{\text{test}}$ and $\mathcal{D}_{\text{train}}$. The generalization error then is calculated by subtracting the loss of the training data $\mathcal{D}_{\text{train}}$ from the loss of the testing data $\mathcal{D}_{\text{test}}$.

$$\mathcal{E} := \mathcal{L}(\mathcal{D}_{\text{test}}) - \mathcal{L}(\mathcal{D}_{\text{train}}) \tag{2}$$

Figure 2 shows the generalization error and compares no regularization aginst *weight decay* with various λ values and our new method of *weight resclaing*. The base/none experiment without regularization shows the highest generalization error but this was expected since it starts to overfit around epoch 5. The line with *weight decay* and $\lambda = 1e^{-4}$ still exhibits significant overfitting, as the chosen λ is too small. Conversely, the line for $\lambda = 1e^{-2}$ shows that the value is too large, resulting in excessive penalization of big weights, which leads to underfitting. The line for *weight decay* with a well-chosen $\lambda = 1e^{-3}$, demonstrates a balance between overfitting and underfitting while achieving high accuracies. To this, the line for *weight rescaling* shows similar behavior with a slightly lower generalization error.

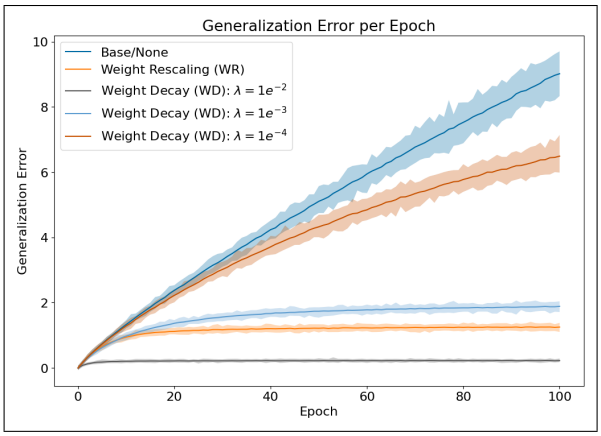

Fig. 2: The y-axis shows the generalization error along the epochs on the x-axis. The model is a three layer MLP trained on CIFAR-10. The line in the middle shows the mean generalization error for each experiment and the area shows the min and max results for 100 repeated runs at the specific epoch.

E. Weight Normalization (WN)

The method of *weight normalization* as proposed by Salimans et al. [18] splits the weight vector w into two trainable parameters for direction v and magnitude g., where v is a k-dimensional vector, with the same dimensions as the weights w and g is a scalar. Dividing v with its norm $||v||$ and multiplying it with g allows to train the magnitude g independent from the direction v.

$$w = \frac{v}{||v||} g \tag{3}$$

weight normalization differs from *weight rescaling* as it trains the magnitude for g directly, while in *weight rescaling* the magnitude is rescaled periodically to the value defined by the initialization method. Additionally, *weight normalization* is incorporated into the optimization process, while *weight rescaling* is executed once per epoch. The authors of *weight normalization* report a speed up in the convergence during training.

F. Weight ReScaling (WRS) by Liu et al.

Liu et al [14] proposed a method which rescales the weight norm to the unit norm after a specified number of optimization steps for each layer. The authors report increased performance by their method.

$$w \leftarrow \frac{w}{||w||} \tag{4}$$

Unfortunately we were only made aware of this method and its name during the final phase of creating this paper. When we refer to this method we use *Weight ReScaling (WRS) by Liu et al.* and when we refer to our method we use *weight rescaling*. Our method rescales the weights periodically during training just like this method. However, *weight rescaling* does not scale the weight norm to the unit norm, but to the scale according to the chosen initialization

method. *Weight ReScaling (WRS) by Liu et al.* is proposed as an extension to *batch normalization*, while our method of *weight rescaling* works independent of whether other regularization methods are used.

G. Initialization

The goal of initialization is to set the neural network in a trainable state. This includes breaking parameter symmetry and establishing a descent flow of activations and gradients. parameter symmetry is the concept, that if two neurons get the same input, the same activation function and the same initial parameters, then a deterministic learning algorithm would always update the two neurons the same in the same way, which hinders the learning process. If all parameters are initalized with the same value, the network behaves like it only consists of one neuron. To break parameter symmetry we initialize the parameters of a neural network by a random distribution. The inputs for a neuron can be described by random variables, which are summed up. For the outcoming variance it holds that the variance of the sum is the summed up variance. Xavier et al. show in [4] that the variance of a layer at initialization should be 1 to avoid exponentially increasing or decreasing variance through the layers. This means that the variance of each input should be at $1/n$, where n is the number of input features (not weights) coming into the neuron. For the first layer in a neural network, $n^{(1)}$ is the number of input features given by the data. We use $\ell \in L = \{1, \dots, L\}$ as layer index for a network with L layers to denote $n^{(\ell)}$. If the layers are independent and identically distributed (i.i.d) with a zero mean, having a variance of 1 is helpful, since the resulting variance is a product. If each layer variance is 1, the resulting product is 1 as well. With these findings Xavier et al. proposed [4] an initialization method that is based on sampling initial weights from a uniform distribution with suitable variance. In their experiments they use a uniform distribution

$$w \sim U\left[-\sigma_{\text{init}}\sqrt{3}, \sigma_{\text{init}}\sqrt{3}\right] \tag{5}$$

with the standard deviation σ_{init} given by

$$\sigma^{(l)}_{\text{Xavier}} := \sqrt{\frac{2}{n^{(\ell)} + n^{(\ell+1)}}} \tag{6}$$

Xavier initialization was designed with a *linear* activation function in mind. However, a neural network with *linear* activation functions can only capture linear relationships and a multi-layer network reduces into a single layer. With nonlinear activation functions the neural network is able to capture nonlinear relationships and multiple layers can be stacked on top of each other. To introduce a nonlinear activation function Xavier et al. [4] use *tanh* in their experiments, since it is zero-centered like a *linear*.

Pytorch [16] uses an optional $gain$ scaling factor to multiply with $\sigma^{(l)}_{\text{Xavier}}$, which Tensorflow [1] does not. The scaling factor depends on the activation function, which is 1 when ignored, $\frac{5}{3}$ with *tanh* and $\sqrt{2}$ with *relu*.

The authors of the Kaiming initialization develop their method in [7] specifically for the nonzero-centered *relu* activation function. Pytorch uses their method as default and samples from a uniform distribution, where the boundaries are defined by (5) with the standard deviation σ_{init} given by

$$\sigma^{(l)}_{\text{Kaiming}} := \frac{\text{gain}}{\sqrt{n^{(\ell)}}} \tag{7}$$

The popularity of the Kaiming initialization method nowadays is mainly due to the populatity of *relu* as a nonlinear activation function in neural networks. For our experiments we initialize from a normal distribution with zero mean and the standard deviation given by either $\sigma^{(l)}_{\text{Xavier}}$ or $\sigma^{(l)}_{\text{Kaiming}}$ for the weights and setting the biases to zero.

H. Theory

We base the theory on the work of [4] and [7]. The pre-activation $p^{(\ell)}$ for one neuron of a dense layer is defined as:

$$p^{(\ell)} = w^{(\ell)} x^{(\ell)} + b^{(\ell)}, \tag{8}$$

where $w^{(\ell)}$ represents a $n^{(\ell)} \times n^{(\ell+1)}$ dimensional weight matrix. At initialization, each element in $w^{(\ell)}$ represents a random variable, which is independent and identically distributed (i.i.d.). $x^{(\ell)}$ is the $n^{(\ell)}$-dimensional activation of the previous layer, which we assume to be i.i.d. as in [4]. $w^{(\ell)}$ and $x^{(\ell)}$ are independent of each other. $b^{(\ell)}$ is the bias, which is 0 at initialization and is thus ignored for the caluculation of the variance. By applying the activation function ϕ to the pre-activations $p^{(\ell)}$, we obtain the layer output activations $y^{(\ell)} = \phi(p^{(\ell)})$ that serve as inputs for the next layer $x^{(\ell+1)} = y^{(\ell)}$.

The authors of [7] show that the resulting output variance for a neural network using the *relu* activation function at initialization is

$$\text{Var}\left[y^{(L)}\right] = \text{Var}\left[y^{(1)}\right] \left(\prod_{\ell=2}^{L} \frac{n^{(\ell)}}{2} \text{Var}\left[w^{(\ell)}\right]\right) \tag{9}$$

We discuss this relation under the conditions that arise during training. (9) relies on the elements in $w^{(\ell)}$ being i.i.d. with zero mean, which is true for initialization, since we sample the elements of $w^{(\ell)}$ from a random normal distribution with zero mean. Furthermore, $w^{(\ell)}$ is independent to each other layer. However, during training, the elements develop correlations, which is why the variance through the network can not be calculated as a product of the variances of the weights anymore: $\text{Var}[w^{(\ell)} w^{(\ell+1)}] \neq \text{Var}[w^{(\ell)}]\text{Var}[w^{(\ell+1)}]$. This means that the variance for $\text{Var}[y^L]$ is not the product of the weight variances anymore like shown in (9). Furthermore, their mean values deviate from zero. The definition for $\text{Var}[y^L]$ becomes more dependent on the development of the random variables, since their variance becomes dependent on their mean and covariances. The fact that $b^{(\ell)}$ could be ignored at initialization is not the case for training, since their mean, variance and covariance changes as well, additionally influencing the variance calculation.

IV. Weight Rescaling (WR)

We assume that $\mathrm{Var}[y^L]$ becomes more volatile through training and the variance per layer becomes more and more chaotic as a self reinforcing mechanism. One of the goals of initialization stated in [4] is for the variance to be the same throughout each layer. We take this idea and apply it to the training process, where the goal for the variance is to stay approximately the same during training for each layer by

$$\mathrm{Var}(y_e^{(\ell)}) \approx \mathrm{Var}(y_{e+1}^{(\ell)}) \quad \ell \in L, \quad \text{for } e = e_1, \ldots, E-1, \tag{10}$$

where E denotes the total amount of epochs a network is trained. The optimal variance of a trained neural network with correlating weights remains an open research question, which is why we can not enforce it for now. However, we introduce *weight rescaling* as a technique, by which we do not determine the optimal variance of the trained neural network but rescale the weights $w^{(\ell)}$ to the initialization variance after each epoch. With this, the development of the variance is limited by the amount of data used in an epoch and we introduce a form of upper/lower boundary for the variance.

Weight rescaling sets the magnitude of the weights of a neural network by standardizing the weights and then scaling them according to a initialization method. Instead of training parameters like in *batch normalization*, we take the knowledge gained from initialization and rescale the weights to the standard deviation σ_{init}, which can be $\sigma^{(l)}_{\text{Xavier}}$, $\sigma^{(l)}_{\text{Kaiming}}$ or another initialization method. When interpreted as vectors, weights $w^{(\ell)}$ can be divided into their direction and magnitude. We assume that the orientation contains the structural information of a given feature, while the magnitude describes their descriptive strength. We do not want to change the learned direction but only their magnitudes. In some sense, by enforcing the standard deviation to scale back we can re-initialize the weights but with a better internal structure. The first step is to standardize the weights by calculating the z-scores from the weights of a given layer $w^{(\ell)}$ like shown in (11). μ represents a function that calculates the mean, and σ represents a function that calculates the standard deviation of a given variable.

$$z^{(\ell)} \leftarrow \frac{w^{(\ell)} - \mu(w^{(\ell)})}{\sigma(w^{(\ell)})} \tag{11}$$

In the second step, the z-scores $z^{(\ell)}$ are multiplied with the standard deviation given by the initialization strategy σ_{init}. If the mean of the weights $\mu(w^{(\ell)})$ deviates from zero, the z-score changes the signs of values close to the mean value. To counteract this, the initial mean of the weights $\mu(w^{(\ell)})$ has to be added again.

$$w^{(\ell)} \leftarrow z^{(\ell)} \cdot \sigma_{\text{init}} + \mu(w^{(\ell)}) \tag{12}$$

Weight rescaling only relies on the weights of a layer and does not incorporate activation or gradient information. This allows it to be applied at arbitrary moments during training. Experiments show good results when *weight rescaling* is applied once per epoch. This allows the variance of the weights only to grow to a point given by the dataset. We treat *weight rescaling* as a form of re-initialization and reset the biases to 0 each time *weight rescaling* is executed. In the experiments, *weight rescaling* is used on all fully connected and conv2d Layers.

V. Methods

The following segment shows a description of the evaluation procedure that encompasses model selection for the statistical analysis. This is followed by the experimental setup, covering aspects such as training, dataset and model architectures.

A. Evaluation Metrics

Each experiment describes a configuration on which a model is trained, e.g., an architecture with *weight rescaling* is denoted as an experiment and the same architecture with no regularization method as a different experiment. To assess the stochastic effects from random intialization and random data shuffle during training, we execute the training multiple times and call each execution a run. Taking the mean test accuracy of all runs allows to observe the performance of the training process and determine the amount of overfitting in a plot. However, the mean test accuracy throughout training is misleading to determine the experiments performance, since the metric incorporates factors like overfitting and model convergence speed, which are undesireable in the performance metric. For a performance metric we have to take into account how neural networks are used in practice, which is the same reason, why overfitting is less problematic than it first seems. In practice, a model selection step is implemented which takes the epoch at which the model performs best on the validation dataset $\mathcal{D}_{\text{val}}$ and then evaluate on the test dataset $\mathcal{D}_{\text{test}}$. For an accurate metric the training process is not important, but only the best performing model of each run is. We randomly sample a family of runs $\mathcal{R}$ from our random initialization distribution and execute each run $r \in \mathcal{R}$ for E epochs. Let $\mathrm{acc}_r(e; \mathcal{D}_{\text{val}})$ denote the validation accuracy on dataset $\mathcal{D}_{\text{val}}$ for run r at epoch e. Then, we define the best performance of each run r over all epochs as:

$$\mathrm{acc}^*_r(\mathcal{D}_{\text{val}}) := \max_{e \in \{1,\ldots,E\}} (acc_r(e; \mathcal{D}_{\text{val}})) \quad \text{for } r = 1, \ldots, R \tag{13}$$

Instead of selecting the best validation accuracy for each run at every epoch, we want to select the epoch with the best validation accuracy for each run. We define:

$$e^*_r := \underset{e \in \{1,\ldots,E\}}{\arg\max} (acc_r(e; \mathcal{D}_{\text{val}})) \quad \text{for } r = 1, \ldots, R \tag{14}$$

By collecting the epochs at which the run reaches its maximum valudation accuracy we can observe if the experiment has the ability to increase in performance, if the training coninues for longer. For this we set:

$$e^* = [e^*_1, \ldots, e^*_R] \tag{15}$$

Taking the mean of e^* is a metric which indicates that the model shows overfitting at an early point in training, while a

big mean epoch indicates that the model still has the ability to improve the performance when training is continued. For the performance metric, we take the best performing e_r^* from $\mathcal{D}_{\text{val}}$ and compute the corresponding accuracy on test dataset $\mathcal{D}_{\text{test}}$. We set:

$$\text{acc}_r(\mathcal{D}_{\text{test}}) := \text{acc}_r(e_r^*; \mathcal{D}_{\text{test}}) \tag{16}$$

Finally, we set $\text{acc}(\mathcal{D}_{\text{test}})$ as a collection of test accuracies for the best performing runs.

$$\text{acc}(\mathcal{D}_{\text{test}}) := [\text{acc}_1(\mathcal{D}_{\text{test}}), \ldots, \text{acc}_R(\mathcal{D}_{\text{test}})] \tag{17}$$

This allows to reduce the impact of the selection process on the metric. To make statements about the performance of an experiment, we calculate the mean and standard deviation of $\text{acc}(\mathcal{D}_{\text{test}})$. Afterwards we use the distribution of $\text{acc}(\mathcal{D}_{\text{test}})$ for each experiment and test with the Two sided Mann-Whitney-U-Test if the experimental differences are significant.

B. Experimental Setup

In the optimization process, we employ Stochastic Gradient Descent (SGD) with a momentum of 0.9 and a static learning rate of $1e^{-3}$ to minimize the cross-entropy loss. The dataset is CIFAR10, which Alex Krizhevsky [11] proposed. This dataset consists of 60,000 RGB color images with dimensions of 32×32 pixels which belong to 10 distinct classes. The categories encompass real-life objects such as airplanes, cats, or trucks. The CIFAR10 dataset is partitioned into three subsets: a training set $\mathcal{D}_{\text{train}}$ containing 40,000 images, a validation set $\mathcal{D}_{\text{val}}$ with 10,000 images, and a testing set $\mathcal{D}_{\text{test}}$ consisting of another 10,000 images. The MLP contains three fully connected layers with the input layer having 32 neurons, the hidden layer 32 neurons and the output layer having 10 neurons for the 10 classes of CIFAR10. The number of trainable parameters in this architecture is 199,082 and it uses *relu* as activation function. LeNet-5 was introduced in [13] and consists of Convolution, Maxpooling and Dense Layers. The architecture contains 62,006 trainable parameters and uses *tanh* as activation function. AlexNet was introduced in [12] and can be seen as a bigger version of LeNet-5. However, it incorporates *relu* instead of *tanh* as the activation function. The architecture contains 57,044,810 trainable parameters. For AlexNet we use the architecture implemented in Pytorch and adapt the last layer to contain 10 neurons instead of 1000. The architecture contains *dropout* as regularization method for the fully connected classifier. ResNet was introduced by [6] and uses residual layers which forward the activation of a layer by adding the previous activation on top of the current activation. For ResNet we use the architecture implemented in Pytorch, which includes *batch normalization*, and we adapt the last layer to contain 10 neurons instead of 1000. The architecture contains 23,520,842 trainable parameters and uses *relu* as activation function.

VI. Results

The paper presents empirical evidence regarding the application of *weight rescaling* in various deep learning models trained on the CIFAR10 dataset. In the first section, we compare the performance of *weight decay* with *weight rescaling* using a MLP with *relu* activation functions. The second section investigates the impact of *weight rescaling* when the activation function is *tanh* in the LeNet-5 model. In the third section, we apply *weight rescaling* to an AlexNet model with *dropout* regularization. Finally, we examine the effectiveness of *weight rescaling* on a ResNet-50 model with *batch normalization*. All experiments were repeated multiple times and tested against random fluctuations to ensure statistical significance. The sample size for the experiments ranges between 100 to 20, which is why we take the standard threshold of $\alpha_{base} = 5e^{-2}$ as a baseline. Since we apply multiple statistical comparisons on the results we have to counteract the family-wise error rate, which we do by applying the Holm-Bonferroni method [9]. We do this by calculating a specific α for each comparison by dividing the α_{base} by the number of tests $\mathcal{T}$ by that are done on the data.

$$\alpha = \frac{\alpha_{base}}{\mathcal{T}}. \tag{18}$$

In the tables a * indicates that the p-value is below the α for this comparison.

A. MLP: Weight Decay vs. Weight Rescaling

Weight decay and *weight rescaling* show similarities as they both do not change the models' architecture but are employed in the training process. This section serves to show the similarities and differences among *weight decay* and *weight rescaling* on the MLP model, where each experiment contains 100 runs. In order to establish an appropriate *weight decay* setting, three distinct λ values ($1e^{-2}$, $1e^{-3}$, $1e^{-4}$) are chosen based on best practices and empirical testing. The optimal value for λ, is likely situated between these options and depends on both, the architecture of the model and the characteristics of the dataset. Figure 3 shows the MLP training process for each experiment. The experiments for base/none without regularitation and *weight decay* with a small $\lambda = 1e^{-4}$ show clear overfitting by spiking around epoch 10 and dropping of afterwards. With a big $\lambda = 1e^{-2}$ the test accuracy does not show overfitting. However it does not reach its full potential, recalling the observations of the low generatlization error in Figure 2. *Weight rescaling* and *weight decay* with $\lambda = 1e^{-3}$ show a similar training curve of the test accuracy without overfitting. However there is one striking difference between the training curve of *weight rescaling* and *weight decay* with $\lambda = 1e^{-3}$, which is that *weight rescaling* converges slower than *weight decay* with $\lambda = 1e^{-3}$.

The results show that *weight rescaling* does in fact avoid overfitting, just like *weight decay* does, if a good λ value is selected. However, another important feature of regularization is to increase the performance. By selecting the best performing models of a run according to (17) and calculating

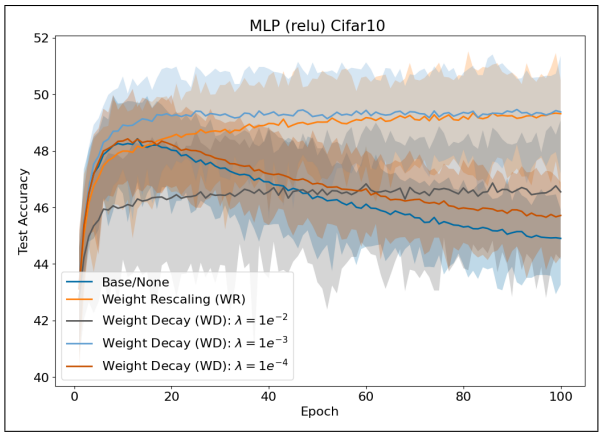

Fig. 3: MLP trained on CIFAR10 for the Base/None experiment without regularization, *weight rescaling* and *weight decay* with various λ values. The line in the middle shows the mean test accuracy and the area shows the min and max results for 100 repeated runs at the specific epoch.

their distributional statistics an increased performance is revealed. The mean and standard deviation are shown in Table I. The results display increased performance for *weight rescaling* and *weight decay* with $\lambda = 1e^{-3}$.

TABLE I: The mean and standard deviation for the test accuracies calculated by 17 and the overfitting indicator calculated by 15 for the MLP experiments.

Regularization	acc($\mathcal{D}_{\text{test}}$)	e^*
Base/None	$48.72\% \pm 0.48$	14.02 ± 04.40
WR	$50.08\% \pm 0.50$	76.01 ± 20.52
WD: $\lambda = 1e^{-2}$	$48.00\% \pm 0.45$	58.93 ± 24.27
WD: $\lambda = 1e^{-3}$	$50.17\% \pm 0.48$	61.41 ± 23.76
WD: $\lambda = 1e^{-4}$	$48.90\% \pm 0.53$	15.86 ± 05.17

While the performance is increased we have to check if the effect is significant and is not explained by random sampling. Table II shows the Mann-Whitney-U-Test of the acc($\mathcal{D}_{\text{test}}$) distributions. Due to combinatory complexity the table does not list the test results for *weight decay* with a $\lambda = 1e^{-2}$ and $\lambda = 1e^{-4}$, since they show results close to the Base experiment. The results show small p-values for *weight rescaling* and *weight decay* with $\lambda = 1e^{-3}$ when compared to the Base/None experiment, which strongly indicates that their results stem from different distributions. This concludes that the performance increase is not due to random fluctuations but due to a significant impact of *weight rescaling* and *weight decay* with a $\lambda = 1e^{-3}$. Even though *weight rescaling* shows a slightly lower mean accuracy than *weight decay* with $\lambda = 1e^{-3}$, the p-value between them is large, indicating that the performance increase between them is not significant.

B. LeNet-5: Tanh as Activation Function

Figure 4 shows the test accuracy for LeNet-5, where each experiment contains 50 runs. LeNet-5 does not utilize any measures for regularization which is why clear overfitting is

TABLE II: Results of the two sided Mann Whitney U Tests for independent samples on the MLP experiments. The * indicates significant differences in the rank sum of the two groups with a p-value below $\alpha = 1.\overline{6}e^{-2}$.

Group 1	Group 2	Mann-Whitney-U-Test statistics	p-value
Base/None	WR	$1.20e^{1}$	$1.01e^{-31}*$
Base/None	WD: $\lambda = 1e^{-3}$	$3.00e^{0}$	$1.87e^{-32}*$
WR	WD: $\lambda = 1e^{-3}$	$2.08e^{3}$	$1.31e^{-1}$

visible in the lines for Base/None. Since LeNet-5 utilizes *tanh* for this comparison we experiment with the initialization methods for Xavier and Kaiming in combination with *weight rescaling*. Like stated in III-G Pytorch allows to use of the gain scaling factor with Xavier initialization. Experiments with different different gain values show similar results, which is why we only report the results for $gain = 1$. For the Kaiming experiments we act like *relu* is used and take $gain = \sqrt{2}$ to calculate $\sigma^{(l)}_{\text{Kaiming}}$. Even Though *weight rescaling* reduces overfitting when applied with Xavier or Kaiming, it does not entirely avoid it like in the MLP results 3 with *relu*.

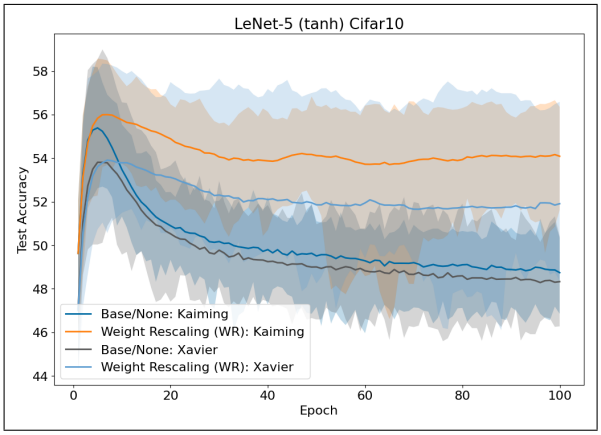

Fig. 4: Test accuracy graph of LeNet-5 where each experiment contains 50 runs. Showing strong overfitting for the base experiments without *weight rescaling* and less overfitting in experiments With *weight rescaling*. Even though the network uses *tanh* as an activation function *weight rescaling* with Kaiming shows better results.

Figure 4 demonstrates the Test Accuracy during training. Interestingly, LeNet-5 achieves better results when using Kaiming initialization with or without *weight rescaling* compared to Xavier initialization, despite *tanh* being its activation function. This is unexpected because the experiments from Xavier et al. [4] were done with *tanh* and we expected that Xavier initialization would create better results with activation functions that are zero-centered like the *linear* activation function. Kaiming initialization was developed with *relu* in mind, which is not zero-centered.

Furthermore, it appears that our regularization method of *weight rescaling* is less effective in enhancing models with *tanh* as their activation function compared to those using

relu.

TABLE III: LeNet-5 CIFAR10 result table which shows that Kaiming initialization and *weight rescaling* works better than Xavier even though *tanh* is used in the LeNet-5 architecture. However, the low mean e^* shows that all experiment are overfitting.

Regularization	acc($\mathcal{D}_{\text{test}}$)	e^*
Base: Xavier	$53.99\% \pm 1.22$	06.04 ± 01.51
WR: Xavier	$54.21\% \pm 1.19$	15.76 ± 20.31
Base: Kaiming	$55.49\% \pm 0.99$	04.78 ± 00.97
WR: Kaiming	$56.22\% \pm 1.06$	17.18 ± 22.03

The acc($\mathcal{D}_{\text{test}}$) and e^* results in Table III confirm these observations. The mean e^* for *weight rescaling* is higher than for the Base/None experiment. The mean e^* is low when compared to the results of the MLP in Table I or the results for AlexNet and ResNet-50 in Table V. The mean acc($\mathcal{D}_{\text{test}}$) with *weight rescaling* is slightly increased compared to the base model in the case of Xavier with $54.21\% - 53.99\% = 0.22\%$. However the performance increase is much larger in the case of Kaiming with $56.22\% - 55.39\% = 0.73\%$. Another interesting fact is that initialization using Kaiming performed better in general with a difference of $55.49\% - 53.99\% = 1.5\%$.

TABLE IV: Comparison between LeNet-5 with and without *weight rescaling* when applied with Xavier or Kaiming initialization. Significance is shown by the two sided Mann Whitney U test for independent samples. The * indicates significant differences in the rank sum of the two groups with a p-value below $alpha = 8.\overline{3}e^{-3}$.

Group 1	Group 2	Mann-Whitney-U-Test	
		statistics	p-value
Base: Kaiming	WR: Kaiming	$1.16e^2$	$5.38e^{-4}*$
Base: Kaiming	Base: Xavier	$9.25e^1$	$4.80e^{-10}*$
Base: Kaiming	WR: Xavier	$1.60e^2$	$5.04e^{-8}*$
WR: Kaiming	Base: Xavier	$6.80e^1$	$2.94e^{-13}*$
WR: Kaiming	WR: Xavier	$7.20e^1$	$3.95e^{-12}*$
Base: Xavier	WR: Xavier	$5.20e^2$	$2.66e^{-1}$

For the statistical analysis in Table IV, most groups show significant differences to the other group with a small p-value. This tells us that the performance improvements from *weight rescaling* with Kaiming are significant. However, the p-value for "Base: Xavier" and "WR: Xavier" is large, which indicates that *weight rescaling* has no impact on the performance when used with Xavier initialization on a *tanh* network. The results show that *weight rescaling* improves the performance on *tanh* networks, but it does not avoid overfitting. This indicate that *weight rescaling* does not create optimal results when combined with an activation function with a "s-shaped curve" like *tanh* compared to a noncurved activation function like *relu*. For all other experiments in this paper we use *weight rescaling* with Kaiming.

C. AlexNet: Dropout and Weight Rescaling

Figure 5 shows the test accuracy during training for the AlexNet architecture. The Base experiments with *dropout* is shown in blue and the experiment with *dropout* and *weight rescaling* is shown in orange. Upon examining the blue line, it is evident that the performance of the experiment without *weight rescaling* differs from those observed in MLP VI-A and LeNet-5 VI-B. The other experiments display a sharp increase followed by a decline in test accuracy. However, AlexNet demonstrates distinct behavior. Its test accuracy rises and stays steady until reaching epoch 20, at which point the runs begin to deteriorate rapidly. Unlike the overfitted models that remain well beyond chance level, AlexNet experiences a total collapse and does not fully recover. The *dropout* rate is 50% and AlexNet shows a regularizing effect in the beginning, but it might be too high for longer training. Adding *weight rescaling* to the training process fixes the model collapse. The results of AlexNet in combination with LeNet-5 show that *weight rescaling* does not only create significantly better results on Fully Connected Layer, but in architectures with Conv2d Layers as well.

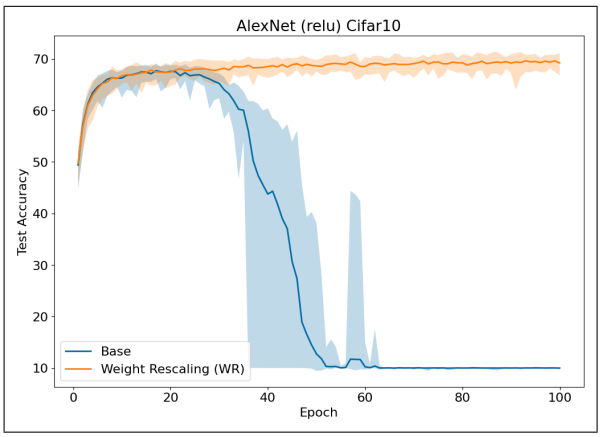

Fig. 5: Test accuracy graphs for the AlexNet experiments with *dropout* in blue and *dropout* plus *weight rescaling* in orange, where each experiments contains 20 runs.

D. ResNet-50: Batch Normalization and Weight Rescaling

Figure 6 shows the accuracy Test Accuracy during training of 40 repeated runs. The blue line shows the Base experiment with *batch normalization* and the orange line shows the experiment with *batch normalization* and *weight rescaling*. Compared to AlexNet, ResNet-50 does not collapse and it shows no signs of overfitting. Generally the two curves look simlar to each other, indicating successful regularization in both experiments.

The result in Table V show the performance of AlexNet and ResNet with and without *weight rescaling*. We can see that in both cases *weight rescaling* increases the performace of the architecture. For AlexNet the performance increase is $70.63\% - 68.33\% = 2.33\%$ and for ResNet-50 the increase is $70.31\% - 69.46\% = 0.85\%$. While the performance improvement for AlexNet is much larger, Table VI shows that both improvements are significant.

In addition to the significantly improved performance, *weight rescaling* improved the ability to reach better per-

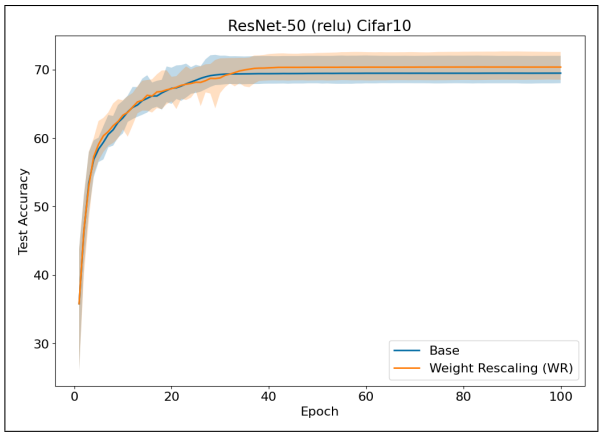

Fig. 6: Test accuracy for ResNet-50 with *batch normalization* in blue and with *batch normalization* plus *weight rescaling* in orange, where each experiment contains 40 runs.

TABLE V: Performance Table of AlexNet and ResNet with and without *weight rescaling*. For both architectures *weight rescaling* did improve the performance

Architecture	Regularization	acc($\mathcal{D}_{\text{test}}$)	e^*
AlexNet	Base	$68.33\% \pm 0.42$	18.20 ± 03.59
AlexNet	WR	$70.63\% \pm 0.54$	80.10 ± 16.30
ResNet-50	Base	$69.46\% \pm 0.80$	76.50 ± 22.30
ResNet-50	WR	$70.31\% \pm 0.86$	61.17 ± 19.18

formance on later epochs on AlexNet, by stabilizing training as well.

TABLE VI: The tests show that the performance improvements by *weight rescaling* with Kaiming for AlexNet and ResNet-50 are significant. Significance is shown by the two sided Mann Whitney U test for independent samples. The * indicates significant differences in the rank sum of the two groups. Since the test uses different data for each test the $alpha = 5e^{-2}$ is equal to the α_{base}.

Group 1	Group 2	Mann-Whitney-U-Test	
		statistics	p-value
AlexNet: Base	AlexNet: WR	$0e^{0}$	$6.80e^{-8}*$
ResNet: Base	ResNet: WR	$1.10e^{1}$	$5.53e^{-4}*$

With the experiments we are able to show that each architecture shows a significant boost in performance by adding *weight rescaling*. The results are comparable to *weight decay*, without the need to define additional hyperparameter. *Weight rescaling* works best on architectures utilizing *relu* as activation function with Kaiming initialization and Kaiming based *weight rescaling*. While a *tanh* architecture can be significantly improved by Kaiming based *weight rescaling* it still showed overfitting, which was not present in any of the *relu* architectures. For the case of AlexNet with *dropout*, *weight rescaling* showed stabilizing properties in the training process in addition to significantly improving the performance. Finally *weight rescaling* was able to significantly improve the results of ResNet-50, which already utilizes *batch normalization*.

VII. Discussion

In this section we go back to the theory presented in III-H and explain how the variance propagation through training changes when *weight rescaling* is applied. The upper plot of Figure 7 shows the variance of the pre-activations in each layer without regularization. The Figures show the pre-activations $p^{(l)}$ instead of the activations after the activation function for consistency sake, since the last layer of the MLP does not include a *relu* activation function. All 100 runs were initialized by a normal distribution with a standard deviantion given by $\sigma^{(l)}_{\text{Kaiming}}$. The pre-activation increases quickly after initialization and does not decrease during training The pre-activation variance for the input and output layer increase much slower compared to the hidden layer. We assume that the pre-activation variance for the input and output layer are bound by the data in the supervised classification task on CIFAR10. The drastic increase in variance in the hidden layer indicates that the model becomes less stable. Small changes in the incoming activations to the hidden layer can result in big changes to the output of the hidden layer, which makes the model more sensitive to noise.

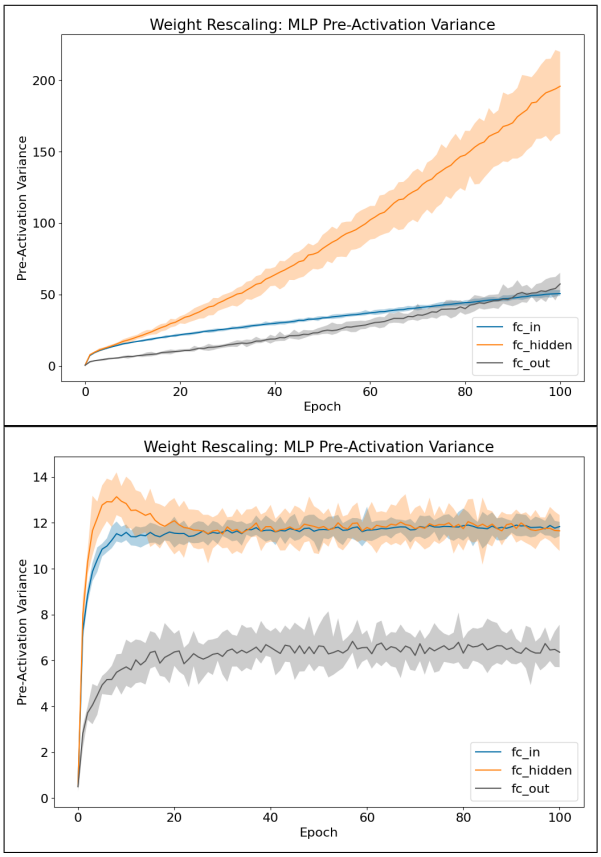

Fig. 7: The x-axis shows the epoch and the y-axis shows the pre-activation variance. The plots show the three layers of 100 MLP networks, where each color represents a different layer. The upper plot does not use regularization, while the lower plot uses *weight rescaling*.

The lower plot of Figure 7 shows the variance of the

pre-activations in each layer with *weight rescaling* as regularization method. At the beginning of training, models are initialized with low variance, but after the first epoch, the variance increases rapidly. The difference between having no regularization and using *weight rescaling* is evident. In contrast to unregulated models, where variance grows indefinitely, *weight rescaling* limits the variance for each layer. This prevents self-reinforcing effects on variance growth, making the model more stable and less sensitive to noise. Additionally, the goal definition from (10) holds true for later epochs but not for early ones where the variance deviates before it appears to converge towards a layer-specific point.

VIII. Conclusion and Outlook

In general, using initialization strategies during training has proven to be an effective technique for stabilizing the training process and improving performance in neural networks. One such method is *weight rescaling*, which serves as a regularization technique without requiring additional hyperparameters while being efficient in terms of memory requirements and computational resources. With the advancement of new initialization strategies that involve sampling weights from random distributions, *weight rescaling* can be further improved by incorporating their standard deviation σ_{init} into the scaling formula from (12).

In the future, we aim to demonstrate the effectiveness of *weight rescaling* beyond just supervised classification tasks. This includes scenarios where *batch normalization* cannot be applied, such as in recurrent models or highly sensitive applications like reinforcement learning and generative models. To further explore this topic, additional experiments could involve modifying the variance convergence point by adjusting the learning rate, applying *weight rescaling* multiple times within an epoch, or changing the layer width. In this context it might be beneficial to incorporate weight rescaling into the loss function by approximating optimal variance for each layer, which develops in the lower plot of Figure 7. Another interesting area for exploration could involve rescaling biases in a similar manner to weights instead of resetting them to 0. Furthermore, comparing the performance of *weight rescaling* with other regularization techniques like *layer normalization* could be of interest.

Acknowledgment

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 456666331

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang. Tensorflow: A system for large-scale machine learning. *CoRR*, abs/1605.08695, 2016.

[2] Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU), February 2019. arXiv:1803.08375 [cs, stat].

[3] Johan Bjorck, Carla Gomes, Bart Selman, and Kilian Q. Weinberger. Understanding Batch Normalization, November 2018. arXiv:1806.02375 [cs, stat].

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, March 2010.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, November 2016.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015. arXiv:1512.03385 [cs].

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[8] Jeff Heaton. Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning. *Genetic Programming and Evolvable Machines*, 19(1):305–307, June 2018.

[9] Sture Holm. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979.

[10] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, March 2015. arXiv:1502.03167 [cs].

[11] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. *University of Toronto*, 2009.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[13] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[14] Ziquan Liu, Yufei Cui, Jia Wan, Yu Mao, and Antoni B. Chan. Weight Rescaling: Effective and Robust Regularization for Deep Neural Networks with Batch Normalization, June 2022. arXiv:2102.03497 [cs, stat].

[15] N. Morgan and H. Bourlard. Generalization and Parameter Estimation in Feedforward Nets: Some Experiments. In *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[17] Daniel A. Roberts, Sho Yaida, and Boris Hanin. The Principles of Deep Learning Theory, May 2022. arXiv:2106.10165 [hep-th, stat].

[18] Tim Salimans and Durk P Kingma. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[19] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How Does Batch Normalization Help Optimization?, April 2019. arXiv:1805.11604 [cs, stat].

[20] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.