NORMALIZATION IS DEAD, LONG LIVE NORMALIZATION!

Pieter-Jan Hoedt[*], **Sepp Hochreiter**[†] **& Günter Klambauer**
ELLIS Unit Linz, Institute for Machine Learning
Johannes Kepler University, Linz, Austria

ABSTRACT

Training residual networks without batch normalization is suprisingly challenging. Recently, normalizer-free variants were trained successfully, indicating that it might be possible after all. We look at how *normalizer-free* these networks really are and how their results could teach us something about normalization and signal propagation. These ideas were originally presented as a contribution to the first blog post track at ICLR 2022[1]. The main goal of this PDF version is to provide this content in a more traditional format[2]. Please, cite the original blog post contribution when referencing this work.

1 INTRODUCTION

Since the advent of Batch Normalization (BN), almost every State-Of-The-Art (SOTA) method uses some form of normalization. After all, normalization generally speeds up learning and leads to models that generalize better than their unnormalized counterparts. This turns out to be especially useful when using some form of skip connections, which are prominent in Residual Networks (ResNets), for example. However, Brock et al. (2021a) suggest that SOTA performance can also be achieved using *ResNets without normalization*!

The fact that Brock et al. went out of their way to get rid of something as simple as BN in ResNets, for which BN happens to be especially helpful, does raise a few questions:

1. Why get rid of BN in the first place?
2. How (easy is it) to get rid of BN in ResNets?
3. Is BN going to become obsolete in the near future?
4. Does this allow us to gain insights into why BN works so well?
5. Wait a second... Are they getting rid of normalization or just BN?

The goal of this blog post is to provide some insights w.r.t. these questions using the results from Brock et al. (2021a).

2 NORMALIZATION

To set the scene for a world without normalization, we start with an overview of normalization layers in neural networks. Batch Normalization (BN) is probably the most well-known method, but there are plenty of alternatives. Despite the variety of normalization methods, they all build on the same principle ideas.

[*]`hoedt@ml.jku.at`
[†]Insitute of Advanced Reearch in Artificial Intelligence (IARAI)
[1]`https://iclr-blog-track.github.io/2022/03/25/unnormalized-resnets/`
[2]so that web crawlers would be able to find and index it

2.1 Origins

The design of modern normalization layers in neural networks is mainly inspired by data normalization (LeCun et al., 1998; Schraudolph, 1998; Ioffe & Szegedy, 2015). In the setting of a simple linear regression, it can be shown (see e.g. LeCun et al., 1998) that the second-order derivative, i.e., the Hessian, of the objective is exactly the covariance of the input data, $\mathcal{D}$:

$$\frac{1}{|\mathcal{D}|}\sum_{(\boldsymbol{x},y)\in\mathcal{D}} \nabla^2_{\boldsymbol{w}} \frac{1}{2}(\boldsymbol{w}^\mathsf{T}\boldsymbol{x} - y)^2 = \frac{1}{|\mathcal{D}|}\sum_{(\boldsymbol{x},y)\in\mathcal{D}} \boldsymbol{x}\boldsymbol{x}^\mathsf{T}. \tag{1}$$

If the Hessian of an optimization problem is (close to) the identity, it becomes much easier to find a solution (LeCun et al., 1998). Therefore, learning should become easier if the input data is whitened — i.e., is transformed to have an identity covariance matrix. However, full whitening of the data is often costly and might even degenerate generalization performance (Wadia et al., 2021). Instead, the data is *normalized* to have zero mean and unit variance to get at least some of the benefits of an identity Hessian.

When considering multi-layer networks, the expectation would be that things get more complicated. However, it turns out that the benefits of normalizing the input data for linear regression directly carry over to the individual layers of a multi-layer network (LeCun et al., 1998). Therefore, simply normalizing the inputs to a layer — i.e., the outputs from the previous layer — should also help to speed up the optimization of the weights in that layer. Using these insights, (Schraudolph, 1998) showed empirically that centering the activations effectively speeds up learning.

Also initialization strategies commonly build on these principles (e.g. LeCun et al., 1998; Glorot & Bengio, 2010; He et al., 2015). Since the initial parameters of a layer are independent of the inputs, they can easily be tuned. When tuned correctly, it can be assured that the (pre)-activations of each layer are normalized throughout the network before the first update. However, as soon as the network is being updated, the distributions change and the normalizing properties of the initialization get lost (Ioffe & Szegedy, 2015).

2.2 Batch Normalization

In contrast to classical initialization methods, BN is able to maintain fixed mean and variance of the activations as the network is being updated (Ioffe & Szegedy, 2015). Concretely, this is achieved by applying a typical data normalization to every mini-batch of data, $\mathcal{B}$:

$$\hat{\boldsymbol{x}} = \frac{\boldsymbol{x} - \boldsymbol{\mu}_\mathcal{B}}{\boldsymbol{\sigma}_\mathcal{B}}. \tag{2}$$

Here $\boldsymbol{\mu}\_\mathcal{B} = \frac{1}{|\mathcal{B}|}\sum\_\boldsymbol{x} \in \mathcal{B}\boldsymbol{x}$ is the mean over the inputs in the mini-batch and $\boldsymbol{\sigma}_\mathcal{B}$ is the corresponding standard deviation. Also, note that the division is element-wise and generally is numerically stabilized by some ε when implemented. In case a zero mean and unit variance is not desired, it is also possible to apply an affine transformation $\boldsymbol{y} = \boldsymbol{\gamma} \odot \hat{\boldsymbol{x}} + \boldsymbol{\beta}$ with learnable scale ($\boldsymbol{\gamma}$) and mean ($\boldsymbol{\beta}$) parameters (Ioffe & Szegedy, 2015). Putting these formulas together in PyTorch (Paszke et al., 2019) code, BN can be summarized as follows:

```python
def batch_normalize(x, gamma=1., beta=0., eps=1e-5):
    mu = torch.mean(x, dim=(0, -1, -2))
    var = torch.var(x, dim=(0, -1, -2))
    x_hat = (x - mu) / torch.sqrt(var + eps)
    return gamma * x_hat + beta
```

The above description explains the core operation of BN during training. However, during inference, it is not uncommon to desire predictions for single samples. Obviously, this would cause trouble because a mini-batch with a single sample has zero variance. Therefore, it is common to accumulate the statistics that are used for normalization ($\boldsymbol{\mu}\_\mathcal{B}$ and $\boldsymbol{\sigma}\_\mathcal{B}^2$) over multiple mini-batches during training. These accumulated statistics can then be used as estimators for the mean and variance during inference. This makes it possible for BN to be used on single samples during inference.

The original reason for introducing BN was to alleviate the so-called *internal covariate shift*, i.e. the change of distributions as the network updates. More recent research has pointed out, however, that internal covariate shift does not necessarily deteriorate learning dynamics (Santurkar et al., 2018). Apparently, (Ioffe & Szegedy, 2015) also realized that simply normalizing the signal does not suffice to achieve good performance:

> [...] the model blows up when the normalization parameters are computed outside the gradient descent step.

All of this seems to indicate that part of the success of BN is due to the effects it has on the gradient signal. The affine transformation in BN simply scales the gradient, such that $\nabla_{\hat{\boldsymbol{x}}}\mathcal{L} = \boldsymbol{\gamma} \odot \nabla_{\boldsymbol{y}}\mathcal{L}$. The normalization operation, on the other hand, transforms the gradient, $\boldsymbol{g} = \nabla_{\hat{\boldsymbol{x}}}\mathcal{L}$, as follows:

$$\nabla_{\boldsymbol{x}}\mathcal{L} = \frac{1}{\boldsymbol{\sigma}_{\mathcal{B}}}\big(\boldsymbol{g} - \mu_g\,\mathbf{1} - \operatorname{cov}(\boldsymbol{g}, \hat{\boldsymbol{x}}) \odot \hat{\boldsymbol{x}}\big), \tag{3}$$

where $\mu_g = \sum_{\boldsymbol{x}\in\mathcal{B}} \nabla_{\hat{\boldsymbol{x}}}\mathcal{L}$ and $\operatorname{cov}(\boldsymbol{g}, \hat{\boldsymbol{x}}) = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{x}\in\mathcal{B}} \boldsymbol{g} \odot \hat{\boldsymbol{x}}$. Note that this directly corresponds to centering the gradients, which is also supposed to improve learning speed (Schraudolph, 1998).

In the end, everyone seems to agree that one of the main benefits of BN is that it enables higher learning rates (Ioffe & Szegedy, 2015; Bjorck et al., 2018; Santurkar et al., 2018; Luo et al., 2019), which results in faster learning and better generalization. An additional benefit is that BN is scale-invariant and therefore much less sensitive to weight initialization (Ioffe & Szegedy, 2015; Ioffe, 2017).

2.3 Alternatives

Why would we ever want to get rid of BN then? Although BN provides important benefits, it also comes with a few downsides:

- BN does not work well with *small batch sizes* (Ba et al., 2016; Salimans & Kingma, 2016; Ioffe, 2017). For a batch size of one, we have zero standard deviation, but also with a few samples, the estimated statistics are often not accurate enough.
- BN is not directly applicable to certain input types (Ba et al., 2016, also see Figure 1) and performs poorly when there are *dependencies between samples* in a mini-batch (Ioffe, 2017).
- BN uses *different statistics for inference* than those used during training (Ba et al., 2016; Ioffe, 2017). This is especially problematic if the distribution during inference is different or drifts away from the training distribution.
- BN does not play well with *other regularization* methods (Hoffer et al., 2018). This is especially known for L_2-regularization (Hoffer et al., 2018) and dropout (Li et al., 2019).
- BN introduces a significant *computational overhead* during training (Ba et al., 2016; Salimans & Kingma, 2016; Gitman & Ginsburg, 2017). Because of the running averages, also memory requirements increase when introducing BN.

Therefore, alternative normalization methods have been proposed to solve one or more of the problems listed above while trying to maintain the benefits of BN.

One family of alternatives simply computes the statistics along different dimensions (see Figure 2). **Layer Normalization (LN)** is probably the most prominent example in this category (Ba et al., 2016). Instead of computing the statistics over samples in a mini-batch, LN uses the statistics of the feature vector itself. This makes LN invariant to weight shifts and scaling individual samples. BN, on the other hand, is invariant to data shifts and scaling individual neurons. LN generally outperforms BN in fully connected and recurrent networks but does not work well for convolutional architectures according to (Ba et al., 2016). **Group Normalization (GN)** is a slightly modified version of LN that also works well for convolutional networks (Wu & He, 2018). The idea of GN is to compute statistics over groups of features in the feature vector instead of all features. For convolutional networks that should be invariant to changes in contrast, statistics can also be computed

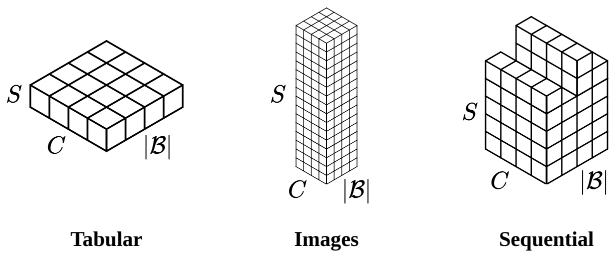

Figure 1: Different input types in terms of their typical batch size ($|\mathcal{B}|$), the number of channels/features (C) and the *size* of the signal (S) (e.g. width times height for images). Image inspired by (Wu & He, 2018).

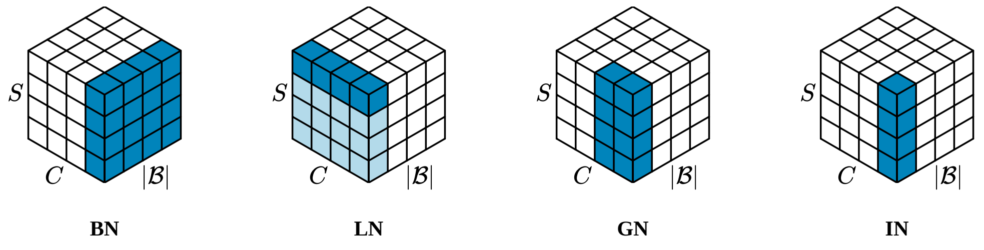

Figure 2: Normalization methods (Batch, Layer, Instance and Group Normalization) and the parts of the input they compute their statistics over. $|\mathcal{B}|$, C, and S are batch size, number of channels/features and signal size, respectively (cf. Figure 1). The lightly shaded region for LN indicates the additional context that is typically used for image data. Image has been adapted from (Wu & He, 2018).

over single image channels for each sample. This gives rise to a technique known as **Instance Normalization (IN)**, which proved especially helpful in the context of style transfer (Ulyanov et al., 2017).

Instead of normalizing the inputs, it is also possible to get a normalizing effect by rescaling the weights of the network (Arpit et al., 2016). Especially in convolutional networks, this can significantly reduce the computational overhead. With **Weight Normalization (WN)** (Salimans & Kingma, 2016), the weight vectors for each neuron are normalized to have unit norm. This idea can also be found in a(n independently developed) technique called **Normalization Propagation (NP)**(Arpit et al., 2016). However, in contrast to WN, NP accounts for the effect of (ReLU) activation functions. In some sense, NP can be interpreted as a variant of BN where the statistics are computed theoretically (in expectation) rather than on the fly. **Spectral Normalization (SN)**, on the other hand, makes use of an induced matrix norm to normalize the entire weight matrix (Miyato et al., 2018). Concretely, the weights are scaled by the reciprocal of an approximation of the largest singular value of the weight matrix.

Whereas WN, NP and SN still involve the computation of some weight norm, it is also possible to obtain normalization without any computational overhead. By creating a forward pass that induces attracting fixed points in mean and variance, **Self-Normalizing Networks (SNN)** Klambauer et al. (2017) are able to effectively normalize the signal. To achieve these fixed points, it suffices to carefully scale the ELU activation function (Clevert et al., 2016) and the initial variance of the weights. Additionally, Klambauer et al. (2017) provide a way to tweak dropout so that it does not interfere with the normalization. Maybe it is useful to point out that SNNs do not consist of explicit normalization operations. In this sense, an SNN could already be seen as an example of *normalizer-free* networks.

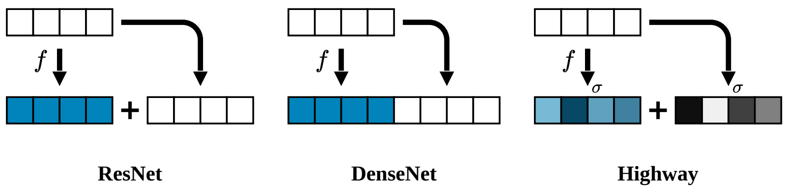

Figure 3: Variations on skip connections in ResNets, Densenets and Highway networks. The white blocks correspond to the input / skip connection and the blue blocks correspond to the output of the non-linear transformation. The greyscale blocks are values between zero and one and correspond to masks.

3 Skip Connections

With normalization out of the way, we probably want to tackle the *skip connections*. After all, (Brock et al., 2021a) mainly aim to rid Residual Networks (ResNets) of normalization. Although skip connections already existed long before ResNets were invented, they are often considered as one of the main contributions by the work of (He et al., 2016b). In some sense, it almost seems as if skip connections could only become popular after BN was invented. Especially if we consider the effects of skip connections on the statistics of signals flowing through the network.

3.1 History

Shortcut or *skip connections* make it possible for information to bypass one or more layers in a neural network. Mathematically, they are typically expressed using a formalism of the form

$$\boldsymbol{y} = \boldsymbol{x} + f(\boldsymbol{x}), \tag{4}$$

where f represents some non-linear transformation (He et al., 2016b;a). This non-linear transformation is typically a sub-network that is commonly referred to as the *residual branch* or *residual connection*. When the outputs of the residual branch have different dimensions, it is typical to use a linear transformation to match the output dimension of the skip connection with that of the residual connection.

Since it often helps to have a few lines of code to understand these vague descriptions, an implementation of the skip connections from (He et al., 2016a) is given below. The comments aim to highlight the differences with the ResNets from (He et al., 2016b). For a complete implementation of this skip connection module, we refer to the code in appendix A.1 at the end of this post.

```
def forward(self, x):
    x = self.preact(x)  # diff 1: compute global pre-activations
    skip = self.downsample(x)
    residual = self.residual_branch(x)
    # return torch.relu(residual + skip) (diff 2)
    return residual + skip
```

Skip connections became very popular in computer vision due to the work of He et al. (2016b). However, they were already commonly used as a trick to improve learning in multi-layer networks before deep learning was even a thing (Ripley, 1996). Similar to normalization methods, skip connections can improve the condition of the optimization problem by making it harder for the Hessian to become singular (van der Smagt & Hirzinger, 1998). However, skip connections also have benefits in the forward pass: e.g., (Srivastava et al., 2015) argue that information should be able to flow through the network without being altered. He et al. (2016b), on the other hand, claim that learning should be easier if the network can focus on the non-linear part of the transformation (and ignore the linear component).

The general formulation of skip connections that we provided earlier, captures the idea of skip connections very well. As you might have expected, however, there are plenty of variations on the exact formulation (a few of which are illustrated in Figure 3). Strictly speaking, even He et al. (2016b) do not adhere to their own formulation because they apply an activation function on what we denoted as $\boldsymbol{y}$ (He et al., 2016a, see code snippet). In DenseNets (Huang et al., 2017a), the outputs of the skip and residual connections are concatenated instead of aggregated by means of a sum. This retains more of the information for subsequent layers. Other variants of skip connections make use of masks to select which information is passed on. Highway networks (Srivastava et al., 2015) make use of a gating mechanism similar to that in Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997). These gates enable the network to learn how information from the skip connection is to be combined with that of the residual branch. Similarly, Transformers (Vaswani et al., 2017) could be interpreted as a variation on highway networks without residual branches. This comparison does only hold, however, if you are willing to interpret the attention mask as some form of complex gate for the skip connection.

3.2 Moment Control

Traditional initialization techniques manage to provide a stable starting point for the propagation of mean and variance in fully connected layers, but they do not work so well in ResNets. The key problem is that the variance can not remain constant when simple additive skip connections are used. After all, the variance is linear and unless the non-linear transformation branch would output a zero-variance signal, the output variance must be greater than the input variance. Moreover, if the signal would have a strictly positive mean, also the mean would start drifting when residual layers are chained together. Luckily, these drifting effects can be mitigated to some extent, e.g. by using BN. However, are there alternative approaches and if yes, what are these approaches?

Before we come to possible solutions, it might be useful to point out that these drift effects are due to the simple *additive* skip connections used in ResNets. For example, the gating mechanism that is used to control the skip connection in highway networks makes the mean shift much less of a problem than in ResNets. In the case of DenseNets, the concatenation does not affect either mean or variance if the residual branch produces outputs with similar statistics as the inputs. Therefore, we mainly focus on these simple *additive* skip connections in ResNets.

Similar to standard initialization methods, the key idea to counter drifting in ResNets is to stabilize the variance propagation. To this end, a slightly modified formulation of skip connections is typically used (e.g. Szegedy et al., 2016; Balduzzi et al., 2017; Hanin & Rolnick, 2018):

$$\boldsymbol{y} = \alpha \boldsymbol{x} + \beta f(\alpha \boldsymbol{x}), \tag{5}$$

which is equivalent to the original formulation when $\alpha = \beta = 1$. The key advantage of this formulation is that the variance can be controlled (to some extent) by tuning the newly introduced scaling factors α and β. In terms of code, these modifications could look something like

```
def forward(self, x):
    x = self.preact(self.alpha * x)
    skip = self.downsample(x)
    residual = self.residual_branch(x)
    return self.beta * residual + skip
```

A very simple counter-measure to the variance explosion in ResNets is to set $\alpha = 1/\sqrt{2}$ (Balduzzi et al., 2017). Assuming that the residual branch approximately preserves the variance, the variances of $\boldsymbol{y}$ and $\boldsymbol{x}$ should be roughly the same. In practice, however, it seems to be more common to tune the β factor instead of α (Balduzzi et al., 2017). For instance, simply setting β to some small value (e.g., in the range $[0.1, 0.3]$) can already help ResNets (with BN) to stabilize training (Szegedy et al., 2016). It turns out that having small values for β can help to preserve correlations between gradients, which should benefit learning (Balduzzi et al., 2017).

Similar findings were established through the analysis of the variance propagation in ResNets by (Hanin & Rolnick, 2018). Eventually, they propose to set $\beta = b^l$ after the l-th skip connection, with $0 < b < 1$ to make sure that the sum of scaling factors from all layers converges. Arpit et al. (2019) additionally take the backward pass into account and show that $\beta = L^{-1}$ provides stable variance propagation in a ResNet with L skip connections. Learning the scaling factor β in each layer can also make it possible to keep the variance under control (Zhang et al., 2019; De & Smith, 2020).

4 Normalizer-Free ResNets

It could be argued that the current popularity of skip connections is due to BN. After all, without BN, the skip connections in ResNets would have suffered from the drifting effects discussed earlier (sec. 3.2). However, this does not take away that BN does have a few practical issues (sec. 2.3) and there are alternative techniques to control these drifting effects. Therefore, it makes sense to research the question of whether BN is just a useful or a *necessary* component of the ResNet architecture.

4.1 Old Ideas

Whereas some alternative normalization methods aim to simply provide normalization in scenarios where BN does not work so well, other methods have been explicitly designed to reduce or get rid of the normalization computations (e.g. Arpit et al., 2016; Salimans & Kingma, 2016; Klambauer et al., 2017). Even the idea of training ResNets without BN is practically as old as ResNets themselves. With their Layer-Sequential Unit-Variance (LSUV) initialization, Mishkin & Matas (2016) showed that it is possible to replace BN with good initialization for small datasets (CIFAR-10). Similarly, Arpit et al. (2019) are able to close the gap between WN and BN by reconsidering weight initialization in ResNets.

Getting rid of BN in ResNets was posed as an explicit goal by (Zhang et al., 2019), who proposed the so-called FixUp initialization scheme. On top of introducing the learnable β parameters and the $L^{-1/(2k-2)}$ scaling for all layers k in each of the L residual branches, they set the initial weights for the last layer in each residual branch to zero and introduce scalar biases before every layer in the network. With these tricks, Zhang et al. show that FixUp can provide *almost* the same benefits as BN for ResNets in terms of trainability and generalization. Using a different derivation, De & Smith (2020) end up with a very similar solution to train ResNets without BN, which they term SkipInit. The key difference with FixUp is that the initial value for the learnable β parameter is set to be less than $1/\sqrt{L}$. As a result, SkipInit does not require the rescaling of initial weights in residual branches or setting weights to zero, which are considered crucial parts of the FixUp strategy (Zhang et al., 2019).

Also Shao et al. (2020) suggest to use a simple scaling strategy to replace BN in ResNets. They propose to use a slightly modified scaling of the form, $\boldsymbol{y} = \alpha \boldsymbol{x} + \beta f(\boldsymbol{x})$, where $\alpha^2 = 1 - \beta^2$ and $\beta^2 = 1/(l + c)$ for the l-th skip connection. Here, c is an arbitrary constant, which was eventually set to be the number of residual branches, L. For a single-layer ResNet ($l = c = 1$), this is equivalent to setting $\alpha = 1/\sqrt{2}$, as suggested by Balduzzi et al. (2017). However, the more general approach should assure that the outputs of residual branches are weighted similarly at the output of the network, independent of their depth.

4.2 Imitating Signal Propagation

Although the results of prior work look promising, there is still a performance gap compared to ResNets with BN. To close this gap, (Brock et al., 2021a) suggest studying the propagation of mean and variance through ResNets by means of so-called Signal Propagation Plots (SPPs). These SPPs simply visualize the squared mean and variance of the activations after each skip connection, as well as the variance at the end of every residual branch (before the skip connection).

To compute these values, the forward pass of the network must be slightly tweaked. To this end, we can define a new method or a function that simulates the forward pass and extracts the necessary statistics for each skip connection, as in listing 1.

This allows us to analyse the statistics for a single skip connection. By propagating a white noise signal (e.g., `torch.randn(1000, 3, 224, 224))`) through the entire ResNet, we obtain the data that allows us to produce SPPs. We refer to the end of this post for an example implementation (see appendix A.3) of a full Normalizer-Free ResNet (NF-ResNet) with `signal_prop` method.

Figure 4 provides an example of the SPPs for a pre-activation ResNet or v2 ResNets (cf. He et al., 2016a) with and without BN. The SPPs on the left clearly illustrate that BN transforms the exponential growth to a linear increase in ResNets, as described in theory (e.g. Balduzzi et al., 2017; De & Smith, 2020). When focusing on ResNets with BN (on the right of Figure 4), it is clear that

Listing 1: code to extract data for SPP

```
@torch.no_grad()
def signal_prop(self, x, dim=(0, -1, -2)):
    # forward code
    x = self.preact(x)
    skip = self.downsample(x)
    residual = self.residual_branch(x)
    out = residual + skip

    # compute necessary statistics
    out_mu2 = torch.mean(out.mean(dim) ** 2).item()
    out_var = torch.mean(out.var(dim)).item()
    res_var = torch.mean(residual.var(dim)).item()
    return out, (out_mu2, out_var, res_var)
```

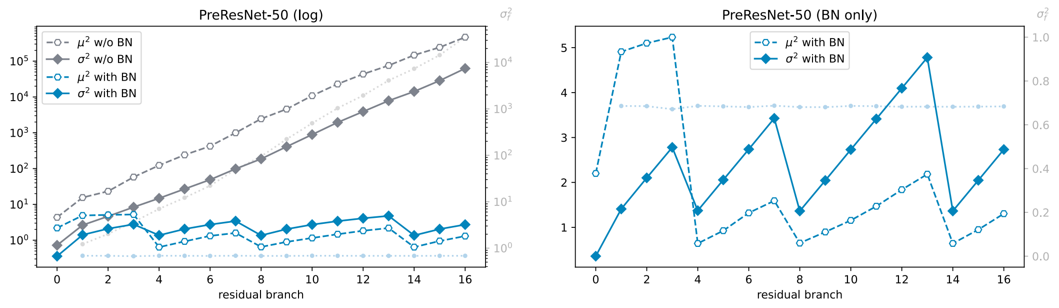

Figure 4: Example Signal Propagation Plots (SPPs) for a pre-activation (v2) ResNet-50 at initialization. SPPs show the squared mean (μ^2) and variance (σ^2) of the pre-activations after each skip connection (x-axis), as well as the variance of the residuals before the skip connection (σ_f^2, y-axis on the right). The left plot illustrates the difference between ResNets with and without BN layers. The plot on the right shows the same SPP for a ResNet with BN without the logarithmic scaling (cf. `BN->ReLU` in Figure 1 of Brock et al., 2021a). Note that ResNet-50 has four sub-nets with 3, 4, 6 and 3 skip connections, respectively.

mean and variance are reduced after every sub-net, each of which consists of a few skip connections. This reduction is due to the *pre-activation* block (BN + ReLU) that is inserted between every two sub-nets in these ResNets (remember the code snippet from earlier?).

The goal of NF-ResNets is to get rid of the BN layers in ResNets while preserving the characteristics visualized in the SPPs (Brock et al., 2021a). To get rid of the exponential variance increase in unnormalized ResNets, it suffices to set $\alpha = 1/\sqrt{\mathrm{Var}[\boldsymbol{x}]}$ in our modified formulation of ResNets. Here, $\mathrm{Var}[\boldsymbol{x}]$ is the variance over all samples in the dataset, such that the α scaling effectively mirrors the division by $\boldsymbol{\sigma}_{\mathcal{B}}$ in BN (assuming a large enough batch size). Unlike BN, however, the scaling in NF-ResNets is computed analytically for every skip connection. This is possible if the inputs to the network are properly normalized (i.e., have unit variance) and if the residual branch, f, properly preserves variance (i.e. is initialized correctly). The β parameter, on the other hand, is simply used as a hyper-parameter to directly control the variance increase after every skip connection.

It might be useful to point out that the proposed α scaling does not perfectly conform with our general formulation for ResNets. After all, the pre-activation layers mostly end up affecting only the inputs to the residual branch, such that $\boldsymbol{y} = \boldsymbol{x} + \beta f(\alpha \boldsymbol{x})$ (see code in appendix A for details). Only between the different sub-networks, which consist of multiple skip connections, the pre-activations are applied globally and the signal will be normalized. This also explains the variance drops in the SPPs for regular ResNets (see Figure 4). Note that this also means that the variance within sub-networks of an NF-ResNets will increase in the same way as for a ResNet with BN. Although

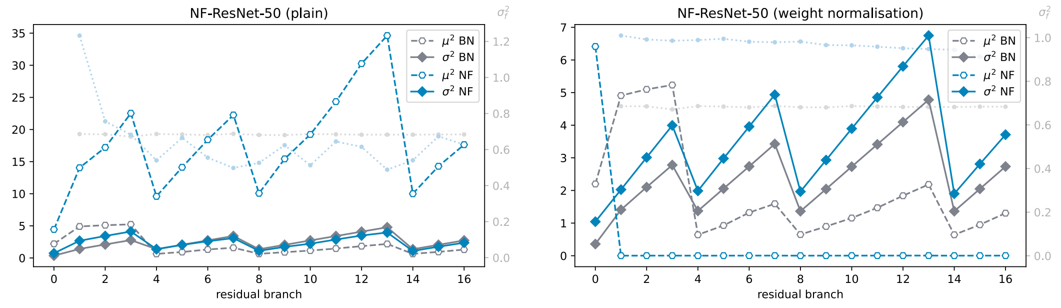

Figure 5: SPPs comparing an NF-ResNet-50 to a ResNet with BN at initialization. The NF-ResNet in the left plot only uses the α and β scaling parameters (cf. `NF, He Init` in Figure 2 Brock et al., 2021a). The right plot displays the behavior of an NF-ResNet with CWN (cf. `NF, Scaled WS` in Figure 2 Brock et al., 2021a). Note that the variance of the residuals in the right plot should give some insights as to why the curves do not overlap.

it would have been perfectly possible to maintain a steady variance, NF-ResNets are effectively designed to mimic the signal propagation due to BN layers in regular ResNets.

As can be seen on the left plot in Figure 5, a plain NF-ResNet effectively imitates the variance propagation of the baseline ResNet pretty accurately. The propagation of the squared mean in NF-ResNets, on the other hand, looks nothing like that from the BN model. After all, the considerations that lead to the scaling parameters only cover the variance propagation. On top of that, it turns out that the variance of the residual branches (right before it is merged with the skip connection) is not particularly steady. This indicates that the residual branches do not properly preserve variance, which is necessary for the analytic computations of α to be correct.

It turns out that both of these discrepancies can be resolved by introducing a variant of CWN (Huang et al., 2017b) to NF-ResNets. CWN simply applies WN after subtracting the weight mean from each weight vector, which ensures that every output has zero mean and that the variance of the weights is constant. Brock et al. (2021a) additionally rescale the normalized weights to account for the effect of activation functions (cf. Arpit et al., 2016). The effect of including the rescaled CWN in NF-ResNets is illustrated in the right part of Figure 5.

4.3 Performance

Empirically, Brock et al. (2021a) show that NF-ResNets with standard regularization methods perform on par with traditional ResNets that are using BN. An important detail[3] that is not apparent from the text, however, is that their baseline ResNets use the (standard) `BN->ReLU` order and not the `ReLU->BN` order, which served as the model for the signal propagation of NF-ResNets. This is also why the SPPs in Figure 5, which depict the `ReLU -> BN` order, do not perfectly overlap, unlike the figures in (Brock et al., 2021a).

Because BN does induce computational overhead, it seems natural to expect NF-ResNets to allow for more computationally efficient models. Therefore, Brock et al. (2021a) also compare NF-ResNets with a set of architectures that are optimized for efficiency. However, it turns out that some of these architectures do not play well with the weight normalization that is typically used in NF-ResNets. As a result, normalizer-free versions of EfficientNets (Tan & Le, 2019) lag behind their BN counterparts. When applied to (naive) RegNets (Radosavovic et al., 2020), however, the performance gap between with EfficientNets can be reduced by introducing the NF-ResNet scheme. In subsequent work, (Brock et al., 2021b) show that NF-ResNets in combination with gradient clipping are able to outperform similar networks with BN.

[3]`https://github.com/deepmind/deepmind-research/blob/ba761289c157fc151c7f06aa37b812d8100561db/nfnets/resnet.py#L158-L159`

5 DISCUSSION

NF-ResNets show that it is possible to build networks without BN that are able to achieve competitive prediction performance. It is not yet entirely clear whether the ideas of NF-ResNets could make BN entirely obsolete, however. Therefore, it should be interesting to take a closer look at what the limitations of NF-ResNets are. Assuming that the ideas in NF-ResNets can make BN (at least partly) obsolete, this should also provide some insights as to what the important factors are to explain the success of BN.

5.1 LIMITATIONS

First of all, the exact procedure for scaling residual branches is only meaningful for architectures that make use of simple additive skip connections. This means that it is not possible to directly apply the ideas behind NF-ResNets on arbitrary architectures to get rid of BN layers. Even similar architectures that make use a different kind of skip connection (e.g., DenseNets, Highway Networks, . . .) are probably not compatible with this exact approach. Furthermore, NF-ResNets still rely on (other) normalization methods to attain good performance — in contrast to what their name might suggest. Brock et al. (2021a) emphasize that they effectively do away with *activation normalization*, but they do rely on an adaptation of WN to replace BN. In this sense, it is arguable whether NF-ResNets are truly normalizer-free. Finally, some of the problems with BN are not resolved or reintroduced when building competitive NF-ResNets. E.g., there are still differences between training and testing when using plain dropout regularization, CWN still introduces a certain computational overhead during training, etc.

5.2 INSIGHTS

In the end, an NF-ResNet can be interpreted as consisting of different components that model parts of what BN normally does. For example, the α scaling factor used in NF-ResNets clearly models the division by the standard deviation of BN. It is also easy to see that the implicit regularization that is attributed to BN can be replaced by explicit regularization schemes. Furthermore, the mean subtraction in BN is practically implemented by the weight centering in CWN. Also, the scale-invariance of the weights due to BN is re-introduced through CWN. However, the input scale-invariance that BN introduces in each layer is lost when using CWN. When considering the entire residual branch (or network), however, α does enable some sort of scale-invariance for the entirety of this branch (or network). Finally, the affine transformation after the normalization in BN is modeled by scaling the result of CWN. Note that the affine shift does not need to be modeled explicitly, since CWN does not annihilate the regular bias parameters of the layers it acts upon, in contrast to BN.

Although the effects of BN on the forward pass seem to be modeled quite well by NF-ResNets, the effects on the backward pass seem to be largely ignored by Brock et al. (2021a). This might indicate that the performance differences might be explained by the effect of BN on the backward pass. Follow-up work by Brock et al. (2021b) also suggest that these effects might not be unimportant. After all, the gradient flow in NF-ResNets is only affected by the scaling factors, α and β, since CWN does not otherwise affect the gradients w.r.t. the inputs. Therefore, regular NF-ResNets do not have a gradient centering (Schraudolph, 1998) component, as can be found in BN layers. However, an adaptive gradient clipping scheme (Brock et al., 2021b) seems to provide an effective alternative to what BN does in the backward pass.

5.3 CONCLUSION

NF-ResNets show that it is possible to get rid of BN in ResNets without throwing away predictive performance. However, NF-ResNets still rely on WN schemes to make the models competitive with their BN counterparts. Therefore, it could be argued that NF-ResNets are not entirely *normalizer-free*. It almost seems as if NF-ResNets are an example of how BN can be imitated using different components, rather than how to get rid of it. This also means that it is hard to distil meaningful insights as to why/how BN works so well. One thing that this approach does make clear is that the backward dynamics due to BN should be part of the explanation.

In terms of the questions we set out to answer at the start, we could summarize as follows:

1. Why get rid of BN in the first place?
   The dependency on batch statistics does raise some concerns (sec. 2.3).
2. How (easy is it) to get rid of BN in ResNets?
   Although it is one of the reasons why ResNets made skip connections so popular, there are plenty of alternative tricks that can achieve similar effects (sec. 3.2).
3. Is BN going to become obsolete in the near future?
   It does not look like BN will disappear soon, because the techniques to get rid of BN are probably too specific to the ResNet architecture (sec. 5.1).
4. Does this allow us to gain insights into why BN works so well?
   NF-ResNets practically copy the forward dynamcis of ResNets with BN, which seems to suggest that the backward dynamics of BN play an important role(sec. 5.2).
5. Wait a second... Are they getting rid of normalization or just BN?
   Despite their name, NF-ResNets merely replace BN by another normalization technique (sec. 5.3).

TL;DR: NF-ResNets, rescaled ResNets with Centered Weight Normalization (CWN), can be used to imitate the forward pass of ResNets with BN, but they do not help much to explain what makes BN so successful.

ACKNOWLEDGMENTS

Special thanks go to Katharina Prinz, Niklas Schmidinger and the anonymous reviewer for their constructive feedback.

The ELLIS Unit Linz, the LIT AI Lab, the Institute for Machine Learning, are supported by the Federal State Upper Austria. IARAI is supported by Here Technologies. We thank the projects AI-MOTION (LIT-2018-6-YOU-212), AI-SNN (LIT-2018-6-YOU-214), DeepFlood (LIT-2019-8-YOU-213), Medical Cognitive Computing Center (MC3), INCONTROL-RL (FFG-881064), PRIMAL (FFG-873979), S3AI (FFG-872172), DL for GranularFlow (FFG-871302), AIRI FG 9-N (FWF-36284, FWF-36235), ELISE (H2020-ICT-2019-3 ID: 951847). We thank Audi.JKU Deep Learning Center, TGW LOGISTICS GROUP GMBH, Silicon Austria Labs (SAL), FILL Gesellschaft mbH, Anyline GmbH, Google, ZF Friedrichshafen AG, Robert Bosch GmbH, UCB Biopharma SRL, Merck Healthcare KGaA, Verbund AG, Software Competence Center Hagenberg GmbH, TÜV Austria, Frauscher Sensonic and the NVIDIA Corporation.

REFERENCES

Devansh Arpit, Yingbo Zhou, Bhargava Kota, and Venu Govindaraju. Normalization Propagation: A Parametric Technique for Removing Internal Covariate Shift in Deep Networks. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pp. 1168–1176, New York City, NY, USA, June 2016. PMLR. URL `https://proceedings.mlr.press/v48/arpitb16.html`. arXiv: 1603.01431.

Devansh Arpit, Victor Campos, and Yoshua Bengio. How to Initialize your Network? Robust Initialization for WeightNorm & ResNets. In *Advances in Neural Information Processing Systems*, volume 32, pp. 10902–10911, Vancouver, BC, Canada, December 2019. Curran Associates, Inc. URL `https://papers.nips.cc/paper/2019/hash/e520f70ac3930490458892665cda6620-Abstract.html`. arXiv: 1906.02341.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization. arXiv: 1607.06450, July 2016. URL `http://arxiv.org/abs/1607.06450`.

David Balduzzi, Marcus Frean, Lennox Leary, J. P. Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. The Shattered Gradients Problem: If resnets are the answer, then what is the question? In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pp. 342–350, Sydney, Australia, June 2017. PMLR. URL `http://proceedings.mlr.press/v70/balduzzi17b.html`.

Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. Understanding Batch Normalization. In *Advances in Neural Information Processing Systems*, volume 31, pp. 7694–7705, Montréal, QC, Canada, December 2018. Curran Associates, Inc. URL `https://proceedings.neurips.cc/paper/2018/hash/36072923bfc3cf47745d704feb489480-Abstract.html`. arXiv: 1806.02375.

Andrew Brock, Soham De, and Samuel L. Smith. Characterizing signal propagation to close the performance gap in unnormalized ResNets. In *International Conference on Learning Representations*, volume 9, virtual, March 2021a. URL `https://openreview.net/forum?id=IX3Nnir2omJ`. arXiv: 2101.08692.

Andy Brock, Soham De, Samuel L. Smith, and Karen Simonyan. High-Performance Large-Scale Image Recognition Without Normalization. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pp. 1059–1071, virtual, June 2021b. PMLR. URL `https://proceedings.mlr.press/v139/brock21a.html`. arXiv: 2102.06171.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In *International Conference on Learning Representations*, volume 4, San Juan, Puerto Rico, February 2016. URL `http://arxiv.org/abs/1511.07289`. arXiv: 1511.07289.

Soham De and Samuel L. Smith. Batch Normalization Biases Residual Blocks Towards the Identity Function in Deep Networks. In *Advances in Neural Information Processing Systems*, volume 33, pp. 19964–19975, virtual, December 2020. Curran Associates, Inc. URL `https://proceedings.neurips.cc//paper/2020/hash/e6b738eca0e6792ba8a9cbcba6c1881d-Abstract.html`. arXiv: 2002.10444.

Igor Gitman and Boris Ginsburg. Comparison of Batch Normalization and Weight Normalization Algorithms for the Large-scale Image Classification. arXiv: 1709.08145, October 2017. URL `http://arxiv.org/abs/1709.08145`.

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*, volume 9 of *Proceedings of Machine Learning Research*, pp. 249–256, Sardinia, Italy, May 2010. PMLR. URL `http://proceedings.mlr.press/v9/glorot10a.html`.

Boris Hanin and David Rolnick. How to Start Training: The Effect of Initialization and Architecture. In *Advances in Neural Information Processing Systems*, volume 31, pp. 571–581, Montréal, QC, Canada, 2018. Curran Associates, Inc. URL `https://proceedings.neurips.cc/paper/2018/hash/d81f9c1be2e08964bf9f24b15f0e4900-Abstract.html`. arXiv: 1803.01719.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034, Santiago, Chile, December 2015. IEEE. doi: 10.1109/ICCV.2015.123. URL `https://openaccess.thecvf.com/content_iccv_2015/html/He_Delving_Deep_into_ICCV_2015_paper.html`. arXiv: 1502.01852.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (eds.), *Computer Vision – ECCV 2016*, Lecture Notes in Computer Science, pp. 630–645, Amsterdam, Netherlands, September 2016a. Springer International Publishing. ISBN 978-3-319-46493-0. doi: 10.1007/978-3-319-46493-0_38. arXiv: 1603.05027.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Las Vegas, NV, USA, June 2016b. IEEE. ISBN 978-1-4673-8851-1. doi: 10.1109/CVPR.2016.90. URL `https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html`. arXiv:1502.01852.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9 (8):1735–1780, November 1997. ISSN 0899-7667, 1530-888X. doi: 10.1162/neco.1997.9.8.1735. URL https://www.mitpressjournals.org/doi/abs/10.1162/neco.1997.9.8.1735.

Elad Hoffer, Ron Banner, Itay Golan, and Daniel Soudry. Norm matters: efficient and accurate normalization schemes in deep networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 31, pp. 2160–2170, Montréal, QC, Canada, December 2018. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2018/file/a0160709701140704575d499c997b6ca-Paper.pdf. arXiv: 1803.01814.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, Honolulu, Hawaii, USA, July 2017a. IEEE. doi: 10.1109/CVPR.2017.243. URL https://openaccess.thecvf.com/content_cvpr_2017/html/Huang_Densely_Connected_Convolutional_CVPR_2017_paper.html. ISSN: 1063-6919.

Lei Huang, Xianglong Liu, Yang Liu, Bo Lang, and Dacheng Tao. Centered Weight Normalization in Accelerating Training of Deep Neural Networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2822–2830, Venice, Italy, October 2017b. IEEE. doi: 10.1109/ICCV.2017.305. URL https://openaccess.thecvf.com/content_iccv_2017/html/Huang_Centered_Weight_Normalization_ICCV_2017_paper.html.

Sergey Ioffe. Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models. In *Advances in Neural Information Processing Systems*, volume 30, pp. 1945–1953, Long Beach, CA, USA, December 2017. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2017/hash/c54e7837e0cd0ced286cb5995327d1ab-Abstract.html. arXiv:1702.03275.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, pp. 448–456, Lille, France, June 2015. PMLR. URL http://proceedings.mlr.press/v37/ioffe15.html. arXiv: 1502.03167.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-Normalizing Neural Networks. In *Advances in Neural Information Processing Systems*, volume 30, pp. 971–980, Long Beach, CA, USA, December 2017. Curran Associates, Inc. arXiv: 1706.02515.

Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. Efficient BackProp. In Genevieve B. Orr and Klaus-Robert Müller (eds.), *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science, pp. 9–50. Springer, Berlin, Heidelberg, 1 edition, August 1998. ISBN 978-3-540-49430-0. doi: 10.1007/3-540-49430-8_2. URL https://doi.org/10.1007/3-540-49430-8_2.

Xiang Li, Shuo Chen, Xiaolin Hu, and Jian Yang. Understanding the Disharmony Between Dropout and Batch Normalization by Variance Shift. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2682–2690, Long Beach, CA, USA, June 2019. IEEE. doi: 10.1109/CVPR.2019.00279. URL https://openaccess.thecvf.com/content_CVPR_2019/html/Li_Understanding_the_Disharmony_Between_Dropout_and_Batch_Normalization_by_Variance_CVPR_2019_paper.html. arXiv: 1801.05134.

Ping Luo, Xinjiang Wang, Wenqi Shao, and Zhanglin Peng. Towards Understanding Regularization in Batch Normalization. In *International Conference on Learning Representations*, volume 6, New Orleans, LA, USA, April 2019. URL https://openreview.net/forum?id=HJlLKjR9FQ. arXiv: 1809.00846.

Dmytro Mishkin and Jiri Matas. All you need is a good init. In *International Conference on Learning Representations*, volume 4, San Juan, Puerto Rico, February 2016. URL http://arxiv.org/abs/1511.06422. arXiv: 1511.06422.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral Normalization for Generative Adversarial Networks. In *International Conference on Learning Representations*, volume 6, Vancouver, BC, Canada, February 2018. URL https://openreview.net/forum?id=B1QRgziT-. arXiv: 1802.05957.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32, pp. 8026–8037, Vancouver, BC, Canada, December 2019. Curran Associates, Inc. arXiv: 1912.01703.

Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing Network Design Spaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10425–10433, Seattle, WA, USA, June 2020. IEEE. doi: 10.1109/CVPR42600.2020.01044. URL https://openaccess.thecvf.com/content_CVPR_2020/html/Radosavovic_Designing_Network_Design_Spaces_CVPR_2020_paper.html. arXiv: 2003.13678.

Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge, UK, 1996. ISBN 978-0-521-71770-0. doi: 10.1017/CBO9780511812651. URL https://www.cambridge.org/core/books/pattern-recognition-and-neural-networks/4E038249C9BAA06C8F4EE6F044D09C5C.

Tim Salimans and Diederik P. Kingma. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems*, volume 29, pp. 901–909, Barcelona, Spain, December 2016. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2016/hash/ed265bc903a5a097f61d3ec064d96d2e-Abstract.html. arXiv: 1602.07868.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How Does Batch Normalization Help Optimization? In *Advances in Neural Information Processing Systems*, volume 31, pp. 2483–2493, Montréal, QC, Canada, December 2018. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2018/hash/905056c1ac1dad141560467e0a99e1cf-Abstract.html. arXiv: 1805.11604.

Nicol N. Schraudolph. Centering Neural Network Gradient Factors. In Genevieve B. Orr and Klaus-Robert Müller (eds.), *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science, pp. 207–226. Springer, Berlin, Heidelberg, 1 edition, August 1998. ISBN 978-3-540-49430-0. doi: 10.1007/3-540-49430-8_11. URL https://doi.org/10.1007/3-540-49430-8_11.

Jie Shao, Kai Hu, Changhu Wang, Xiangyang Xue, and Bhiksha Raj. Is normalization indispensable for training deep neural network? In *Advances in Neural Information Processing Systems*, volume 33, pp. 13434–13444, virtual, December 2020. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2020/hash/9b8619251a19057cff70779273e95aa6-Abstract.html.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training Very Deep Networks. In *Advances in Neural Information Processing Systems*, volume 28, pp. 2377–2385, Montréal, QC, Canada, December 2015. Curran Associates, Inc. arXiv: 1507.06228.

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. arXiv: 1602.07261, August 2016. URL http://arxiv.org/abs/1602.07261.

Mingxing Tan and Quoc Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97, pp. 6105–6114, Long Beach, CA, USA, May 2019. PMLR. URL https://proceedings.mlr.press/v97/tan19a.html. arXiv: 1905.11946.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Improved Texture Networks: Maximizing Quality and Diversity in Feed-Forward Stylization and Texture Synthesis. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4105–4113, Honolulu, Hawaii, USA, July 2017. IEEE. doi: 10.1109/CVPR.2017.437. URL https://openaccess.thecvf.com/content_cvpr_2017/html/Ulyanov_Improved_Texture_Networks_CVPR_2017_paper.html. arXiv: 1701.02096.

Patrick van der Smagt and Gerd Hirzinger. Solving the Ill-Conditioning in Neural Network Learning. In Genevieve B. Orr and Klaus-Robert Müller (eds.), *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science, pp. 193–206. Springer, Berlin, Heidelberg, 1 edition, August 1998. ISBN 978-3-540-49430-0. doi: 10.1007/3-540-49430-8_10. URL https://doi.org/10.1007/3-540-49430-8_10.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, volume 30, pp. 5998–6008, Long Beach, CA, USA, December 2017. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html. arXiv: 1706.03762.

Neha Wadia, Daniel Duckworth, Samuel S. Schoenholz, Ethan Dyer, and Jascha Sohl-Dickstein. Whitening and Second Order Optimization Both Make Information in the Dataset Unusable During Training, and Can Reduce or Prevent Generalization. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pp. 10617–10629, virtual, July 2021. PMLR. URL http://proceedings.mlr.press/v139/wadia21a.html. arXiv: 2008.07545.

Yuxin Wu and Kaiming He. Group Normalization. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (eds.), *Computer Vision – ECCV 2018*, Lecture Notes in Computer Science, pp. 3–19, Munich, Germany, 2018. Springer International Publishing. ISBN 978-3-030-01261-8. doi: 10.1007/978-3-030-01261-8_1. URL https://openaccess.thecvf.com/content_ECCV_2018/html/Yuxin_Wu_Group_Normalization_ECCV_2018_paper.html. arXiv: 1803.08494.

Hongyi Zhang, Yann N. Dauphin, and Tengyu Ma. Fixup Initialization: Residual Learning Without Normalization. In *International Conference on Learning Representations*, volume 6, New Orleans, LA, USA, March 2019. URL https://openreview.net/forum?id=H1gsz30cKX. arXiv: 1901.09321.

A Extra Code Snippets

To facilitate the implementation of pre-residual ResNets in PyTorch (Paszke et al., 2019) and to give a full example of how to implement the signal propagation plotting, we provide additional code snippets.

A.1 Pre-activation ResNets

The first snippet implements skip connections according to (He et al., 2016a). The comments aim to highlight the differences with the ResNets from (He et al., 2016b), for which an implementation[4] is included in the Torchvision library.

```python
from torch import nn
from typing import Callable


class PreResidualBottleneck(nn.Module):

    expansion = 4

    def __init__(self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: nn.Module = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        norm_layer: Callable[[int], nn.Module] = None,
        no_preact: bool = False,  # additional argument
    ):
        super().__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d

        ### pre-activations ###
        preact_layers = [] if no_preact else [
            norm_layer(inplanes),
            nn.ReLU(),
        ]
        if downsample is None:
            self.preact = nn.Identity()
            residual_preact = preact_layers
        else:
            self.preact = nn.Sequential(*preact_layers)
            residual_preact = []
        ### pre-activations ###

        kernel_size = 3
        width = groups * (planes * base_width // 64)
        self.downsample = nn.Identity() if downsample is None else downsample
        self.residual_branch = nn.Sequential(
            *residual_preact,  # include residual pre-activations
            nn.Conv2d(inplanes, width, 1, bias=False),
            norm_layer(width), nn.ReLU(),
            nn.Conv2d(width, width, kernel_size, stride, padding=dilation,
```

[4] https://github.com/pytorch/vision/blob/v0.11.2/torchvision/models/resnet.py#L86-L141

```
                    dilation=dilation, groups=groups, bias=False),
                norm_layer(width), nn.ReLU(),
                nn.Conv2d(width, planes * self.expansion, 1, bias=False),
                # norm_layer(planes * self.expansion),
            )

    def forward(self, x):
        x = self.preact(x)  # compute global pre-activations
        skip = self.downsample(x)
        residual = self.residual_branch(x)
        # return torch.relu(residual + skip)
        return residual + skip
```

A.2 NF-ResNets

When comparing the code for a skip connection between an NF-ResNet and a regular ResNet with BN, we find that there are only a few minor changes. So much so that it is more efficient to consider the `diff` output than the full code.

```
@@ -4,3 +4,3 @@

-class PreResidualBottleneck(nn.Module):
+class NFResidualBottleneck(nn.Module):

@@ -16,3 +16,4 @@
         dilation: int = 1,
-        norm_layer: Callable[[int], nn.Module] = None,
+        alpha: float = 1.,
+        beta: float = 1.,
         no_preact: bool = False,  # additional argument
@@ -20,4 +21,3 @@
         super().__init__()
-        if norm_layer is None:
-            norm_layer = nn.BatchNorm2d
+        self.beta = beta

@@ -25,3 +25,3 @@
         preact_layers = [] if no_preact else [
-            norm_layer(inplanes),
+            Scaling(alpha),
             nn.ReLU(),
@@ -41,8 +41,8 @@
             *residual_preact,  # include residual pre-activations
-            nn.Conv2d(inplanes, width, 1, bias=False),
-            norm_layer(width), nn.ReLU(),
+            nn.Conv2d(inplanes, width, 1, bias=True),
+            nn.ReLU(),
             nn.Conv2d(width, width, kernel_size, stride, padding=dilation,
-                      dilation=dilation, groups=groups, bias=False),
-            norm_layer(width), nn.ReLU(),
-            nn.Conv2d(width, planes * self.expansion, 1, bias=False),
+                      dilation=dilation, groups=groups, bias=True),
+            nn.ReLU(),
+            nn.Conv2d(width, planes * self.expansion, 1, bias=True),
             # norm_layer(planes * self.expansion),
@@ -55,2 +55,2 @@
         # return torch.relu(residual + skip)
-        return residual + skip
+        return self.beta * residual + skip
```

The patch above shows that apart from removing the BN layers and introducing the α and β parameters, the BN layer in the pre-activation has to be replaced by the α scaling that is introduced in NF-ResNets. These changes are effectively everything that needs to be done. To be fair, this `Scaling` module is not standard in PyTorch, but it is easy enough to create it:

```
class Scaling(nn.Module):

    def __init__(self, scale: float):
        self.scale = scale

    def __repr__(self):
        return f"{self.__class__.__name__}({self.scale})"

    def forward(self, x):
        return self.scale * x
```

Putting everything together, including the `signal_prop` method introduced earlier (sec. 4.2, the resulting code should correspond to the following:

```
from torch import nn
from typing import Callable


class NFResidualBottleneck(nn.Module):

    expansion = 4

    def __init__(self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: nn.Module = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        alpha: float = 1.,
        beta: float = 1.,
        no_preact: bool = False,
    ):
        super().__init__()
        self.beta = beta

        ### pre-activations ###
        preact_layers = [] if no_preact else [
            Scaling(alpha),
            nn.ReLU(),
        ]
        if downsample is None:
            self.preact = nn.Identity()
            residual_preact = preact_layers
        else:
            self.preact = nn.Sequential(*preact_layers)
            residual_preact = []
        ### pre-activations ###

        kernel_size = 3
        width = groups * (planes * base_width // 64)
        self.downsample = nn.Identity() if downsample is None else downsample
        self.residual_branch = nn.Sequential(
            *residual_preact,
            nn.Conv2d(inplanes, width, 1, bias=True),
            nn.ReLU(),
            nn.Conv2d(width, width, kernel_size, stride, padding=dilation,
                      dilation=dilation, groups=groups, bias=True),
            nn.ReLU(),
            nn.Conv2d(width, planes * self.expansion, 1, bias=True),
        )

    def forward(self, x):
        x = self.preact(x)
        skip = self.downsample(x)
        residual = self.residual_branch(x)
        return self.beta * residual + skip

    @torch.no_grad()
    def signal_prop(self, x, dim=(0, -1, -2)):
        # forward code
        x = self.preact(x)
        skip = self.downsample(x)
        residual = self.residual_branch(x)
        out = self.beta * residual + skip

        # compute necessary statistics
        out_mu2 = torch.mean(out.mean(dim) ** 2).item()
        out_var = torch.mean(out.var(dim)).item()
        res_var = torch.mean(residual.var(dim)).item()
        return out, (out_mu2, out_var, res_var)
```

The code for a full NF-ResNet (with multiple multi-layer sub-nets) can be found in the code snippets in appendix A.3.

A.3 Multi-layer SPPs

In order to give an example of how to collect the SPP data for a multi-layer ResNet, the snippet below provides code for an NF-ResNet. For the sake of *brevity*, the implementation for CWN has been omitted here. This code is inspired by the `ResNet` implementation[5] from Torchvision. If you want to use this code, make sure that the `NFResidualBottleneck` module also provides a `signal_prop` method, as introduced in sec. 4.2.

```
class NFResidualNetwork(nn.Module):

    @staticmethod
    def initialisation(m: nn.Module):
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight)
            if m.bias is not None:
                nn.init.zeros_(m.bias)

    def __init__(self, layers: tuple, num_classes: int = 1000, beta: float = 1.):
        super().__init__()
        block = NFResidualBottleneck
        self._inplanes = 64
        self._expected_var = 1.
        self.beta = beta

        self.intro = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
```

[5]https://github.com/pytorch/vision/blob/v0.11.2/torchvision/models/resnet.py#L144-L249

```
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        )
        self.subnet1 = self._make_subnet(block, 64, layers[0], no_preact=True)
        self.subnet2 = self._make_subnet(block, 128, layers[1], stride=2)
        self.subnet3 = self._make_subnet(block, 256, layers[2], stride=2)
        self.subnet4 = self._make_subnet(block, 512, layers[3], stride=2)

        self.classifier = nn.Sequential(
            nn.ReLU(),
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Linear(512 * block.expansion, num_classes),
        )

        self.apply(self.initialisation)
        # self.apply(CentredWeightNormalization(dim=(1, 2, 3)))

    def _make_subnet(self, block, planes: int, num_layers: int,
                     stride: int = 1, no_preact: bool = False):
        downsample = None
        if stride != 1 or self._inplanes != planes * block.expansion:
            downsample = nn.Conv2d(self._inplanes, planes * block.expansion, 1, stri

        layers = []
        # compute expected variance analytically
        alpha = 1. / self._expected_var ** .5
        self._expected_var = 1. + self.beta ** 2
        layers.append(block(
            self._inplanes, planes, stride, downsample,
            alpha=alpha, beta=self.beta, no_preact=no_preact
        ))
        self._inplanes = planes * block.expansion
        for _ in range(1, num_layers):
            # track expected variance analytically
            alpha = 1. / self._expected_var ** .5
            self._expected_var += self.beta ** 2
            layers.append(block(
                self._inplanes, planes, alpha=alpha, beta=self.beta
            ))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.intro(x)
        x = self.subnet1(x)
        x = self.subnet2(x)
        x = self.subnet3(x)
        x = self.subnet4(x)
        return self.classifier(x)

    @torch.no_grad()
    def signal_prop(self, x, dim=(0, -1, -2)):
        x = self.intro(x)

        statistics = [(
            torch.mean(x.mean(dim) ** 2).item(),
            torch.mean(x.var(dim)).item(),
            float('nan'),
)]
for subnet in (self.subnet1, self.subnet2, self.subnet3, self.subnet4):
    for layer in subnet:
        x, stats = layer.signal_prop(x, dim)
        statistics.append(stats)

# convert list of tuples to tuple of lists
sp = tuple(map(list, zip(*statistics)))
return self.classifier(x), sp
```