CHARACTERIZING SIGNAL PROPAGATION TO CLOSE THE PERFORMANCE GAP IN UNNORMALIZED RESNETS

Anonymous authors
Paper under double-blind review

ABSTRACT

Batch Normalization is a key component in almost all state-of-the-art image classifiers, but it also introduces practical challenges: it breaks the independence between training examples within a batch, can be surprisingly expensive to compute, and often results in unexpected bugs. Building on recent theoretical analyses of deep ResNets at initialization, we propose a simple set of analysis tools to characterize signal propagation on the forward pass, and leverage these tools to design highly performant ResNets without activation normalization layers. Crucial to our success is an adapted version of the recently proposed Weight Standardization. Our analysis tools show how this technique preserves the signal in networks with ReLU or Swish activation functions by ensuring that the per-channel activation means do not grow with depth. Across a range of FLOP budgets, our networks attain performance competitive with the state-of-the-art EfficientNets on ImageNet.

1 INTRODUCTION

BatchNorm has become a core computational primitive in deep learning (Ioffe & Szegedy, 2015), and it is used in almost all state-of-the-art image classifiers (Tan & Le, 2019). A number of different benefits of BatchNorm have been identified. It smoothens the loss landscape (Santurkar et al., 2018), which allows us to train with larger learning rates (Bjorck et al., 2018), and the noise arising from the minibatch estimates of the batch statistics introduces implicit regularization (Luo et al., 2018). Crucially, recent theoretical work (Balduzzi et al., 2017; Zhang et al., 2019; De & Smith, 2020) has demonstrated that BatchNorm ensures good signal propagation at initialization in deep residual networks with identity skip connections (He et al., 2016b;a), and this benefit has enabled practitioners to train deep ResNets with hundreds or even thousands of layers.

However, BatchNorm also has many disadvantages. Its behavior is strongly dependent on the batch size (performing poorly when the batch size is too small or too large), and it introduces a discrepancy between the behaviour of the model during training and at inference time. BatchNorm also adds substantial implementation complexity in distributed training, which often introduces unexpected bugs or makes models difficult to replicate on different hardware. A number of alternative normalization layers have been proposed (Ba et al., 2016; Wu & He, 2018), but typically these alternatives generalize poorly or introduce their own drawbacks, such as added compute costs at inference.

Another line of work has sought to eliminate layers which normalize hidden activations entirely. A common trend is to initialize residual branches to output zeros (Goyal et al., 2017; Zhang et al., 2019; De & Smith, 2020; Bachlechner et al., 2020), which ensures that the signal is dominated by the skip path early in training. However while this strategy enables us to train deep ResNets with thousands of layers, it still degrades generalization when compared to well-tuned baselines (De & Smith, 2020). These simple initialization strategies are also not applicable to more complicated architectures like EfficientNets (Tan & Le, 2019), the current state of the art on ImageNet (Russakovsky et al., 2015).

This work seeks to establish a general recipe for training deep ResNets without normalization layers, which achieve test accuracy competitive with the state-of-the-art. Our contributions are as follows:

- We introduce Signal Propagation Plots (SPPs): a simple set of visualizations which help us inspect signal propagation at initialization on the forward pass in deep residual networks. Leveraging these SPPs, we show how to design unnormalized ResNets which are constrained to have signal propagation properties similar to batch-normalized ResNets.

- We identify a key failure mode in unnormalized ResNets with ReLU or Swish activations. Because the mean output of these activation functions is positive, when they are used with Gaussian weights, the mean value of the hidden activations on each channel grow proportional to the network depth. To resolve this, we propose Scaled Weight Standardization, a minor modification of the recently proposed Weight Standardization (Qiao et al., 2019), which prevents the growth in the mean signal, leading to a substantial boost in performance.
- We apply our normalization-free network structure in conjunction with Scaled Weight Standardization to ResNets on ImageNet, where we for the first time attain performance which is comparable or better than batch-normalized ResNets on networks as deep as 288 layers.
- Finally, we apply our normalization-free approach to the RegNet architecture (Radosavovic et al., 2020). By combining this architecture with the compound scaling strategy proposed by Tan & Le (2019), we develop a class of models without normalization layers which are competitive with the current ImageNet state-of-the-art across a range of FLOP budgets.

2 BACKGROUND

Deep ResNets at initialization: The combination of BatchNorm (Ioffe & Szegedy, 2015) and skip connections (Srivastava et al., 2015; He et al., 2016a) has allowed practitioners to train deep ResNets with hundreds or thousands of layers. To understand this effect, a number of papers have analyzed signal propagation in normalized ResNets at initialization (Balduzzi et al., 2017; Yang et al., 2019). In a recent work, De & Smith (2020) showed that in normalized ResNets with Gaussian initialization, the activations on the ℓth residual branch are suppressed by factor of $O(\sqrt{\ell})$, relative to the scale of the activations on the skip path. This biases the residual blocks in deep ResNets towards the identity function at initialization, ensuring well-behaved gradients. In unnormalized networks, one can preserve this benefit by introducing a learnable scalar at the end of each residual branch, initialized to zero (Zhang et al., 2019; De & Smith, 2020; Bachlechner et al., 2020). This simple change is sufficient to train deep ResNets with thousands of layers without normalization. However, while this method is easy to implement and achieves excellent convergence on the training set, it still achieves lower test accuracies than normalized networks when compared to well-tuned baselines.

These insights from studies of batch-normalized ResNets are also supported by theoretical analyses of unnormalized networks (Taki, 2017; Yang & Schoenholz, 2017; Hanin & Rolnick, 2018). These works suggest that, in ResNets with identity skip connections, if the signal does not explode on the forward pass, the gradients will neither explode nor vanish on the backward pass. Hanin & Rolnick (2018) conclude that multiplying the hidden activations on the residual branch by a factor of $O(1/d)$ or less, where d denotes the network depth, is sufficient to ensure trainability at initialization.

Alternate normalizers: To counteract the limitations of BatchNorm in different situations, a range of alternative normalization schemes have been proposed, each operating on different components of the hidden activations. These include LayerNorm (Ba et al., 2016), InstanceNorm (Ulyanov et al., 2016), GroupNorm (Wu & He, 2018), and many more (Huang et al., 2020). While these alternatives remove the dependency on the batch size and typically work better than BatchNorm for very small batch sizes, they also introduce limitations of their own, such as introducing additional computational costs during inference time. Furthermore for image classification, these alternatives still tend to achieve lower test accuracies than well-tuned BatchNorm baselines. As one exception, we note that the combination of GroupNorm with Weight Standardization (Qiao et al., 2019) was recently identified as a promising alternative to BatchNorm in ResNet-50 (Kolesnikov et al., 2019).

3 SIGNAL PROPAGATION PLOTS

Although papers have recently theoretically analyzed signal propagation in ResNets (see Section 2), practitioners rarely empirically evaluate the scales of the hidden activations at different depths inside a specific deep network when designing new models or proposing modifications to existing architectures. By contrast, we have found that plotting the statistics of the hidden activations at different points inside a network, when conditioned on a batch of either random Gaussian inputs or training examples, can be extremely beneficial. This practice both enables us to immediately detect hidden bugs in our implementation before launching an expensive training run destined to fail, and also allows us to identify surprising phenomena which might be challenging to derive from scratch.

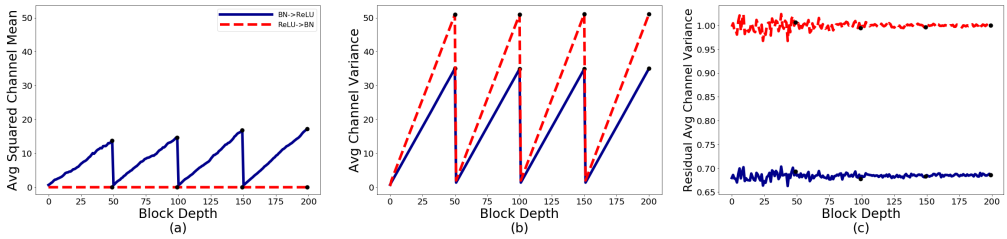


Figure 1: Signal Propagation Plot for a ResNetV2-600 with BatchNorm, ReLU nonlinearities, and He-initialized weights, in response to an $\mathcal{N}(0, 1)$ input at 512px resolution. Black dots indicate the end of a stage. Blue plots use the ordering BN-ReLU-Conv while red plots use ReLU-BN-Conv.

We therefore propose to formalize this good practice by introducing Signal Propagation Plots (SPPs), a simple graphical method for visualizing signal propagation on the forward pass in deep ResNets. We assume identity residual blocks of the form $x_{\ell+1} = f_\ell(x_\ell) + x_\ell$, where x_ℓ denotes the input to the ℓth block and f_ℓ denotes the function computed by the ℓth residual branch. We consider 4-dimensional input and output tensors with dimensions denoted by $NHWC$, where N denotes the batch dimension, C denotes the channels, and H and W denote the two spatial dimensions. To generate SPPs, we initialize a single set of weights according to the network initialization scheme, and then provide the network with a batch of input examples sampled from a unit Gaussian distribution. Then, we plot the following hidden activation statistics at the output of each residual block:

- Average Channel Squared Mean, computed as the square of the mean across the NHW axes, and then averaged across the C axis. In a network with good signal propagation, we would expect the mean activations on each channel, averaged across a batch of examples, to be close to zero. Importantly, we note that it is necessary to measure the averaged squared value of the mean, since the means of different channels may have opposite signs.
- Average Channel Variance, computed by taking the Channel Variance across the NHW axes, and then averaging across the C axis. We find this to generally be the most informative measure of the signal magnitude, and to clearly show signal explosion or attenuation.
- Average Channel Variance on the end of the residual branch, before merging with the skip path. This helps assess whether the layers on the residual branch are correctly initialized.

We explore several other possible choices of statistics one could measure in Appendix G, but we have found these three to be the most informative. We also experiment with feeding the network real data samples instead of random noise, but find that this step does not meaningfully affect the key trends. Finally, we emphasize that SPPs do not capture every property of signal propagation, and only consider the statistics of the forward pass. Despite this simplicity, we find SPPs surprisingly useful for analyzing deep ResNets.

As an example, in Figure 1 we present the SPP for a 600-layer pre-activation ResNet (He et al., 2016a)[1] with BatchNorm, ReLU, and He initialization (He et al., 2015), comparing BN-ReLU ordering (more standard) to ReLU-BN ordering. Immediately, several key patterns emerge. First, we note that the Average Channel Variance grows linearly with the depth in a given stage, and resets at each transition block to a fixed value close to 1. The linear growth arises because, at initialization, the variance of the activations satisfy $\mathrm{Var}(x_{\ell+1}) = \mathrm{Var}(x_\ell) + \mathrm{Var}(f_\ell(x_\ell))$, while BatchNorm ensures that the variance of the activations on the residual branch is independent of depth (De & Smith, 2020). The variance reset arises from the transition blocks, where the skip connection is replaced by a convolution operating on a normalized input, undoing any signal growth on the skip path.

With the BN-ReLU-Conv ordering, the Average Squared Channel Means display similar behavior, growing linearly with depth between transition blocks. This may seem surprising, since we expect BatchNorm to center the activations. However, with this ordering, the final convolution in a block receives a rectified input that has positive mean. As we show in the following section, this causes the output of the residual branch to also have positive mean, and explains why $\mathrm{Var}(f_\ell(x_\ell)) \approx 0.68$ for all depths ℓ. This "mean-shift" will be explicitly counteracted by normalization layers, enabling

[1]See Appendix E for an overview of ResNet blocks and their order of operations.

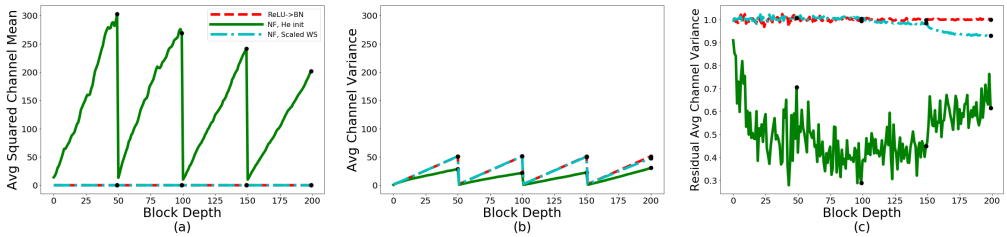


Figure 2: SPPs for three different variants of the ResNetV2-600 network (with ReLU activations). In red, we show a batch normalized network with ReLU-BN-Conv ordering. In green we show a normalizer-free network with He-initialized weights. In cyan, we show a normalizer-free network with Scaled Weight Standardization. We note that the SPPs for a normalizer-free network with Scaled Weight Standardization are almost identical to those for the batch normalized network.

stable training in normalized networks, but will have serious consequences when attempting to remove normalizer layers, as discussed below. In contrast, the ReLU-BN-Conv ordering trains equally stably while avoiding this mean-shift, with $\mathrm{Var}(f_\ell(x_\ell)) \approx 1$ for all ℓ.

4 Normalizer-Free ResNets (NF-ResNets)

With SPPs in hand to aid our analysis, we now seek to develop ResNet variants without normalization layers, which have good signal propagation, are stable during training, and reach test accuracies competitive with batch-normalized ResNets. We begin with two observations from Section 3. First, at standard initializations, BatchNorm downscales the input to each residual block by a factor proportional to the standard deviation of the input signal (De & Smith, 2020). Second, each residual block increases the variance of the signal by an approximately constant factor. We propose to mimic this effect by using residual blocks of the form $x_{\ell+1} = x_\ell + \alpha f_\ell(x_\ell/\beta_\ell)$, where x_ℓ denotes the input to the ℓth residual block and $f_\ell(\cdot)$ denotes the ℓth residual branch. We design the network such that:

- $f(\cdot)$, the function computed by the residual branch, is parameterized to be variance preserving at initialization, i.e., $\mathrm{Var}(f_\ell(z)) = \mathrm{Var}(z)$ for all ℓ. This constraint enables us to reason about the signal growth in the network, and estimate the variances analytically.
- β_ℓ is a fixed scalar, chosen as $\sqrt{\mathrm{Var}(x_\ell)}$, the expected empirical standard deviation of the activations x_ℓ at initialization. This ensures the input to $f_\ell(\cdot)$ has unit variance.
- α is a scalar hyperparameter which controls the rate of variance growth between blocks.

We compute the expected empirical variance at residual block ℓ analytically according to $\mathrm{Var}(x_\ell) = \mathrm{Var}(x_{\ell-1}) + \alpha^2$, with an initial expected variance of $\mathrm{Var}(x_0) = 1$, and we set $\beta_\ell = \sqrt{\mathrm{Var}(x_\ell)}$. A similar approach was proposed by Arpit et al. (2016) for non-residual networks. As noted in Section 3, the signal variance in normalized ResNets is reset at each transition layer due to the shortcut convolution receiving a normalized input. We mimic this reset by having the shortcut convolution in transition layers operate on (x_ℓ/β_ℓ) rather than x_ℓ, ensuring unit signal variance at the start of each stage ($\mathrm{Var}(x_{\ell+1}) = 1 + \alpha^2$ following each transition layer). For simplicity, we call residual networks employing this simple scaling strategy *Normalizer-Free ResNets* (NF-ResNets).

4.1 ReLU Activations Induce Mean Shifts

We plot the SPPs for Normalizer-Free ResNets (NF-ResNets) with $\alpha = 1$ in Figure 2. In green, we consider a simple NF-ResNet, which initializes the convolutions with Gaussian weights using He initialization (He et al., 2015). Although one might expect this simple recipe to be sufficient to achieve good signal propagation, we observe two unexpected features in practice. First, the average value of the squared channel mean grows rapidly with depth, achieving large values which exceed the average variance on the channel. This indicates a "mean shift", whereby the hidden activations for different training inputs (in this case different vectors sampled from the unit normal) are strongly correlated (Jacot et al., 2019). Second, as also observed for BN-ReLU-Conv networks in section 3, the scale of the empirical variances on the residual branch are consistently smaller than one.

To identify the origin of these effects, in Figure 7 (in Appendix F) we provide a similar SPP for a linearized version of ResNetV2-600 without ReLU activation functions. When the ReLU activations are removed, the averaged squared channel means remain close to zero for all block depths, and the empirical variance on the residual branch fluctuates around one. This motivates the following question: why might ReLU activations cause the scale of the mean activations on a channel to grow?

To develop an intuition for this phenomenon, consider the transformation $z = Wg(x)$, where W is arbitrary and fixed, and $g(\cdot)$ is an activation function that acts component-wise on iid inputs x such that $g(x)$ is also iid. Thus, $g(\cdot)$ can be any popular activation function like ReLU, tanh, SiLU, etc. Let $\mathbb{E}(g(x_i)) = \mu_g$ and $\mathrm{Var}(g(x_i)) = \sigma_g^2$ for all dimensions i. It is straightforward to show that the expected value and the variance of any unit i of the output $z_i = \sum_j^N W_{i,j} g(x_j)$ is given by:

$$\mathbb{E}(z_i) = N\mu_g \mu_{W_{i,\cdot}}, \quad \text{and} \quad \mathrm{Var}(z_i) = N\sigma_g^2(\sigma_{W_{i,\cdot}}^2 + \mu_{W_{i,\cdot}}^2), \tag{1}$$

where $\mu_{W_{i,\cdot}}$ and $\sigma_{W_{i,\cdot}}$ are the mean and standard deviation of the ith row of W:

$$\mu_{W_{i,\cdot}} = \frac{1}{N}\sum_j^N W_{i,j}, \quad \text{and} \quad \sigma_{W_{i,\cdot}}^2 = \frac{1}{N}\sum_j^N W_{i,j}^2 - \mu_{W_{i,\cdot}}^2. \tag{2}$$

Now consider $g(\cdot)$ to be the ReLU activation function, i.e., $g(x) = \max(x, 0)$. Then $g(x) \geq 0$, which implies that the input to the linear layer has positive mean (except for the edge case when all inputs are 0). In particular, notice that if $x_i \sim \mathcal{N}(0, 1)$ for all i, then $\mu_g = 1/\sqrt{2\pi} > 0$. Since we know that $\mu_g > 0$, if $\mu_{W_{i,\cdot}}$ is also non-zero, then the output of the transformation, z, will also exhibit a non-zero mean. Crucially, even if we sample W from a distribution centred around zero (as is typically done with standard Gaussian initializations like He initialization), any specific weight matrix drawn from this distribution will almost surely have a non-zero empirical mean, and consequently the outputs of the residual branches will have mean values shifted away from zero. This simple NF-ResNet model with He-initialized weights is unstable and performs poorly for deeper networks.

4.2 Scaled Weight Standardization

To prevent the emergence of a mean shift, and to ensure that the residual branch $f_\ell(\cdot)$ is variance preserving, we propose Scaled Weight Standardization (a minor modification of Weight Standardization (Qiao et al. (2019))). We re-parameterize the convolutional layers, by imposing,

$$\hat{W}_{i,j} = \gamma \cdot \frac{W_{i,j} - \mu_{W_{i,\cdot}}}{\sigma_{W_{i,\cdot}}\sqrt{N}}, \tag{3}$$

where the mean μ and variance σ are computed across the fan-in extent of the convolutional filters. We initialize the underlying parameters W from Gaussian weights, while γ is a fixed constant. As in Qiao et al. (2019), we impose this constraint throughout training as a differentiable operation in the forward pass of the network. Recalling equation 1, we can immediately see that the output of the transformation using Scaled WS, $z = \hat{W}g(x)$, has expected value $\mathbb{E}(z_i) = 0$ for all i, thus eliminating the mean shift. Furthermore, the variance $\mathrm{Var}(z_i) = \gamma^2\sigma_g^2$, meaning that for a correctly chosen γ, which depends on the non-linearity g, the layer will be variance preserving. Scaled Weight Standardization is cheap during training and free at inference, scales well (with the number of parameters rather than the number of activations), introduces no discrepancy in training and test behavior, and its implementation will not differ in distributed training. These desirable properties make it a compelling alternative for replacing BatchNorm.

The SPP of a normalizer-free ResNet-600 employing Scaled WS is shown in Figure 2 in cyan. As we can see, Scaled Weight Standardization eliminates the growth of the average channel squared means at initialization. Indeed, the SPPs are almost identical to the SPPs for a batch-normalized network employing the ReLU-BN-Conv ordering, shown in red. Note that we select the constant γ to ensure that the channel variance on the residual branch is close to one (discussed further below). The variance on the residual branch decays slightly near the end of the network due to zero padding.

4.3 Determining Nonlinearity-Specific Constants γ

The final ingredient we need is to determine the value of the gain γ, in order to ensure that the variances of the hidden activations on the residual branch are close to 1 at initialization. Note that

the value of γ will depend on the specific nonlinearity used in the network. We derive the value of γ assuming that the input x to the nonlinearity is sampled iid from $\mathcal{N}(0,1)$. For ReLU networks, this implies that the outputs, $g(x) = \max(x, 0)$, will be sampled from the rectified Gaussian distribution with variance $\sigma_g^2 = (1/2)(1 - (1/\pi))$ (Arpit et al., 2016). Since $\mathrm{Var}(\hat{W}g(x)) = \gamma^2\sigma_g^2$, to ensure $\mathrm{Var}(\hat{W}g(x)) = 1$, we set $\gamma = 1/\sigma_g = \frac{\sqrt{2}}{\sqrt{1-\frac{1}{\pi}}}$. While the $x \sim \mathcal{N}(0,1)$ will not typically be true unless the network width is large, we find this approximation to work well in practice.

For simple nonlinearities like the ReLU or tanh, the analytical variance of the non-linearity $g(x)$ when x is drawn from the unit normal may be known or easy to derive. For other nonlinearities, such as SiLU ((Elfwing et al., 2018; Hendrycks & Gimpel, 2016), recently popularized as Swish (Ramachandran et al., 2017)), analytically determining the variance can involve solving difficult integrals, or may even not have an analytical form. In practice, we find that it is sufficient to numerically approximate this value with the simple procedure of drawing many N dimensional vectors x, computing the empirical variance $\mathrm{Var}(g(x))$ for each vector, and taking the square root of the average of this empirical variance. We provide an example in Appendix D showing how this can be accomplished for any nonlinearity with just a few lines of code.

4.4 Other Building Blocks and Relaxed Constraints

Our method generally requires that any additional operations used in a network maintain good signal propagation, which means many common building blocks must be modified. As with selecting γ values, the necessary modification can be determined analytically or empirically. For example, the popular Squeeze-and-Excitation operation (S+E, Hu et al. (2018)), $y = sigmoid(MLP(pool(h)) * h$, involves multiplication by an activation in $[0, 1]$, and will tend to attenuate the signal and make the model unstable. This attenuation can clearly be seen in the SPP of a normalizer-free ResNet using these blocks (see Figure 9 in Appendix F). If we examine this operation in isolation using our simple numerical procedure explained above, we find that the expected variance is 0.5 (for unit normal inputs), indicating that we simply need to multiply the output by 2 to recover good signal propagation. We empirically verified that this simple change is sufficient to restore training stability.

In practice, we find that either a similarly simple modification to any given operation is sufficient to maintain good signal propagation, or that the network is sufficiently robust to the degradation induced by the operation to train well without modification. We also explore the degree to which we can relax our constraints and still maintain stable training. As an example of this, to recover some of the expressivity of a normal convolution, we introduce learnable affine gains and biases to the Scaled WS layer (the gain is applied to the weight, while the bias is added to the activation, as is typical). While we could constrain these values to enforce good signal propagation by, for example, scaling the output y with a scalar proportional to the values of the gains, we find that this is not necessary for stable training, and that stability is not impacted when these parameters vary freely. Relatedly, we find that employing a learnable scalar multiplier at the end of the residual branch initialized to 0 (Goyal et al., 2017; De & Smith, 2020) helps when training networks over 150 layers, even if we ignore this modification when computing β_ℓ. In our final models, we employ several such relaxations without loss of training stability. We provide detailed explanations for each operation and any modifications we make in Appendix C (also detailed in our model code in Appendix D).

4.5 Summary

In summary, the core recipe for a Normalizer-Free ResNet (NF-ResNet) is:

1. Compute and forward propagate the expected signal variance β_ℓ^2, which grows by α^2 at each residual block ($\beta_0 = 1$). Downscale the input to each residual block by β_ℓ.
2. Additionally, downscale the input to the convolution on the skip path in transition blocks by β_ℓ, and reset $\beta_{\ell+1} = 1 + \alpha^2$ following a transition block.
3. Employ Scaled Weight Standardization in all convolutional layers, computing γ, the gain specific to the activation function $g(x)$, as the reciprocal of the expected standard deviation, $\frac{1}{\sqrt{\mathrm{Var}(g(x))}}$, assuming $x \sim \mathcal{N}(0,1)$.

Code is provided in Appendix D for a reference Normalizer-Free network.

5 EXPERIMENTS

5.1 AN EMPIRICAL EVALUATION ON RESNETS

	NF-ResNets (ours)		BN-ResNets	
	Unregularized	Regularized	Unregularized	Regularized
ResNet-50	75.82 ± 0.1	76.79 ± 0.1	76.84 ± 0.1	76.36 ± 0.1
ResNet-101	77.14 ± 0.1	78.40 ± 0.1	78.03 ± 0.1	78.15 ± 0.1
ResNet-152	77.63 ± 0.1	79.13 ± 0.1	78.59 ± 0.2	78.78 ± 0.1
ResNet-200	77.92 ± 0.2	79.55 ± 0.1	78.97 ± 0.2	79.16 ± 0.1
ResNet-288	78.06 ± 0.1	79.50 ± 0.1	78.76 ± 0.1	79.54 ± 0.1

Table 1: ImageNet Top-1 Accuracy (%) for Normalizer-Free ResNets and Batch-Normalized ResNets on ImageNet, with and without additional regularization (stochastic depth and dropout). Results are given as the median accuracy $\pm$ the standard deviation across 5 random seeds.

We begin by investigating the performance of Normalizer-Free pre-activation ResNets on the ILSVRC dataset (Russakovsky et al., 2015) relative to batch-normalized ResNets. We use a training setup based on Goyal et al. (2017), and train our models using SGD (Robbins & Monro, 1951) with Nesterov's Momentum (Nesterov, 1983; Sutskever et al., 2013) for 90 epochs with a batch size of 1024 and a learning rate which warms up from zero to 0.4 over the first 5 epochs, then decays to zero using cosine annealing (Loshchilov & Hutter, 2016). We employ standard baseline preprocessing (sampling and resizing distorted bounding boxes, along with random flips), weight decay of 5e-5, and label smoothing of 0.1 (Szegedy et al., 2016a). For Normalizer-Free ResNets (NF-ResNets), we choose $\alpha = 0.2$ via a small sweep, and employ SkipInit as discussed above.

We find that without any additional regularization, our unnormalized models achieve higher training accuracies and lower test accuracies than their batch-normalized counterparts, likely due to the known regularization effect of BatchNorm (Hoffer et al., 2017; Luo et al., 2018). We therefore introduce stochastic depth (Huang et al., 2016) with a rate of 0.1, and Dropout (Srivastava et al., 2014) before the final linear layer with a drop probability of 0.25. We note that adding this same regularization does not substantially impact the performance of the normalized ResNets in our setup, suggesting that BatchNorm is indeed already providing some regularization benefit.

In Table 1 we compare performance of our networks (NF-ResNets) against the baseline (BN-ResNets), across a wide range of network depths. After introducing additional regularization, NF-ResNets achieve competitive performance across all network depths, with our regularized NF-ResNet-288 achieving top-1 accuracy of 79.5%. However, some of the 288 layer normalizer-free models undergo training collapse at the chosen learning rate, but only when unregularized. While we can remove this instability by reducing the learning rate to 0.2, this comes at the cost of test accuracy. We investigate and discuss this observed failure mode in Appendix A.

5.2 DESIGNING PERFORMANT NORMALIZER-FREE NETWORKS

We now turn our attention to developing unnormalized networks which are competitive with the state-of-the-art EfficientNet model family across a range of FLOP budgets (Tan & Le, 2019), We focus primarily on the small budget regime (EfficientNets B0-B4), but report results for B5 and hope to extend our investigation to larger variants in future work.

First, we apply Scaled WS and our Normalizer-Free structure directly to the EfficentNet backbone.[2] While we succeed in training these networks stably without normalization, we find that even after extensive tuning our Normalizer-Free EfficientNets still substantially underperform their batch-normalized baselines. For example, our normalization free B0 variant achieves 73.5% top-1, a 3.2% absolute degradation relative to the baseline. We hypothesize that this is because Weight Standardization imposes a very strong constraint on depth-wise convolutions (which have an input channel count of 1), and that this may remove a substantial fraction of their expressivity. This is in line with

[2]We were unable to train EfficientNets using SkipInit (De & Smith, 2020; Bachlechner et al., 2020). We speculate this may be because the EfficientNet backbone contains both residual and non-residual components.

the empirical observation that removing Scaled WS from the depth-wise convolutions improves the test accuracy, although this comes at the cost of training stability.

To overcome the potentially poor interactions between Scaled Weight Standardization and depthwise convolutions, we next investigate the RegNet model family (Radosavovic et al., 2020). RegNets are slightly modified variants of ResNeXts (Xie et al., 2017), developed via manual architecture search. Crucially, RegNets employ grouped convolutions, which we anticipate are more compatible with Scaled WS than depthwise convolutions, since the fraction of the degrees of freedom in the model weights remaining after the standardization operation is higher.

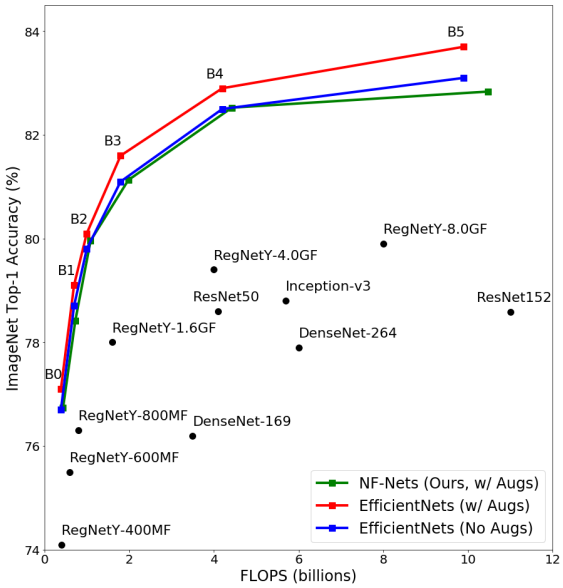


Figure 3: ImageNet top-1 accuracy versus FLOPs. NF-Nets accuracy shown with CutMix+MixUp.

We develop a new base model by taking the 0.4B FLOP RegNet variant, and making several architectural changes (see Appendix C), which cumulatively substantially improve the model performance. In order to assess the performance of our normalization free RegNet across a range of FLOP budgets, we next apply the EfficientNet compound scaling approach (increasing the width, depth, and input resolution in tandem) to obtain model variants at a range of approximate FLOPs targets. Denoting these models NF-Nets, we train variants B0-B5 (analogous to EfficientNet variants) using both baseline preprocessing and combined CutMix (Yun et al., 2019) and MixUp (Zhang et al., 2017) augmentation. We report the results in Figure 3 and Table 2 in Appendix A.

NF-Nets with CutMix (Yun et al., 2019) and MixUp (Zhang et al., 2017) attain comparable performance to EfficientNets with baseline preprocessing, while mildly underperforming EfficientNets with advanced data augmentations (AutoAugment and RandAugment, (Cubuk et al., 2019; 2020)). We adopt CutMix+MixUp training as we find that both AutoAugment and RandAugment degrade performance and stability for NF-Nets. We hypothesize that this occurs because these stronger augmentations change the statistics of the dataset in a way that negatively impacts signal propagation. To support this claim, we note that inserting a single batch-normalization layer after the first convolution in an NF-Net removes these instabilities and enables us to train stably with either AutoAugment or RandAugment (although this does not result in improved performance on the test set). This observation highlights that many advanced techniques like Auto-Augment were designed explicitly to optimize the performance of batch-normalized models, and may be accordingly better suited. We anticipate that the performance of NF-Nets could be further increased by devoting similar computational resources to re-tuning such strategies using NF-Nets, but leave this to future work.

6 Conclusion

We introduce Normalizer-Free Networks, a simple approach for designing ResNet models which do not require normalization layers. Across a range of FLOP budgets, our models achieve performance competitive with the state-of-the-art EfficientNets on ImageNet. Meanwhile, our empirical analysis of signal propagation suggests that batch normalization resolves two key failure modes at initialization in deep ResNets. First, it suppresses the scale of the hidden activations on the residual branch, preventing signal explosion. Second, it prevents the mean squared scale of the activations on each channel from exceeding the variance of the activations between examples. Our Normalizer-Free Networks were carefully designed to resolve both of these failure modes.

REFERENCES

Devansh Arpit, Yingbo Zhou, Bhargava Kota, and Venu Govindaraju. Normalization propagation: A parametric technique for removing internal covariate shift in deep networks. In *International Conference on Machine Learning*, pp. 1168–1176, 2016.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Thomas Bachlechner, Bodhisattwa Prasad Majumder, Huanru Henry Mao, Garrison W Cottrell, and Julian McAuley. Rezero is all you need: Fast convergence at large depth. *arXiv preprint arXiv:2003.04887*, 2020.

David Balduzzi, Marcus Frean, Lennox Leary, JP Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. The shattered gradients problem: If resnets are the answer, then what is the question? In *International Conference on Machine Learning*, pp. 342–350, 2017.

Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. Understanding batch normalization. In *Advances in Neural Information Processing Systems*, pp. 7694–7705, 2018.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 113–123, 2019.

Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 702–703, 2020.

Soham De and Samuel L Smith. Batch normalization biases deep residual networks towards shallow paths. *arXiv preprint arXiv:2002.10444*, 2020.

Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks*, 107:3–11, 2018.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of Wasserstein GANs. In *NIPS*, 2017.

Boris Hanin and David Rolnick. How to start training: The effect of initialization and architecture. In *Advances in Neural Information Processing Systems*, pp. 571–581, 2018.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy. *Nature*, 585(7825):357–362, Sep 2020. ISSN 1476-4687.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016b.

Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 558–567, 2019.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pp. 1731–1741, 2017.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7132–7141, 2018.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pp. 646–661. Springer, 2016.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. Normalization techniques in training dnns: Methodology, analysis and application. *arXiv preprint arXiv:2009.12836*, 2020.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

Arthur Jacot, Franck Gabriel, and Clément Hongler. Freeze and chaos for dnns: an ntk view of batch normalization, checkerboard and boundary effects. *arXiv preprint arXiv:1907.05715*, 2019.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pp. 971–980, 2017.

Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Large scale learning of general visual representations for transfer. *arXiv preprint arXiv:1912.11370*, 2019.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Ping Luo, Xinjiang Wang, Wenqi Shao, and Zhanglin Peng. Towards understanding regularization in batch normalization. *arXiv preprint arXiv:1809.00846*, 2018.

Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 116–131, 2018.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. In *ICLR*, 2018.

Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. *Doklady AN USSR*, pp. (269), 543–547, 1983.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pp. 8026–8037, 2019.

Boris Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, pp. 4(5):1–17, 1964.

Siyuan Qiao, Huiyu Wang, Chenxi Liu, Wei Shen, and Alan Yuille. Weight standardization. *arXiv preprint arXiv:1903.10520*, 2019.

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. 10428–10436, 2020.

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pp. 22(3):400–407, 1951.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, and Michael Bernstein. ImageNet large scale visual recognition challenge. *IJCV*, 115:211–252, 2015.

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*, pp. 901–909, 2016.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pp. 2483–2493, 2018.

Samuel L Smith, Erich Elsen, and Soham De. On the generalization benefit of noise in stochastic gradient descent. *arXiv preprint arXiv:2006.15081*, 2020.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pp. 1139–1147, 2013.

C Szegedy, V Vanhoucke, S Ioffe, J Shlens, and Z Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016a.

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv preprint arXiv:1602.07261*, 2016b.

Masato Taki. Deep residual networks and weight initialization. *arXiv preprint arXiv:1709.02956*, 2017.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pp. 6105–6114, 2019.

Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy. In *Advances in Neural Information Processing Systems*, pp. 8252–8262, 2019.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 3–19, 2018.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500, 2017.

Yunyang Xiong, Hanxiao Liu, Suyog Gupta, Berkin Akin, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Vikas Singh, and Bo Chen. Mobiledets: Searching for object detection architectures for mobile accelerators. *arXiv preprint arXiv:2004.14525*, 2020.

Ge Yang and Samuel Schoenholz. Mean field residual networks: On the edge of chaos. In *Advances in neural information processing systems*, pp. 7103–7114, 2017.

Greg Yang, Jeffrey Pennington, Vinay Rao, Jascha Sohl-Dickstein, and Samuel S Schoenholz. A mean field theory of batch normalization. *arXiv preprint arXiv:1902.08129*, 2019.

Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 6023–6032, 2019.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *arXiv preprint arXiv:1901.09321*, 2019.

APPENDIX A EXPERIMENT DETAILS

Model	#FLOPs	#Params	Top-1 w/o Augs	Top-1 w/ Augs
NFNet-B0	0.44B	8.3M	76.7 ± 0.2	76.5 ± 0.2
EfficientNet-B0	0.39B	5.3M	76.7	77.1
RegNetY-400MF	0.40B	4.3M	74.1	–
NFNet-B1	0.73B	9.8M	78.3 ± 0.3	78.4 ± 0.1
EfficientNet-B1	0.70B	7.8M	78.7	79.1
RegNetY-600MF	0.60B	6.1M	75.5	–
RegNetY-800MF	0.80B	6.3M	76.3	–
MobileNet (Howard et al., 2017)	0.60B	4.2M	70.6	–
MobileNetv2 (Sandler et al., 2018)	0.59B	6.9M	74.7	–
ShuffleNetv2 (Ma et al., 2018)	0.59B	–	74.9	–
NFNet-B2	1.09B	13.4M	78.6 ± 0.4	80.0 ± 0.1
EfficientNet-B2	1.00B	9.2M	79.8	80.1
NFNet-B3	1.98B	17.6M	80.5 ± 0.1	81.1 ± 0.1
EfficientNet-B3	1.80B	12.0M	81.1	81.6
NFNet-B4	4.43B	28.5M	81.4 ± 0.1	82.5 ± 0.1
EfficientNet-B4	4.20B	19.0M	82.5	82.9
RegNetY-4.0GF	4.00B	20.6M	79.4	–
ResNet50	4.10B	26.0M	76.8	78.6
DenseNet-169 (Huang et al., 2017)	3.50B	14.0M	76.2	–
NFNet-B5	10.48B	47.5M	81.7 ± 0.2	82.8 ± 0.1
EfficientNet-B5	9.90B	30.0M	83.1	83.7
RegNetY-12GF	12.10B	51.8M	80.3	–
ResNet152	11.00B	60.0M	78.6	–
Inception-v4 (Szegedy et al., 2016b)	13.00B	48.0M	80.0	–

Table 2: ImageNet Top-1 Accuracy (%) comparison for NF-Nets and recent state-of-the-art models. "w/ Augs" refers to accuracy with advanced augmentations: for EfficientNets, this is with AutoAugment or RandAugment. For NF-Nets, this is with CutMix + MixUp. NF-Nets results are reported as the median and standard deviation across 5 random seeds.

A.1 STABILITY, LEARNING RATES, AND BATCH SIZES

Previous work (Goyal et al., 2017) has established a fairly robust linear relationship between the optimal learning rate (or highest stable learning rate) and batch size for Batch-Normalized ResNets. As noted in Smith et al. (2020), we also find that this relationship breaks down past batch size 1024 for our unnormalized ResNets, as opposed to 2048 or 4096 for normalized ResNets. Both the optimal learning rate and the highest stable learning rate decrease for higher batch sizes. This also appears to correlate with depth: when not regularized, our deepest layers are not always stable with a learning rate of 0.4. While we can mitigate this collapse by reducing the learning rate for deeper nets, this introduces additional tuning expense and is clearly undesirable. It is not presently clear why regularization aids in stability; we leave investigation of this phenomenon to future work.

Taking a closer look at collapsed networks, we find that even though their outputs have exploded (becoming large enough to go NaN), their weight magnitudes are not especially large, even if we remove our relaxed affine transforms and train networks whose layers are purely weight-standardized. The singular values of the weights, however, end up poorly conditioned, meaning that the Lipschitz constant of the network can become quite large, an effect which Scaled WS does not prevent. One might consider adopting one of the many techniques from the GAN literature to regularize or constrain this constant (Gulrajani et al., 2017; Miyato et al., 2018), but as we will show in the following section this added complexity and expense is not necessary to develop performant unnormalized networks.

This collapse highlights an important limitation of our approach, and of SPPs: as SPPs only show signal prop for a given state of a network (i.e. at init), no guarantees are provided far from initialization. This fact drives us to prefer parameterizations like Scaled WS rather than solely relying on initialization strategies, and highlights that while good signal propagation is generally necessary for stable optimization, it is not always sufficient.

APPENDIX B MODIFIED BUILDING BLOCKS

In order to maintain good signal propagation in our Normalizer-Free models, we must ensure that any architectural modifications do not compromise our model's conditioning, as we cannot rely on activation normalizers to automatically correct for such changes. However, our models are not so fragile as to be unable to handle slight relaxations in this realm. We leverage this robustness to improve model expressivity and to incorporate known best practices for model design.

B.1 AFFINE GAINS AND BIASES

First, we add affine gains and biases to our network, similar to those used by activation normalizers. These are applied as a vector gain, each element of which multiplies a given output unit of a reparameterized convolutional weight, and a vector bias, which is added to the output of each convolution. We also experimented with using these as a separate affine transform applied before the ReLU, but moved the parameters next to the weight instead to enable constant-folding for inference. As is common practice with normalizer parameters, we do not weight decay or otherwise regularize these weights.

Even though these parameters are allowed to vary freely, we do not find that they are responsible for training instability, even in networks where we observe collapse. Indeed, we find that for settings which collapse (typically due to learning rates being too high), removing the affine transform has no impact on stability. As discussed in Appendix A, we observe that model instability arises as a result of the collapse of the spectra of the weights, rather than any consequence of the affine gains and biases.

B.2 STOCHASTIC DEPTH

We incorporate Stochastic Depth (Huang et al., 2016), where the output of the residual branch of a block is randomly set to zero during training. This is often implemented such that if the block is kept, its value is downscaled by the keep probability. We remove this downscaling factor to help maintain signal propagation when the signal is kept, but otherwise do not find it necessary to modify this block.

While it is possible that we might have an example where many blocks are dropped and signals are attenuated, in practice we find that, as with affine gains, removing Stochastic Depth does not improve stability, and adding it does not reduce stability. One might also consider a slightly more principled variant of Stochastic Depth in this context, where the skip connection is upscaled by $1+\alpha$ if the residual branch is dropped, resulting in the variance growing as expected, but we did not find this strategy necessary.

B.3 SQUEEZE AND EXCITE LAYERS

As mentioned in Section 4.4, we incorporate Squeeze and Excitation layers (Hu et al., 2018), which we empirically find to reduce signal magnitude by a factor of 0.5, which is simply corrected by multiplying by 2. This was determined using a similar procedure to that used to find γ values for a given nonlinearity, as demonstrated in Appendix D. We validate this empirically by training NF-Net models with unmodified S+E blocks, which do not train stably, and NF-Net models with the additional correcting factor of 2, which do train stably.

B.4 AVERAGE POOLING

In line with best practices determined by He et al. (2019), in our NF-Net models we replace the strided 1x1 convolutions with average pooling followed by 1x1 convolutions, a common alternative also employed in Zagoruyko & Komodakis (2016). We found that average pooling with a kernel of size $k \times k$ tended to attenuate the signal by a factor of k, but that it was not necessary to apply any correction due to this. While this will result in mis-estimation of β values at initialization, it does not harm training (and average pooling in fact improved results over strided 1x1 convolutions in every case we tried), so we simply include this operation as-is.

APPENDIX C MODEL DETAILS

We develop the NF-Nets architecture starting with a RegNetY-400MF architecture (Radosavovic et al. (2020)) a low-latency RegNet variant which also uses Squeeze+Excite blocks (Hu et al., 2018)) and uses grouped convolutions with a group width of 8. Following EfficientNets, we first add an additional expansion convolution after the final residual block, expanding to $1024w$ channels, where w is a model width multiplier hyperparameter. We find this to be very important for performance: if the classifier layer does not have access to a large enough feature basis, it will tend to underfit (as measured by higher training losses) and underperform. We also experimented with adding an additional linear expansion layer after the global average pooling, but found this not to provide the same benefit.

Next, we replace the strided 1x1 convolutions in transition layers with average pooling followed by 1x1 convolutions (following He et al. (2019)), which we also find to improve performance. We switch from ReLU activations to SiLU activations (Elfwing et al., 2018; Hendrycks & Gimpel, 2016; Ramachandran et al., 2017). We find that SiLU's benefits are only realized when used in conjunction with EMA (the model averaging we use, explained below), as in EfficientNets. The performance of the underlying weights does not seem to be affected by the difference in nonlinearities, so the improvement appears to come from SiLU apparently being more amenable to averaging.

We then tune the choice of width w and bottleneck ratio g by sweeping them on the 0.4B FLOP model. Contrary to Radosavovic et al. (2020) which found that inverted bottlenecks (Sandler et al., 2018) were not performant, we find that inverted bottlenecks strongly outperformed their compressive bottleneck counterparts, and select $w = 0.75$ and $g = 2.25$. Following EfficientNets (Tan & Le, 2019), the very first residual block in a network uses $g = 1$, a FLOPs-reducing strategy that does not appear to harmfully impact performance.

We also modify the S+E layers to be wider by making their hidden channel width a function of the block's expanded width, rather than the block's input width (which is smaller in an inverted bottleneck). This results in our models having higher parameter counts than their equivalent FLOP target EfficientNets, but has minimal effect on FLOPS, while improving performance. While both FLOPs and parameter count play a part in the latency of a deployed model, (the quantity which is often most relevant in practice) neither are fully predictive of latency (Xiong et al., 2020). We choose to focus on the FLOPs target instead of parameter count, as one can typically obtain large improvements in accuracy at a given parameter count by, for example, increasing the resolution of the input image, which will dramatically increase the FLOPs.

With our baseline model in hand, we apply the EfficientNet compound scaling (increasing width, depth, and input image resolution) to obtain a family of models at approximately the same FLOPs targets as each EfficientNet variant. We directly use the EfficientNet width and depth multipliers for models B0 through B5, and tune the test image resolution to attain similar FLOP counts (although our models tend to have slightly higher FLOP budgets). Again contrary to Radosavovic et al. (2020), which scales models almost entirely by increasing width and group width, we find that the EfficientNet compound scaling works effectively as originally reported, particularly with respect to image size. Improvements might be made by applying further architecture search, such as tuning the w and g values for each variant separately, or by choosing the group width separately for each variant.

Following Touvron et al. (2019), we train on images of slightly lower resolution than we test on, primarily to reduce the resource costs of training. We do not employ the fine-tuning procedure of Touvron et al. (2019). The exact train and test image sizes we use are visible in our model code in Appendix D.

We train using SGD with Nesterov Momentum, using a batch size of 1024 for 360 epochs, which is chosen to be in line with EfficientNet's schedule of 360 epoch training at batch size 4096. We employ a 5 epoch warmup to a learning rate of 0.4 (Goyal et al., 2017), and cosine annealing to 0 over the remaining epochs (Loshchilov & Hutter, 2016). As with EfficientNets, we also take an exponential moving average of the weights (Polyak, 1964), using a decay of 0.99999 which employs a warmup schedule such that at iteration i, the decay is $decay = min(i, \frac{1+i}{10+i})$. We choose a larger decay than the EfficientNets value of 0.9999, as EfficientNets also take an EMA of the running average statistics of the BatchNorm layers, resulting in a longer horizon for the averaged model.

As with EfficientNets, we find that some of our models attain their best performance before the end of training, but unlike EfficientNets we do not employ early stopping, instead simply reporting performance at the end of training. The source of this phenomenon is that as some models (particularly larger models) reach the end of their decay schedule, the rate of change of their weights slows, ultimately resulting in the averaged weights converging back towards the underlying (less performant weights). Future work in this area might consider examining the interaction between averaging and learning rate.

Following EfficientNets, we also use stochastic depth (modified to remove the rescaling by the keep rate, so as to better preserve signal) with a drop rate that scales from 0 to 0.1 with depth (reduced from the EfficientNets value of 0.2). We swept this value and found the model to not be especially sensitive to it as long as it was not chosen beyond 0.25. We apply Dropout (Srivastava et al., 2014) to the final pooled activations, using the same Dropout rates as EfficientNets for each variant. We also use label smoothing (Szegedy et al., 2016a) of 0.1, and weight decay of 5e-5.

APPENDIX D MODEL CODE

We here provide reference code using Numpy (Harris et al., 2020), JAX (Bradbury et al., 2018), and PyTorch (Paszke et al., 2019).

D.1 NUMERICAL APPROXIMATIONS OF NONLINEARITY-SPECIFIC GAINS

It is often faster to determine the nonlinearity-specific constants γ empirically, especially when the chosen activation functions are complex or difficult to integrate. One simple way to do this is for the SiLU function is to sample many (say, 1024) random C-dimensional vectors (of say size 256) and compute the average variance, which will allow for computing an estimate of the constant.

```
import jax
import jax.numpy as jnp
key = jax.random.PRNGKey(2) # Arbitrary key
# Produce a large batch of random noise vectors
x = jax.random.normal(key, (1024, 256))
y = jax.nn.silu(x)
# Take the average variance of many random batches
gamma = jnp.mean(jnp.var(y, axis=1)) ** -0.5
```

D.2 REFERENCE PYTORCH IMPLEMENTATION OF NFNETS

```
class ScaledWSConv2d(nn.Conv2d):
  """2D Conv layer with Scaled Weight Standardization."""
  def __init__(self, in_channels, out_channels, kernel_size,
               stride=1, padding=0,
               dilation=1, groups=1, bias=True, gain=True,
               eps=1e-4):
    nn.Conv2d.__init__(self, in_channels, out_channels,
                       kernel_size, stride,
                       padding, dilation,
                       groups, bias)
    if gain:
      self.gain = nn.Parameter(
                    torch.ones(self.out_channels, 1, 1, 1))
    else:
      self.gain = None
    # Epsilon, a small constant to avoid dividing by zero.
    self.eps = eps

  def get_weight(self):
    # Get Scaled WS weight OIHW;
    fan_in = np.prod(self.weight.shape[1:])
    mean = torch.mean(self.weight, axis=[1, 2, 3],
                      keepdims=True)
    var = torch.var(self.weight, axis=[1, 2, 3],
                    keepdims=True)
    weight = (self.weight - mean) / (var + self.eps) ** 0.5
    if self.gain is not None:
      weight = weight * self.gain
    return weight

  def forward(self, x):
    return F.conv2d(x, self.get_weight(), self.bias,
                    self.stride, self.padding,
                    self.dilation, self.groups)

class SqueezeExcite(nn.Module):
```

```
  """Simple Squeeze+Excite layers."""

  def __init__(self, in_channels, width, activation):
    super().__init__()
    self.se_conv0 = nn.Conv2d(in_channels, width,
                              kernel_size=1, bias=True)
    self.se_conv1 = nn.Conv2d(width, in_channels,
                              kernel_size=1, bias=True)
    self.activation = activation

  def forward(self, x):
    # Mean pool for NCHW tensors
    h = torch.mean(x, axis=[2, 3], keepdims=True)
    # Apply two linear layers with activation in between
    h = self.se_conv1(self.activation(self.se_conv0(h)))
    # Rescale the sigmoid output and return
    return (torch.sigmoid(h) * 2) * x


class NFBlock(nn.Module):
  """NF-Net block."""

  def __init__(self, in_channels, out_channels, stride=1,
               activation=F.relu, which_conv=ScaledWSConv2d,
               beta=1.0, alpha=1.0,
               expansion=2.25, keep_rate=None,
               se_ratio=0.5, group_size=8):
    super().__init__()
    self.in_ch, self.out_ch = in_channels, out_channels
    self.activation = activation
    self.beta, self.alpha = beta, alpha
    width = int(in_channels * expansion)
    if group_size is None:
      groups = 1
    else:
      groups = width // group_size
      # Round width up if you pick a bad group size
      width = int(group_size * groups)
    self.stride = stride
    self.width = width
    self.conv1x1a = which_conv(in_channels, width,
                               kernel_size=1, padding=0)
    self.conv3x3 = which_conv(width, width,
                              kernel_size=3, stride=stride,
                              padding=1, groups=groups)
    self.conv1x1b = which_conv(width, out_channels,
                               kernel_size=1, padding=0)
    if stride > 1 or in_channels != out_channels:
      self.conv_shortcut = which_conv(in_channels, out_channels,
                                      kernel_size=1, stride=1,
                                      padding=0)
    else:
      self.conv_shortcut = None
    # Hidden size of the S+E MLP
    se_width = max(1, int(width * se_ratio))
    self.se = SqueezeExcite(width, se_width, self.activation)
    self.skipinit_gain = nn.Parameter(torch.zeros(()))

  def forward(self, x):
```

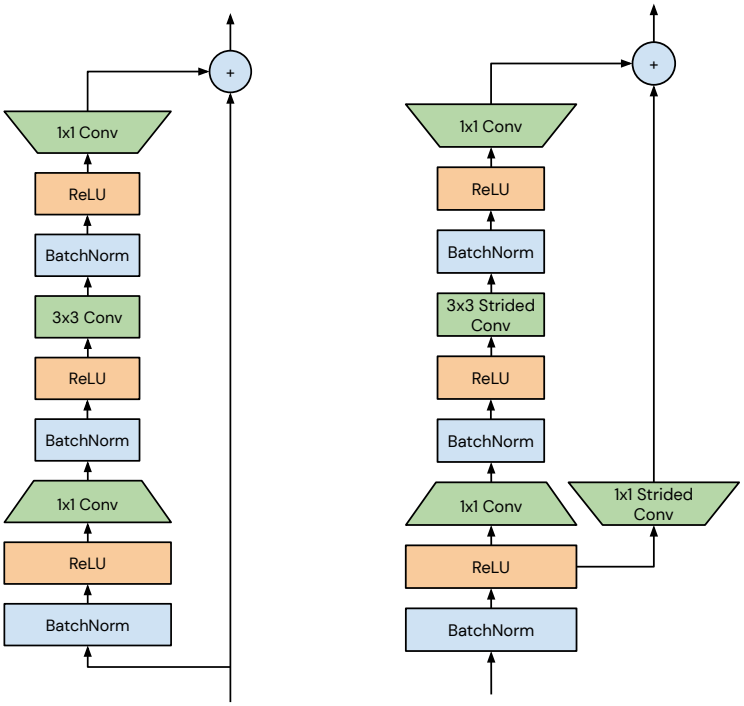
```
    out = self.activation(x) / self.beta
    if self.conv_shortcut is not None:
      shortcut = self.conv_shortcut(F.avg_pool2d(out, 2))
    else:
      shortcut = x
    out = self.conv1x1a(out) # Initial bottleneck conv
    out = self.conv3x3(self.activation(out)) # Spatial conv
    out = self.se(out) # Apply squeeze + excite to middle block.
    out = self.conv1x1b(self.activation(out))
    return out * self.skipinit_gain * self.alpha + shortcut

# Nonlinearities. Note that we bake the constant into the
# nonlinearites rather than the WS layers.
nonlinearities = {
  'silu': lambda x: x * torch.sigmoid(x) / .5595,
  'relu': lambda x: F.relu(x) / (0.5 * (1 - 1 / np.pi)) ** 0.5,
  'identity': lambda x: x}

# Block base widths and depths for each variant
params = {'NFNet-B0': {'width': [48, 104, 208, 440],
                       'depth': [1, 3, 6, 6],
                       'train_imsize': 192, 'test_imsize': 224,},
          'NFNet-B1': {'width': [48, 104, 208, 440],
                       'depth': [2, 4, 7, 7],
                       'train_imsize': 240, 'test_imsize': 256,},
          'NFNet-B2': {'width': [56, 112, 232, 488],
                       'depth': [2, 4, 8, 8],
                       'train_imsize': 240, 'test_imsize': 272,},
          'NFNet-B3': {'width': [56, 128, 248, 528],
                       'depth': [2, 5, 9, 9],
                       'train_imsize': 288, 'test_imsize': 320,},
          'NFNet-B4': {'width': [64, 144, 288, 616],
                       'depth': [2, 6, 11, 11],
                       'train_imsize': 320, 'test_imsize': 384,},
          'NFNet-B5': {'width': [80, 168, 336, 704],
                       'depth': [3, 7, 14, 14],
                       'train_imsize': 384, 'test_imsize': 456,},
          'NFNet-B6': {'width': [88, 184, 376, 792],
                       'depth': [3, 8, 16, 16],
                       'train_imsize': 448, 'test_imsize': 528,},
          'NFNet-B7': {'width': [96, 208, 416, 880],
                       'depth': [4, 10, 19, 19],
                       'train_imsize': 512, 'test_imsize': 600,},
          'NFNet-B8': {'width': [104, 232, 456, 968],
                       'depth': [4, 11, 22, 22],
                       'train_imsize': 600, 'test_imsize': 672,},
}
def count_params(module):
  sum([item.numel() for item in module.parameters()])

class NFNet(nn.Module):
  """Normalizer-Free Networks."""

  def __init__(self, variant='NfNet-B0',
               num_classes=1000, width=0.75, expansion=2.25,
               se_ratio=0.5, group_size=8, alpha=0.2,
               activation='silu', drop_rate=0.0,
               stochdepth_rate=0.0,):
    super().__init__()
```

```
    self.variant = variant
    self.width = width
    self.expansion = expansion
    self.num_classes = num_classes
    self.se_ratio = se_ratio
    self.group_size = group_size
    self.alpha = alpha
    self.activation = nonlinearities.get(activation)
    self.drop_rate = drop_rate
    self.stochdepth_rate = stochdepth_rate
    self.which_conv = functools.partial(ScaledWSConv2d,
                                        gain=True,
                                        bias=True)
    # Get width and depth pattern
    self.width_pattern = [int(val * self.width)
                          for val in params[variant]['width']]
    self.depth_pattern = params[self.variant]['depth']
    # Stem conv
    in_channels = int(self.width_pattern[0])
    self.initial_conv = self.which_conv(3, in_channels,
                                        kernel_size=3,
                                        stride=2,
                                        padding=1)

    # Body
    blocks = []
    expected_var = 1.0
    index = 0
    for block_width, stage_depth in zip(self.width_pattern,
                                        self.depth_pattern):
      for block_index in range(stage_depth):
        # Following EffNets, do not expand first block
        expand_ratio = expansion if index > 0 else 1
        beta = 1./ expected_var ** 0.5
        blocks += [NFBlock(in_channels, block_width,
                           stride=2 if block_index==0 else 1,
                           activation=self.activation,
                           which_conv=self.which_conv,
                           beta=beta, alpha=self.alpha,
                           expansion=expand_ratio,
                           se_ratio=self.se_ratio,
                           group_size=self.group_size)]
        # Keep track of output channel count
        in_channels = block_width
        # Reset expected var at a transition block
        if block_index == 0:
          expected_var = 1.
        # Even if reset occurs, increment expected variance
        expected_var += self.alpha ** 2
        index += 1

    self.blocks = nn.Sequential(*blocks)
    # Final convolution, following EffNets
    self.final_conv = self.which_conv(in_channels,
                                      int(1024 * self.width),
                                      kernel_size=1,
                                      padding=0)
    in_channels = int(1024 * self.width)
    if self.drop_rate > 0.0:
```

```
      self.dropout = nn.Dropout(drop_rate)
    # Classifier layer. Initialize this layer's weight with zeros!
    self.fc = nn.Linear(in_channels,
                        self.num_classes,
                        bias=True)
    torch.nn.init.zeros_(self.fc.weight)

  def forward(self, x):
    """Return the logits without any [log-]softmax."""
    # Stem
    out = self.initial_conv(x)
    # Blocks
    out = self.blocks(out)
    # Final activation + conv
    out = self.activation(self.final_conv(out))
    # Global average pooling
    pool = torch.mean(out, [2, 3])
    if self.drop_rate > 0.0 and self.training:
      pool = self.dropout(pool)
    # Return logits
    return self.fc(pool)
```

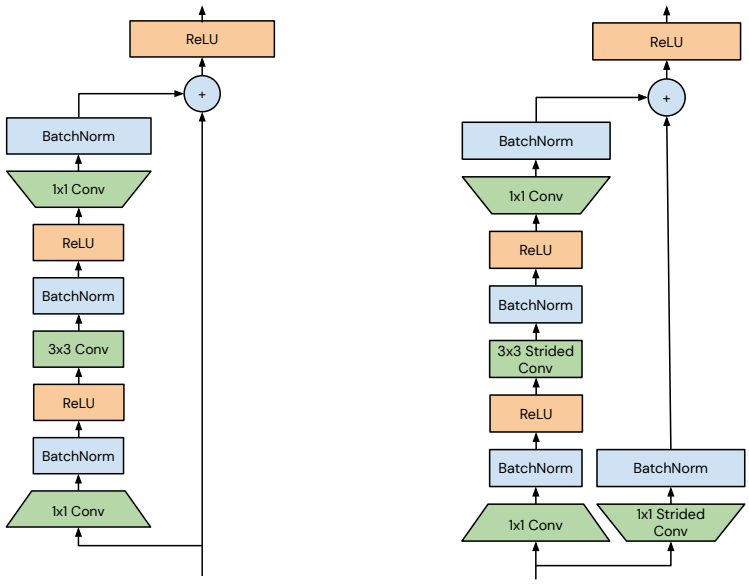
APPENDIX E OVERVIEW OF EXISTING BLOCKS

This appendix contains an overview of several different types of residual blocks.



(a) Pre-Activation ResNet Block (b) Pre-Activation ResNet Transition Block

Figure 4: Residual Blocks for pre-activation ResNets (He et al., 2016a). Note that some variants swap the order of the nonlinearity and the BatchNorm, resulting in signal propagation which is more similar to that of our normalizer-free networks.



(a) Post-Activation ResNet Block (b) Post-Activation ResNet Transition Block

Figure 5: Residual Blocks for post-activation (original) ResNets (He et al., 2016b).

APPENDIX F ADDITIONAL SPPS

In this appendix, we include additional Signal Propagation Plots. For reference, given an $NHWC$ tensor, we compute the measured properties using the equivalent of the following Numpy (Harris et al., 2020) snippets:

- Average Channel Mean Squared:
  `np.mean(np.mean(y, axis=[0, 1, 2]) ** 2)`
- Average Channel Variance:
  `np.mean(np.var(y, axis=[0, 1, 2]))`
- Residual Average Channel Variance:
  `np.mean(np.var(f(x), axis=[0, 1, 2]))`

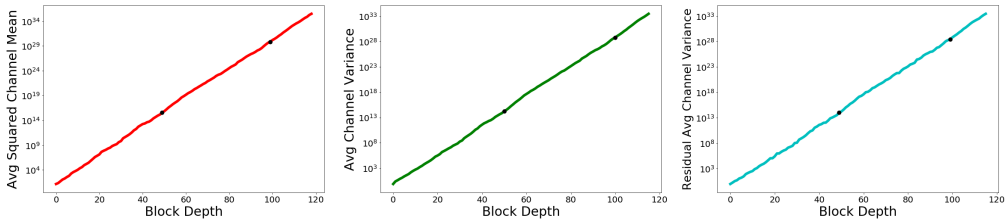


Figure 6: Signal Propagation Plot for a ResNetV2-600 with ReLU and He initialization, without any normalization, on a semilog scale. The scales of all three properties grow logarithmically due to signal explosion.

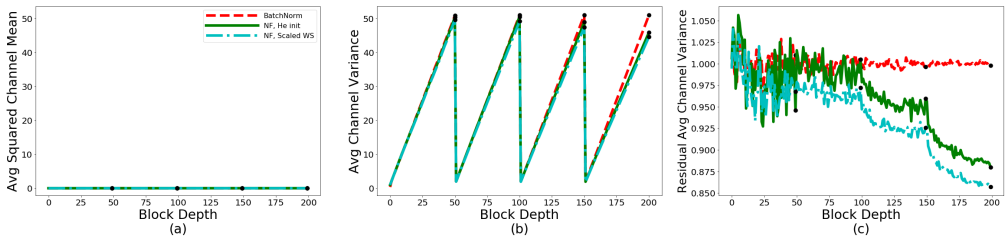


Figure 7: Signal Propagation Plot for 600-layer ResNetV2s with linear activations, comparing BatchNorm against with normalizer-free scaling. Note that the max-pooling operation in the ResNet stem has here been removed so that the inputs to the first blocks are centered.

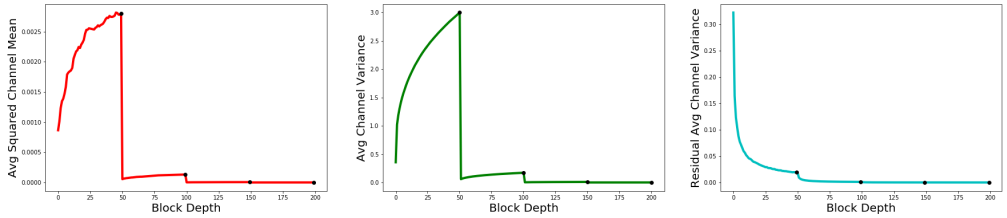


Figure 8: Signal Propagation Plot for a Normalizer-Free ResNetV2-600 with ReLU and Scaled WS, using $\gamma = \sqrt{2}$, the gain for ReLU from (He et al., 2015). As this gain (derived from $\sqrt{\frac{1}{\mathbb{E}[g(x)^2]}}$) is lower than the correct gain ($\sqrt{\frac{1}{Var(g(x))}}$), signals attenuate progressively in the first stage, then are further downscaled at each transition which uses a β value that assumes a higher incoming scale.

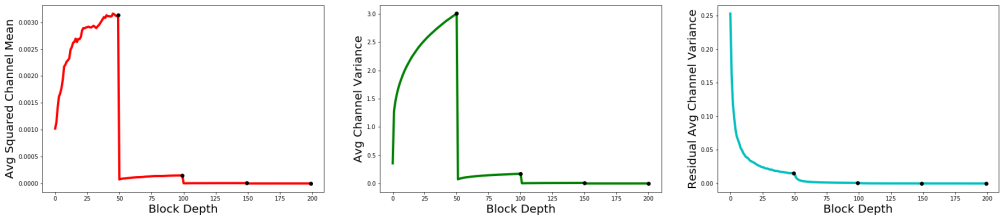


Figure 9: Signal Propagation Plot for a Normalizer-Free ResNetV2-600 with ReLU, Scaled WS with correctly chosen gains, and unmodified Squeeze-and-Excite Blocks. Similar to understimating γ values, unmodified S+E blocks will attenuate the signal.

APPENDIX G NEGATIVE RESULTS

G.1 FORWARD MODE VS DECOUPLED WS

Parameterization methods like Weight Standardization (Qiao et al., 2019), Weight Normalization (Salimans & Kingma, 2016), and Spectral Normalization (Miyato et al., 2018) are typically proposed as "foward mode" modifications applied to parameters during the forward pass of a network. This has two consequences: first, this means that the gradients with respect to the underlying parameters are influenced by the parameterization, and that the weights which are optimized may differ substantially from the weights which are actually plugged into the network.

One alternative approach is to implement "decoupled" variants of these parameterizers, by applying them as a projection step in the optimizer. For example, "Decoupled Weight Standardization" can be implemented atop any gradient based optimizer by replacing W with the normalized $\hat{W}$ after the update step. Most papers proposing parameterizations (including the above) argue that the parameterization's gradient influence is helpful for learning, but this is typically argued with respect to simply ignoring the parameterization during the backward pass, rather than with respect to a strategy such as this.

Using a Forward-Mode parameterization may result in interesting interactions with moving averages or weight decay. For example, with WS, if one takes a moving average of the underlying weights, then applies the WS parameterization to the averaged weights, this will produce different results than if one took the EMA of the Weight-Standardized parameters. Weight decay will have a similar phenomenon: if one is weight decaying a parameter which is actually a proxy for a weight-standardized parameter, how does this change the behavior of the regularization?

We experimented with Decoupled WS and found that it reduced sensitivity to weight decay (presumably because of the strength of the projection step) and often improved the accuracy of the EMA weights early in training, but ultimately led to worse performance than using the originally proposed "forward-mode" formulation. We emphasize that our experiments in this regime were only cursory, and suggest that future work might seek to analyze these interactions in more depth.

We also tried applying Scaled WS as a regularizer ("Soft WS") by penalizing the mean squared error between the parameter W and its Scaled WS parameterization, $\hat{W}$. We implemented this as a direct addition to the parameters following Loshchilov & Hutter (2017) rather than as a differentiated loss, with a scale hyperparameter controlling the strength of the regularization. We found that this scale could not be meaningfully decreased from its maximal value without drastic training instability, indicating that relaxing the WS constraint is better done through other means, such as the affine gains and biases we employ.

G.2 MISCELLANEOUS

- For SPPs, we initially explored plotting activation mean (`np.mean(h)`) instead of the average squared channel mean, but found that this was less informative.
- We also initially explored plotting the average pixel norm: the Frobenius norm of each pixel (reduced across the C axis) then averaged across the NHW axis, `np.mean(np.linalg.norm(h, axis=-1))`. We found that this value did not add any information not already contained in the channel or residual variance measures, and was harder to interpret due to it varying with the channel count.
- We explored NF-ResNet variants which maintained *constant* signal variance, rather than mimicking Batch-Normalized ResNets with signal growth + resets. The first of two key components in this approach was making use of "rescaled sum junctions," where the sum junction in a residual block was rewritten to downscale the shortcut path as $y = \frac{\alpha * f(x) + x}{\alpha^2}$, which is approximately norm-preserving if $f(x)$ is orthogonal to x (which we observed to generally hold in practice). Instead of scaled WS, this variant employed SeLU (Klambauer et al., 2017) activations, which we found to work as-advertised in encouraging centering and good scaling.

  While these networks could be made to train stably, we found tuning them to be difficult and were not able to easily recover the performance of BN-ResNets as we were with the approach ultimately presented in this paper.