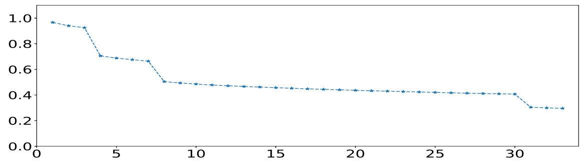
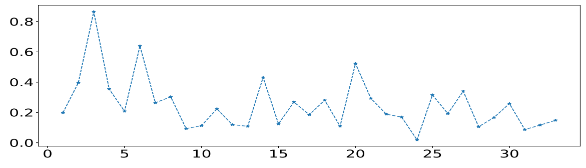
We thank all reviewers for their detailed suggestions and questions. Here are the responses to reviewers' questions.
[R1] *Assumption made in the analysis part should be more explicit. Discuss how realistic they are. Particularly, the analysis section assumes that two coordinates in $\boldsymbol{x}_k$ are uncorrelated.* **[R2]** *Numerical evidence for some of the claims.* Thanks! We will write assumptions more explicitly in the final version. *Two coordinates in $\boldsymbol{x}_k$ being uncorrelated* is a widely used assumption [11]. We guess the reviewer may want to verify the assumption that any coordinate in $\boldsymbol{x}_k$ and any coordinate in $\mathcal{F}(\boldsymbol{x}_k)$ are uncorrelated. If the two are uncorrelated, we have $\mathrm{Var}[\boldsymbol{x}_k] \approx \mathrm{Var}[\alpha_k \boldsymbol{x}_{x-1}] + \mathrm{Var}[\beta_k \mathcal{F}(\boldsymbol{x}_{k-1})]$ from Equation 2 in the submission paper. We define the relative error as:

$$\eta_k = \left| \frac{\mathrm{Var}[\alpha_k \boldsymbol{x}_{x-1}] + \mathrm{Var}[\beta_k \mathcal{F}(\boldsymbol{x}_{k-1})] - \mathrm{Var}[\boldsymbol{x}_k]}{\mathrm{Var}[\boldsymbol{x}_k]} \right| \times 100\% \quad (1)$$

Fig (a) shows the output variance $\mathrm{Var}[\boldsymbol{x}_k]$ and (b) shows the error η_k using ImageNet data and Rescale101 model. The x



(a) y coordinate: residual block output variance



(b) y coordinate: relative error (%)

coordinate is the block index. The sudden drops in (a) are caused by downsample layers. Our assumptions are realistic.

[R2] *Compare against batchnorm + mixup training for 200 epochs; does the method continue to improve (with more epochs)?* By training with mixup for 200 epochs, our baseline model (Table 3, 76.6%) achieve a **77.5%** top-1 accuracy. The improvement (0.9%) is less than mixup (1.5%) since our method is not a regularization technique. We also compare with "Bags of Tricks"[14] (line 253). The result shows our method can continue to improve with most training tricks.
[R2] *Schematically show the per-layer learned scalar multipliers:* Result from a ReScale50 checkpoint (16 scalars from 16 residual blocks, from shallow ones to deep ones): 11.073, 10.339, 10.310, 14.467, 10.082, 13.121, 14.276, 16.991, 12.647, 15.192, 15.548, 17.261, 18.033, 27.445, 23.516, 27.258. More visualization results will be added.
[R2] *Subtract the first batch means vs. subtract the mean from a larger set of data*: we tried to subtract the mean from more data, but find a batch size of 128 is all ready good enough: almost no neurons dies in the beginning of training. If the model is too big to feed a batch of 128 data into GPUs, we can do this on CPU (just do one forward computation).
[R2] *Benefit of including a trainable scalar multiplier?* The benefits are two fold: 1) The convolution weights shrink in the training due to weight decay (paper Fig 2 (a)), thus the large scalars can compensate for the shrink to keep the output variance. 2) The learned scalars up-weights deeper blocks which have more parameters and higher-level features.

[R3] *Inclusion of scalar multipliers contradicts the variance normalization induced by the alpha and beta in 4.1 and 4.2; having a >1 scalar term should still result in the "explosion" of the variance* Thanks! Scalar multipliers are initialized as one, thus are not a problem in the beginning of training. During training, the multipliers are updated to larger values, meanwhile the weights also shrink due to weight decay (paper Fig 2(a)). Without scalar multipliers, the output variance of each residual blocks would shrink as the weights shrink. Large multipliers can compensate for the shrink. Experiments show that the variance of the last block output is always in a proper range in the final model.
[R3] *Variances of reported results*: Thanks! The method stability is indeed vital. We are repeating the reported results with more trials. So far we have two results that may be of most interest to reviewers (mean$\pm$std from 6 runs): RescaleNet50: 76.57%$\pm$0.084%, RescaleNet101: 77.55%$\pm$0.110%. More results will be added to the final version.

[R4] *Initialization of convolution depending on the depth was not been explained in the paper.* Thanks! We mentioned this in line155-161. In Sec 4.1, we derived a $(k+L)^{-1/2}$ scaling coefficient for the k^{th} block (L is the number of blocks). In line143-154 we mentioned this may not be good since the backward gradient would be scaled differently by the layer-wise scaling. A fixed value scaling coefficient, say $L^{-1/2}$, is better. However, the output variance would change if we only replace the scaling coefficient. Since the new coefficient is $L^{-1/2}(k+L)^{1/2}$ times larger, the output variance of the residual branch should be $L^{1/2}(k+L)^{-1/2}$ times smaller. Note that there are three convolutions in Bottleneck block, each convolution should be initialized with a $L^{1/6}(k+L)^{-1/6}$ times smaller standard deviation (In our codes `block_idx` begins at L, so `block_idx+1` is $L+k$). We will write it more explicitly in the final version.
[R4] *Theory derivation is quite similar with previous works:* Thanks! Two differences from previous works' theories: 1) Fixup and Skip-Init are very similar to "zero-gamma" trick: their work use zeros to initialize the last layer in residual branches, mimicking network that has less number of layers and are easier to train. Our method does not benefit from zero initialization and all layers have non-zero outputs, which is more meaningful. It shows that deep networks can be trained well without "zero-gamma". 2) Previous works mainly focus on how to make deep non-normalized networks trainable. We further move on to how to train non-normalized models well (get the same performance as the corresponding normalized models). For example, our method can also apply to non-residual network VGG with a very competitive performance. Compared to previous works, our experiments are also more extensive.