ORIGINAL ARTICLE

A scaling calculus for the design and initialization of ReLU networks

Aaron Defazio[1] (ID) · **Léon Bottou**[2] (ID)

Abstract

We propose a system for calculating a "scaling constant" for layers and weights of neural networks. We relate this scaling constant to two important quantities that relate to the optimizability of neural networks, and argue that a network that is "preconditioned" via scaling, in the sense that all weights have the same scaling constant, will be easier to train. This scaling calculus results in a number of consequences, among them the fact that the geometric mean of the fan-in and fan-out, rather than the fan-in, fan-out, or arithmetic mean, should be used for the initialization of the variance of weights in a neural network. Our system allows for the off-line design & engineering of ReLU (Rectified Linear Unit) neural networks, potentially replacing blind experimentation. We verify the effectiveness of our approach on a set of benchmark problems.

Keywords ReLU · Neural network initialization · Preconditioning · Artificial neural networks

1 Introduction

The design of neural networks is often considered a black-art, driven by trial and error rather than foundational principles. This is exemplified by the success of recent architecture random-search techniques [22, 37], which take the extreme of applying no human guidance at all. Although as a field we are far from fully understanding the nature of learning and generalization in neural networks, this does not mean that we should proceed blindly.

In this work, we define a scaling quantity γ_l for each layer l that approximates two quantities of interest when considering the optimization of a neural network: The ratio of the gradient to the weights, and the average squared singular value of the corresponding diagonal block of the Hessian for layer l. This quantity is easy to compute from the (non-central) second moments of the forward-propagated values and the (non-central) second moments of the backward-propagated gradients. We argue that networks that have constant γ_l are better conditioned than those that

do not, and we analyze how common layer types affect this quantity. We call networks that obey this rule *preconditioned* neural networks.

As an example of some of the possible applications of our theory, we:

- Propose a *principled* weight initialization scheme that can often provide an improvement over existing schemes;
- Show which common layer types automatically result in well-conditioned networks;
- Show how to improve the conditioning of common structures such as bottlenecked residual blocks by the addition of fixed scaling constants to the network.

2 Notation

Consider a neural network mapping x_0 to x_L made up of L layers. These layers may be individual operations or blocks of operations. During training, a loss function is computed for each minibatch of data, and the gradient of the loss is back-propagated to each layer l and weight of the network. We prefix each quantity with Δ to represent the back-propagated gradient of that quantity. We assume a batch-size of 1 in our calculations, although all conclusions hold using mini-batches as well.

Each layer's input activations are represented by a tensor $x_l : n_l \times \rho_l \times \rho_l$ made up of n_l channels, and spatial

✉ Aaron Defazio
adefazio@fb.com

[1] Facebook AI Research New York, 770 Broadway, New York, NY, USA

[2] Facebook AI Research New York, 770 Broadway, New York, NY, USA

dimensions $\rho_l \times \rho_l$, assumed to be square for simplicity (results can be adapted to the rectangular case by using $h_l w_l$ in place of ρ_l everywhere).

3 A model of ReLU network dynamics

Our scaling calculus requires the use of simple approximations of the dynamics of neural networks, in the same way that simplifications are used in physics to make approximate calculations, such as the assumption of zero-friction or ideal gasses. These assumptions constitute a model of the behavior of neural networks that allows for easy calculation of quantities of interest, while still being representative enough of the real dynamics.

To this end, we will focus in this work on the behavior of networks at initialization. Furthermore, we will make strong assumptions on the statistics of forward and backward quantities in the network. These assumptions include:

1. The input to layer l, denoted x_l, is a random tensor assumed to contain i.i.d entries. We represent the element-wise uncentered 2nd moment by $E[x_l^2]$.
2. The back-propagated gradient of x_l is Δx_l and is assumed to be uncorrelated with x_l and iid. We represent the uncentered 2nd-moment of Δx_l by $E[\Delta x_l^2]$.
3. All weights in the network are initialized i.i.d from a centered, symmetric distribution.
4. All bias terms are initialized as zero.

Our calculations rely heavily on the uncentered second moments rather than the variance of weights and gradients. This is a consequence of the behavior of the ReLU activation, which zeros out entries. The effect of this zeroing operation is simple when considering uncentered second moments under a symmetric input distribution, as half of the entries will be zeroed, resulting in a halving of the uncentered second moment. In contrast, expressing the same operation in terms of variance is complicated by the fact that the mean after application of the ReLU is distribution-dependent. We will refer to the uncentered second moment just as the "second moment" henceforth.

4 Activation and layer scaling factors

The key quantity in our calculus is the activation scaling factor ς_l, of the input activations for a layer l, which we define as:

$$\varsigma_l = n_l \rho_l^2 E[\Delta x_l^2] E[x_l^2]. \tag{1}$$

This quantity arises due to its utility in computing other quantities of interest in the network, such as the scaling factors for the weights of convolutional and linear layers. In ReLU networks, many, but not all operations maintain this quantity in the sense that $\varsigma_l = \varsigma_{l+1}$ for a layer $x_{l+1} = F(x_l)$ with operation F, under the assumptions of Sect. 3. Table 1 contains a list of common operations and indicates if they maintain scaling. As an example, consider adding a simple scaling layer of the form $x_{l+1} = \sqrt{2}x_l$ which doubles the second moment during the forward pass and doubles the backward second moment during back-propagation. We can see that:

$$\varsigma_{l+1} = n_{l+1} \rho_{l+1}^2 E[\Delta x_{l+1}^2] E[x_{l+1}^2]$$
$$= n_l \rho_l^2 \frac{1}{2} E[\Delta x_l^2] \cdot 2E[x_l^2] = \varsigma_l$$

Our analysis in our work is focused on ReLU networks primarily due to the fact that ReLU nonlinearities maintain this scaling factor.

Using the activation scaling factor, we define the layer or weight scaling factor of a convolutional layer with kernel $k_l \times k_l$ as:

$$\gamma_l = \frac{\varsigma_l}{n_{l+1} n_l k_l^2 E[W_l^2]^2}. \tag{2}$$

Recall that n_l is the fan-in and n_{l+1} is the fan-out of the layer. This expression also applies to linear layers by taking $k_l = 1$. This quantity can also be defined extrinsically without reference to the weight initialization via the expression:

$$\gamma_l = n_l k_l^2 \rho_l^2 E[x_l^2]^2 \frac{E[\Delta x_{l+1}^2]}{E[x_{l+1}^2]}.$$

we establish this equivalence under the assumptions of Sect. 3 in the Appendix.

5 Motivations for scaling factors

We can motivate the utility of our scaling factor definition by comparing it to another simple quantity of interest. For each layer, consider the ratio of the second moments between the weights, and their gradients:

$$v_l \doteq \frac{E[\Delta W_l^2]}{E[W_l^2]}.$$

This ratio approximately captures the relative change that a single SGD step with unit step-size on W_l will produce. We call this quantity the weight-to-gradient ratio. When $E[\Delta W_l^2]$ is very small compared to $E[W_l^2]$, the weights will stay close to their initial values for longer than when $E[\Delta W_l^2]$ is large. In contrast, if $E[\Delta W_l^2]$ is very large compared to $E[W_l^2]$, then learning can be expected to be unstable, as the sign of the elements of W may change

Table 1 Scaling of common layers

Method	Maintains scaling	Notes
Linear layer	✔	Layer scaling requires geometric initialization
(Strided) convolution	✔	Requires stride equal to the kernel size
Skip connections	✔	Operations within residual blocks will also be scaled correctly against other residual blocks, but not against outside operations.
Average pooling	✔	Requires stride equal to the kernel size
Max pooling	✗	
Dropout	✔	
ReLU/ LeakyReLU	✔	Any positively-homogenous function with degree 1
Sigmoid	✗	
Tanhh	✗	Maintains scaling if entirely within the linear regime

rapidly between optimization steps. A network with constant v_l is also well-behaved under weight-decay, as the ratio of weight-decay second moments to gradient second moments will stay constant throughout the network, keeping the push-pull of gradients and decay constant across the network. This ratio also captures a relative notion of exploding or vanishing gradients. Rather than consider if the gradient is small or large in absolute value, we consider its relative magnitude instead.

Theorem 1 *The weight to gradient ratio v_l is equal to the scaling factor γ_l under the assumptions of Sect. 3*

5.1 Conditioning of the Hessian

The scaling factor of a layer l is also closely related to the singular values of the diagonal block of the Hessian corresponding to that layer. We derive a correspondence in this section, providing further justification for our definition of the scaling factor above. We focus on non-convolutional layers for simplicity in this section, although the result extends to the convolutional case without issue.

ReLU networks have a particularly simple structure for the Hessian for any set of activations, as the network's output is a piecewise-linear function g fed into a final layer consisting of a loss. This structure results in greatly simplified expressions for diagonal blocks of the Hessian with respect to the weights, and allows us to derive expressions involving the singular values of these blocks.

We will consider the output of the network as a composition of two functions, the current layer g, and the remainder of the network h. We write this as a function of the weights, i.e. $f(W_l) = h(g(W_l))$. The dependence on the input to the network is implicit in this notation, and the network below layer l does not need to be considered.

Let $R_l = \nabla^2_{x_{l+1}} h(x_{l+1})$ be the Hessian of h, the remainder of the network after application of layer l (For a linear layer $x_{l+1} = W_l x_l$). Let J_l be the Jacobian of y_l with respect to W_l. The Jacobian has shape $J_l : n_l^{\text{out}} \times (n_l^{\text{out}} n_l^{\text{in}})$. Given these quantities, the diagonal block of the Hessian corresponding to W_l is equal to:

$$G_l = J_l^T R_l J_l.$$

The lth diagonal block of the Generalized Gauss-Newton matrix G [23]. We discuss this decomposition further in the appendix.

Assume that the input-output Jacobian Φ of the remainder of the network above each block is initialized so that $\|\Phi\|_2^2 = O(1)$ with respect to n_{l+1}. This assumption just encodes the requirement that initialization used for the remainder of the network is sensible, so that the output of the network does not blow-up for large widths.

Theorem 2 *Under the assumptions outlined in Sect. 3, for linear layer l, the average squared singular value of G_l is equal to:*

$$n_l E[x_l^2]^2 \frac{E[\Delta x_{l+1}^2]}{E[x_{l+1}^2]} + O\left(\frac{n_l E[x_l^2]^2}{n_{l+1} E[x_{l+1}^2]}\right).$$

The Big-O term is with respect to n_l and n_{l+1}; its precise value depends on properties of the remainder of the network above the current layer.

Despite the approximations required for its derivation, the scaling factor can still be close to the actual average squared singular value. We computed the ratio of the scaling factor (Eq. 2) to the actual expectation $E[(G_l r)^2]$ for a strided (rather than max-pooled, see Table 1) LeNet

model, where we use random input data and a random loss (i.e. for outputs y we use $y^T R y$ for an i.i.d normal matrix R), with batch-size 1024, and 32×32 input images. The results are shown in Fig. 1 for 100 sampled setups; there is generally good agreement with the theoretical expectation.

6 Initialization of ReLU networks

An immediate consequence of our definition of the scaling factor is a rule for the initialization of ReLU networks. Consider a network where the activation scaling factor is constant through-out. Then any two layers l and r will have the same weight scaling factor if $\gamma_l = \gamma_r$, which holds immediately when each layer is initialized with:

$$E[W_l^2] = \frac{c}{k_l \sqrt{n_l n_{l+1}}}, \qquad (3)$$

for some fixed constant c independent of the layer. Initialization using the geometric-mean of the fan-in and fan-out ensures a constant layer scaling factor throughout the network, aiding optimization. Notice that the dependence on the kernel size is also unusual, rather than k_l^2, we normalize by k_l.

6.1 Other initialization schemes

The most common approaches are the Kaiming [11] (sometimes called He) and Xavier [6] (sometimes called Glorot) initializations. The Kaiming technique for ReLU networks is one of two approaches:

$$\begin{aligned} \text{(fan-in)} \quad & \text{Var}[W_l] = \frac{2}{n_l k_l^2} \text{ or} \\ \text{(fan-out)} \quad & \text{Var}[W_l] = \frac{2}{n_{l+1} k_l^2} \end{aligned} \qquad (4)$$

For the feed-forward network above, assuming random activations, the forward-activation variance will remain constant in expectation throughout the network if fan-in initialization of weights [21] is used, whereas the fan-out variant maintains a constant variance of the back-propagated signal. The constant factor 2 corrects for the variance-reducing effect of the ReLU activation. Although popularized by [11], similar scaling was in use in early neural network models that used tanh activation functions [1].

These two principles are clearly in conflict; unless $n_l = n_{l+1}$, either the forward variance or backward variance will become non-constant. No *prima facie* reason for preferring one initialization over the other is provided. Unfortunately, there is some confusion in the literature as many works reference using Kaiming initialization without specifying if the fan-in or fan-out variant is used.

The Xavier initialization [6] is the closest to our proposed approach. They balance these conflicting objectives using the arithmetic mean:

$$\text{Var}[W_l] = \frac{2}{\frac{1}{2}(n_l + n_{l+1}) k_l^2}, \qquad (5)$$

to "... approximately satisfy our objectives of maintaining activation variances and back-propagated gradients variance as one moves up or down the network". This approach to balancing is essentially heuristic, in contrast to the geometric mean approach that our theory directly guides us to.

Figure 2 shows heat maps of the average singular values for each block of the Hessian of a LeNet model under the initializations considered. The use of geometric initialization results in an equally weighted diagonal, in contrast to the other initializations considered.

6.2 Practical application

The dependence of the geometric initialization on kernel size rather than its square will result in a large increase in forward second moments if c is not carefully chosen. We recommend setting $c = 2/k$, where k is the typical kernel size in the network. Any other layer in the network with kernel size differing from this default should be preceded by a fixed scaling factor $x_{l+1} = \alpha x_l$, that corrects for this. For instance, if the typical kernel size is 1, then a 3x3
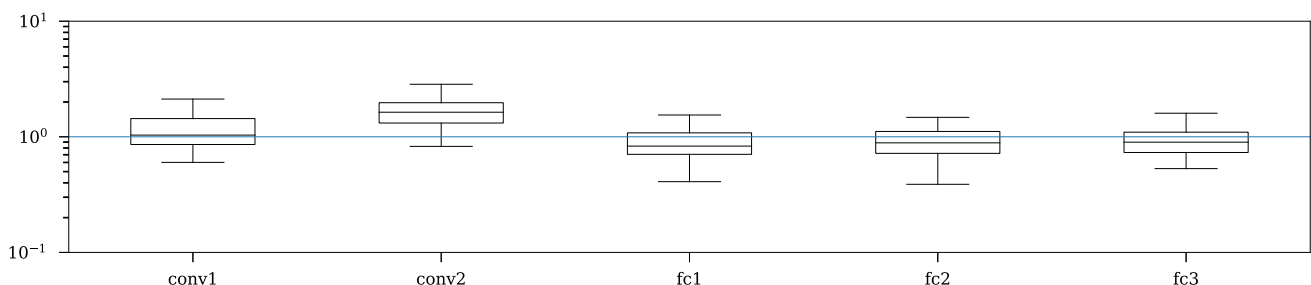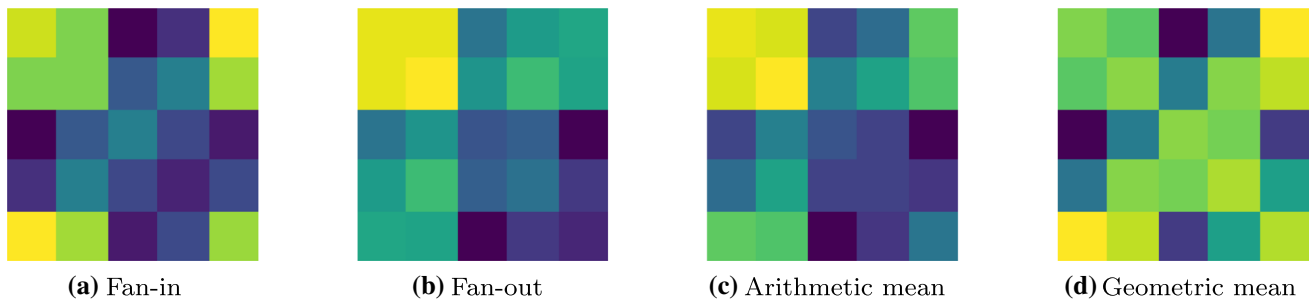

Fig. 1 Distributions of the ratio of theoretical scaling to actual for a strided LeNet network. The ratios are close to the ideal value of 1, indicating good theoretical and practical agreement

(a) Fan-in   **(b)** Fan-out   **(c)** Arithmetic mean   **(d)** Geometric mean

Fig. 2 Average singular value heat maps for the strided LeNet model, where each square represents a block of the Hessian, with blocking at the level of weight matrices (biases omitted). Using geometric initialization maintains an approximately constant block-diagonal weight. The scale goes from Yellow (larger) through green to blue (smaller)

convolution would be preceded with a $\alpha = \sqrt{1/3}$ fixed scaling factor.

In general, we have the freedom to modify the initialization of a layer, then apply a fixed multiplier before or after the layer to "undo" the increase. This allows us to change the behavior of a layer during learning by modifying the network rather than modifying the optimizer. Potentially, we can avoid the need for sophisticated adaptive optimizers by designing networks to be easily optimizable in the first place. In a sense, the need for maintaining the forward or backward variance that motivates that fan-in/fan-out initialization can be decoupled from the choice of initialization, allowing us to choose the initialization to improve the optimizability of the network.

Initialization by the principle of dynamical isometry [30, 35], a form of orthogonal initialization [24] has been shown to allow for the training of very deep networks. Such orthogonal initializations can be combined with the scaling in our theory without issue, by ensuring the input-output second moment scaling is equal to the scaling required by our theory. Our analysis is concerned with the correct initialization when layer widths change within a network, with is a separate concern from the behavior of a network in a large-depth limit, where all layers are typically taken to be the same width. In ReLU networks orthogonal initialization is less interesting, as "... the ReLU nonlinearity destroys the qualitative scaling advantage that linear networks possess for orthogonal weights versus Gaussian" [27].

7 Output second moments

A neural network's behavior is also very sensitive to the second moment of the outputs. We are not aware of any existing theory guiding the choice of output variance at initialization for the case of log-softmax losses, where it has a non-trivial effect on the back-propagated signals, although output variances of 0.01 to 0.1 are reasonable

choices to avoid saturating the nonlinearity while not being too close to zero. The output variance should **always** be checked and potentially corrected when switching initialization schemes, to avoid inadvertently large or small values.

In general, the variance at the last layer may easily be modified by inserting a fixed scalar multiplier $x_{l+1} = \alpha x_l$ anywhere in the network, and so we have complete control over this variance independently of the initialization used. For a simple ReLU convolutional network with all kernel sizes the same, and without pooling layers we can compute the output second moment when using geometric-mean initialization ($c = 2/k$) with the expression:

$$E[x_{l+1}^2] = \frac{1}{2} k_l^2 n_l E[W_l^2] E[x_l^2] = \sqrt{\frac{n_l}{n_{l+1}}} E[x_l^2]. \tag{6}$$

The application of a sequence of these layers gives a telescoping product:

$$E[x_L^2] = \left(\prod_{l=0}^{L-1} \sqrt{\frac{n_l}{n_{l+1}}} \right) E[x_0^2] = \sqrt{\frac{n_0}{n_L}} E[x_0^2].$$

so the output variance is independent of the interior structure of the network and depends only on the input and output channel sizes.

8 Biases

The conditioning of the additive biases in a network is also crucial for learning. Since our model requires that biases be initialized to zero, we can not use the gradient to weight ratio for capturing the conditioning of the biases in the network. The average singular value notion of conditioning still applies, which leads to the following definition: The scaling of the bias of a layer l, $x_{l+1} = C_l(x_l) + b_l$ is defined as:

Table 2 Comparison on 26 LIBSVM repository datasets

Method	Average normalized loss (± 0.01)	Worst in #	Best in #
Arithmetic mean	0.90	14	3
Fan-in	0.84	3	5
Fan-out	0.88	9	**12**
Geometric mean	**0.81**	**0**	6

$$\gamma_l^b = \rho^2 \frac{E[\Delta x_l^2]}{E[x_l^2]}. \tag{7}$$

In terms of the activation scaling this is:

$$
\begin{aligned}
\gamma_l^b &= \rho^2 \frac{E[\Delta x_l^2]}{E[x_l^2]} \\
&= \frac{\varsigma_l}{n E[\Delta x_l^2] E[x_l^2]} \frac{E[\Delta x_l^2]}{E[x_l^2]} \\
&= \frac{\varsigma_l}{n_l E[x_l^2]^2}.
\end{aligned} \tag{8}
$$

From Eq. 6 it's clear that when geometric initialization is used with $c = 2/k$, then:

$$n_{l+1} E[x_{l+1}^2]^2 = n_l E[x_l^2]^2,$$

and so all bias terms will be equally scaled against each other. If kernel sizes vary in the ReLU network, then a setting of c following Sect. 6.2 should be used, combined with fixed scalar multipliers that ensure that at initialization $E[x_{l+1}^2] = \sqrt{\frac{n_l}{n_{l+1}}} E[x_l^2]$.

8.1 Network input scaling balances weights against biases

It is traditional to normalize a dataset before applying a neural network so that the input vector has mean 0 and variance 1 in expectation. This scaling originated when neural networks commonly used sigmoid and tanh non-linearities, which depended heavily on the input scaling. This principle is no longer questioned today, even though there is no longer a good justification for its use in modern ReLU based networks. In contrast, our theory provides direct guidance for the choice of input scaling.

Consider the scaling factors for the bias and weight parameters in the first layer of a ReLU-based network, as considered in previous sections. We assume the data is already centered. Then the scaling factors for the weight and bias layers are:

$$\gamma_0 = n_0 k_0^2 \rho_1^2 E[x_0^2]^2 \frac{E[\Delta y_0^2]}{E[y_0^2]}, \qquad \gamma_{0b} = \rho_1^2 \frac{E[\Delta y_0^2]}{E[y_0^2]}.$$

We can cancel terms to find the value of $E[x_0^2]$ that makes these two quantities equal:

$$E[x_0^2] = \frac{1}{\sqrt{n_0 k_0^2}}.$$

In common computer vision architectures, the input planes are the 3 color channels and the kernel size is $k = 3$, giving $E[x_0^2] \approx 0.2$. Using the traditional variance-one normalization will result in the effective learning rate for the bias terms being lower than that of the weight terms. This will result in potentially slower learning of the bias terms than for the input scaling we propose. We recommend including an initial forward scaling factor in the network of $1/(n_0 k^2)^{1/4}$ to correct for this (Table 2).

9 Experimental results on 26 LIBSVM datasets

We considered a selection of dense and moderate-sparsity multi-class classification datasets from the LibSVM repository, 26 in total, collated from a variety of sources [3–5, 13, 14, 16, 18–20, 25, 28, 34]. The same model was used for all datasets, a non-convolutional ReLU network with 3 weight layers total. The inner-two layer widths were fixed at 384 and 64 nodes, respectively. These numbers were chosen to result in a larger gap between the optimization methods, less difference could be expected if a more typical $2\times$ gap was used. Our results are otherwise generally robust to the choice of layer widths.

For every dataset, learning rate, and initialization combination we ran 10 seeds and picked the median loss after 5 epochs as the focus of our study (The largest differences can be expected early in training). Learning rates in the range 2^1 to 2^{-12} (in powers of 2) were checked for each dataset and initialization combination, with the best learning rate chosen in each case based on the median of the 10 seeds. Training loss was used as the basis of our comparison as we care primarily about convergence rate, and are comparing identical network architectures. Some additional details concerning the experimental setup and which datasets were used are available in the appendix.

Table 2 shows that **geometric initialization is the most consistent** of the initialization approaches considered. The best value in each column is in bold. **It has the lowest loss**, after normalizing each dataset, and it is never the
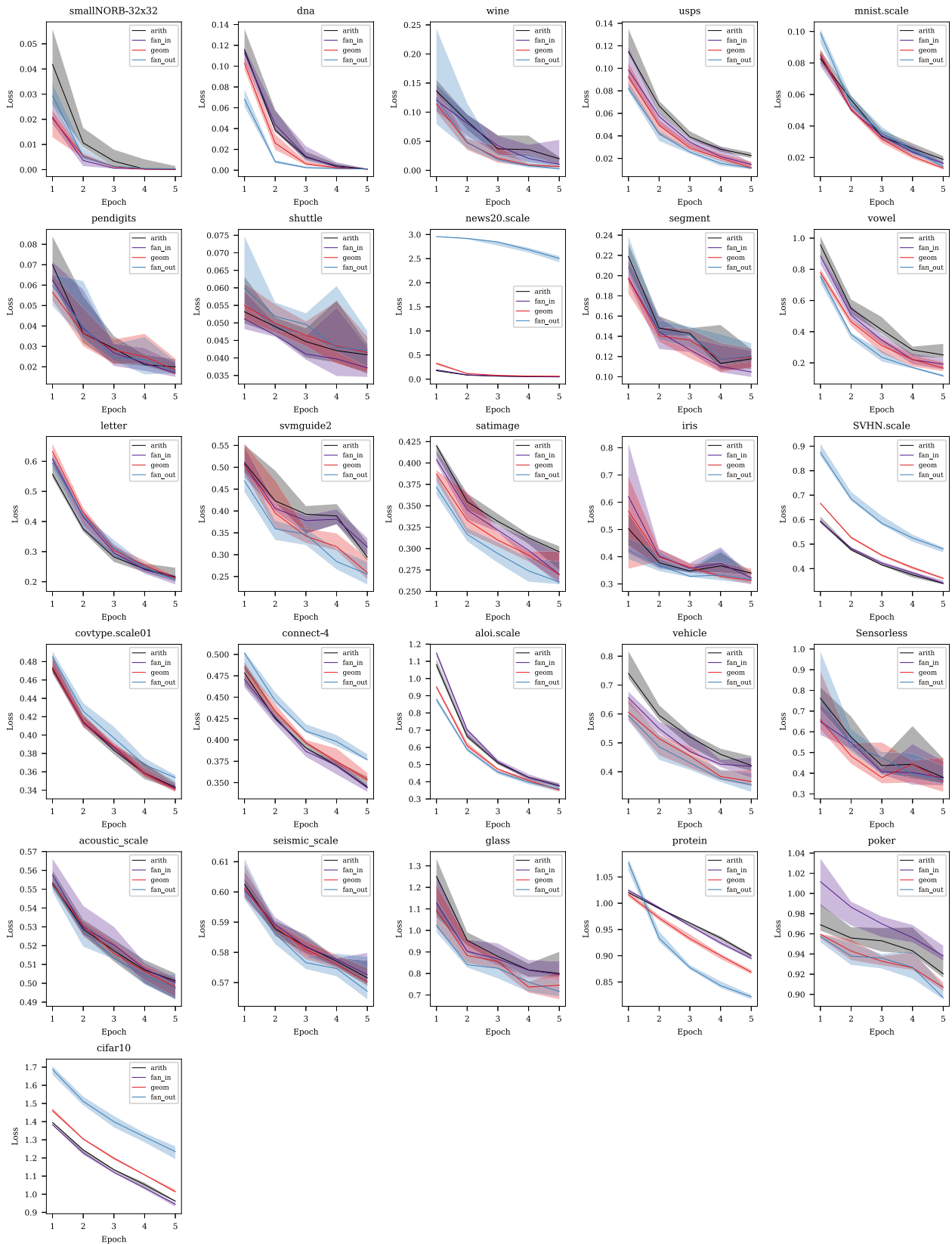
Fig. 3 Training loss comparison across 26 datasets from the LibSVM repository

worst of the 4 methods on any dataset. Interestingly, the fan-out method is most often the best method, but consideration of the per-dataset plots (Fig. 3) shows that it often completely fails to learn for some problems, which pulls up its average loss and results in it being the worst for 9 of the datasets.

10 Convolutional case: AlexNet experiments

To provide a clear idea of the effect of our scaling approach on larger networks we used the AlexNet architecture [17] as a test bench. This architecture has a large variety of filter sizes (11, 5, 3, linear), which according to our theory will affect the conditioning adversely, and which should highlight the differences between the methods. The network was modified to replace max-pooling with striding as max-pooling is not well-scaled by our theory.

Following Sect. 7, we normalize the output of the network at initialization by running a single batch through the network and adding a fixed scaling factor to the network to produce output standard deviation 0.05. We tested on CIFAR-10 following the standard practice as closely as possible, as detailed in the Appendix. We performed a geometric learning rate sweep over a power-of-two grid. Results are shown in Fig. 4 for an average of 40 seeds for each initialization. Preconditioning is a statistically significant improvement ($p = 3.9 \times 10^{-6}$) over arithmetic mean initialization and fan-in initialization, however, it only shows an advantage over fan-out at mid-iterations.
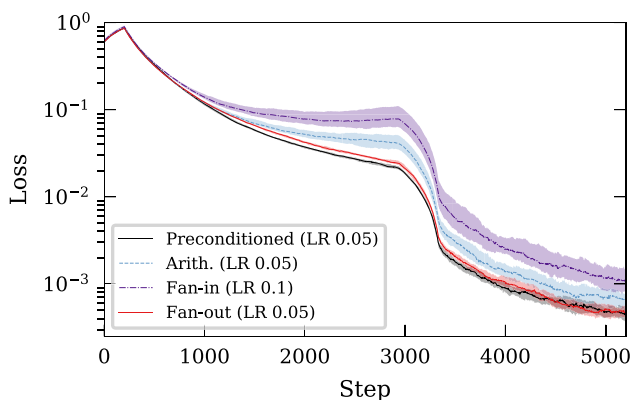
Fig. 4 CIFAR-10 training loss for a strided AlexNet architecture. The median as well as a 25%-75% IQR of 40 seeds is shown for each initialization, where for each seed a sliding window of minibatch training loss over 400 steps is used

11 Case study: unnormalized residual networks

In the case of more complex network architectures, some care needs to be taken to produce well-scaled neural networks. We consider in this section the example of a residual network, a common architecture in modern machine learning. Consider a simplified residual architecture like the following, where we have omitted ReLU operations for our initial discussion:

$$x_1 = C_0(x_0),$$
$$x_2 = B_0(x_1, \alpha_0, \beta_0),$$
$$x_3 = B_1(x_2, \alpha_1, \beta_1),$$
$$x_4 = AvgPool(x_3),$$
$$x_5 = L(x_4).$$

where for some sequence of operations F:

$$B(x, \alpha, \beta) = \alpha x + \beta F(x),$$

we further assume that $\alpha^2 + \beta^2 = 1$ and that $E[F(x)^2] = E[x^2]$ following [31]. The use of weighted residual blocks is necessary for networks that do not use batch normalization [7, 10, 32, 36].

If geometric initialization is used, then C_0 and L will have the same scaling, however, the operations within the residual blocks will not. To see this, we can calculate the activation scaling factor within the residual block. We define the shortcut branch for the residual block as the αx operation and the main branch as the $C(x)$ operation. Let $x_R = \beta C(x)$ and $x_S = \alpha x$, and define $y = x_S + x_R$.

Let ς be the scaling factor at x

$$\varsigma = n\rho^2 E[\Delta x^2]E[x^2],$$

We will use the fact that:

$$E[\Delta x^2] = (\alpha^2 + \beta^2)E[\Delta y^2]$$
$$= E[\Delta y^2].$$

From rewriting the scale factor for x_R, we see that:

$$\varsigma_R = n\rho^2 E[\Delta x_R^2]E[x_R^2]$$
$$= n\rho^2 E[\Delta x^2]E[x_R^2]$$
$$= \beta^2 n_l \rho_l^2 E[\Delta x^2]E[x^2]$$
$$= \beta^2 \varsigma.$$

A similar calculation shows that the residual branch's scaling factor is multiplied by α^2. To ensure that convolutions within the main branch of the residual block have the same scaling as those outside the block, we must multiply their initialization by a factor c. We can calculate the value of c required when geometric scaling is used for an operation in layer l in the main branch:

$$\gamma_l = \frac{\varsigma_R}{n_{l+1} n_l k_l^2 E[W_l^2]^2}$$

$$= \frac{\varsigma_R k_l^2 n_{l+1} n_l}{n_{l+1} n_l k_l^2 c_l^2} = \varsigma_R / c_l^2$$

For γ_l to match γ outside the block we thus need $\gamma_l = \varsigma_R / a_l^2 = (\beta^2 / c^2) \varsigma_R$, i.e. $c = \beta$. If the residual branch uses convolutions (such as for channel widening operations or down-sampling as in a ResNet-50 architecture) then they should be scaled by α. Modifying the initialization of the operations within the block changes $E[F(x)^2]$, so a fixed scalar multiplier must be introduced within the main branch to undo the change, ensuring $E[F(x)^2] = E[x^2]$.

11.1 Design of a pre-activation ResNet block

Using the principle above we can modify the structure of a standard pre-activation ResNet block to ensure all convolutions are well-conditioned both across blocks and against the initial and final layers of the network. We consider the full case now, where the shortcut path may include a convolution that changes the channel count or the resolution. Consider a block of the form:

$$B(x, \alpha, \beta) = \alpha S(x) + \beta F(x)$$

We consider a block with fan-in n and fan-out m. There are two cases, depending on if the block is a downsampling block or not. In the case of a downsampling block, a well-scaled shortcut branch consists of the following sequence of operations:

$$y_0 = \text{AvgPool2D}(x, \text{kernel\_size=2}, \text{stride=2}),$$
$$y_1 = y_0 + b_0,$$
$$y_2 = C(y_1, \text{op=}m, \text{ks=1}, c = \alpha/4),$$
$$y_3 = y_2 / \sqrt{\alpha/4}.$$

In our notation, C is initialized with the geometric initialization scheme of Eq. 3 using numerator $c = \alpha$. Here, *op* is output planes and *ks* is the kernel size. The constant 4 corrects for the downsampling, and the constant α is used to correct the scaling factor of the convolution as described above. In the non-downsampled case, this simplifies to

$$y_0 = x + b_0,$$
$$y_1 = C(y_0, \text{op=}m, \text{ks=1}, c = \alpha),$$
$$y_2 = y_1 / \sqrt{\alpha}.$$

For the main branch of a bottlenecked residual block in a pre-activation network, the sequence begins with a single scaling operation $x_0 = \sqrt{\beta} x$, the following pattern is used, with w being inner bottleneck width.

$$x_1 = \text{ReLU}(x_0)$$
$$x_2 = \sqrt{\frac{2}{\beta}} x_1$$
$$x_3 = x_2 + b_1$$
$$x_4 = C(x_3, \text{op=}w, \text{ks=1}, c = \beta)$$

Followed by a 3x3 conv:

$$x_5 = \text{ReLU}(x_4)$$
$$x_6 = \sqrt{\frac{2}{3\beta}} x_5$$
$$x_7 = x_6 + b_2$$
$$x_8 = C(x_7, \text{op=}w, \text{ks=3}, c = \beta)$$

and the final sequence of operations mirrors the initial operation with a downscaling convolution instead of upscaling. At the end of the block, a learnable scalar $x_9 = \frac{v}{\sqrt{\beta}} x_8$ with $v = \sqrt{\beta}$ is included following the approach of [31], and a fixed scalar corrects for any increases in the forward second moment from the entire sequence, in this case, $x_9 = \sqrt{\frac{m}{\beta n}} x_8$. This scaling is derived from Eq. 6 (β here undoes the initial beta from the first step in the block).

11.2 Experimental results

We ran a series of experiments on an unnormalized pre-activation ResNet-50 architecture using our geometric initialization and scaling scheme both within and outside of the blocks. We compared against the RescaleNet unnormalized ResNet-50 architecture. Following their guidelines, we added dropout which is necessary for good performance and used the same α/β scheme that they used. Our implementation is available in the supplementary material. We performed our experiments on the ImageNet dataset [29], using standard data preprocessing pipelines and hyper-parameters. In particular, we use batch-size 256, decay 0.0001, momentum 0.9, and learning rate 0.1 with SGD, using a 30-60-90 decreasing scheme for 90 epochs. Following our recommendation in Sect. 7, we performed a sweep on the output scaling factor and found that a 0.05 final scalar gives the best results. Across 5 seeds, our approach achieved a test set accuracy of **76.18 (SE 0.04)**, which matches the performance of the RescaleNet within our test framework of 76.13 (SE 0.03). Our approach supersedes the "fixed residual scaling" that they propose as a way of balancing the contributions of each block.

12 Related work

Our approach of balancing the diagonal blocks of the Gauss-Newton matrix has close ties to a large literature studying the input-output Jacobian of neural networks. The Jacobian is the focus of study in a number of ways. The singular values of the Jacobian are the focus of theoretical study in [30, 35], where it's shown that orthogonal initializations better control the spread of the spectrum compared to Gaussian initializations. [8, 9] also study the effect of layer width and depth on the spectrum. Regularization of the jacobian, where additional terms are added to the loss to minimize the Frobenius norm of the Jacobian, can be seen as another way to control the spectrum [12, 33], as the Frobenius norm is the sum of the squared singular values. The spectrum of the Jacobian captures the sensitivity of a network to input perturbations and is key to the understanding of adversarial machine learning, including generative modeling [26] and robustness [2, 15].

13 Conclusion

Although not a panacea, by using the scaling principle we have introduced, neural networks can be designed with a reasonable expectation that they will be optimizable by stochastic gradient methods, minimizing the amount of guess-and-check neural network design. Our approach is a step towards "engineering" neural networks, where aspects of the behavior of a network can be studied in an off-line fashion before use, rather than by a guess-implement-test-and-repeat experimental loop.

Appendix A: Forward and backward second moments

We make heavy use of the equations for forward propagation and backward propagation of second moments, under the assumption that the weights are uncorrelated to the activations or gradients. For a convolution

$$y = C_W(x),$$

with input channels n_l, output channels n_{l+1}, and square $k \times k$ kernels, these formulas are (recall our notation for the second moments is element-wise for vectors and matrices):

$$E[x_{l+1}^2] = n_l k_l^2 E[W^2]E[x_l^2], \qquad (9)$$

$$E[\Delta x_{l+1}^2] = \frac{\rho_l^2 E[\Delta x^2]}{\rho_{l+1}^2 n_{l+1} k_l^2 E[W^2]}. \qquad (10)$$

Appendix B: Scaling properties of common operations

Recall the scaling factor ς:

$$\varsigma_l = n_l \rho_l^2 E[\Delta x_l^2]E[x_l^2];$$

we show in this section how common neural network building blocks effect this factor.

Convolutions

Recall the rules for forward and back-propagated second moments for randomly initialized convolutional layers:

$$E[x_{l+1}^2] = n_l k_l^2 E[W^2]E[x_l^2],$$

$$E[\Delta x_{l+1}^2] = \frac{\rho_l^2 E[\Delta x^2]}{\rho_{l+1}^2 n_{l+1} k_l^2 E[W^2]}.$$

These relations require that W be initialized with a symmetric mean zero distribution. When applied to the scaling factor we see that:

$$\begin{aligned}
\varsigma_{l+1} &= n_{l+1}\rho_{l+1}^2 E[\Delta x_{l+1}^2]E[x_{l+1}^2] \\
&= n_{l+1}\rho_{l+1}^2 \frac{\rho_l^2 E[\Delta x^2]}{\rho_{l+1}^2 n_{l+1} k^2 E[W^2]} n_l k^2 E[W^2]E[x_l^2] \\
&= n_l \rho_l^2 E[\Delta x^2]E[x_l^2] \\
&= \varsigma_l.
\end{aligned}$$

Linear layers

These are a special case of convolutions with $k = 1$ and $\rho = 1$.

Averaging pooling

If the kernel size is equal to the stride, then element-wise we have:

$$\Delta x_l = \frac{1}{k^2} \Delta x_{l+1},$$

So:

$$E[\Delta x_l^2] = \frac{1}{k^4} E[\Delta x_{l+1}^2].$$

ReLU

If the input to a ReLU is centered and symmetrically distributed, then during the forward pass, half of the inputs are zeroed out in expectation, meaning that $E[x_{l+1}^2] = \frac{1}{2}E[x_l^2]$. The backward operation just multiplies Δx_{l+1} by the zero

pattern used during the forward pass, so it also zeros half of the entries, giving $E[\Delta x_l^2] = \frac{1}{2}E[\Delta x_{l+1}^2]$ So:

$$
\begin{aligned}
\varsigma_{l+1} &= n_{l+1}\rho_{l+1}^2 E[\Delta x_{l+1}^2]E[x_{l+1}^2] \\
&= n_l\rho_l^2 E[\Delta x_{l+1}^2]E[x_{l+1}^2] \\
&= n_l\rho_l^2 2E[\Delta x_l^2]\frac{1}{2}E[x_l^2] \\
&= \varsigma_l.
\end{aligned}
$$

Dropout

The reasoning for dropout is essentially the same as for the ReLU. If nodes are dropped out with probability p, then $E[x_{l+1}^2] = (1-p)E[x_l^2]$ and $E[\Delta x_l^2] = (1-p)E[\Delta x_{l+1}^2]$. So scaling is maintained. Note that in the PyTorch implementation, during training the outputs are further multiplied by $1/(1-p)$.

Scalar multipliers

Consider a layer:

$$
x_{l+1} = u_l x_l.
$$

Then

$$
E[x_{l+1}^2] = u_l^2 E[x_l^2],
$$

and

$$
E[\Delta x_l^2] = u_l^2 E[\Delta x_l^2],
$$

so the forward and backward signals are multiplied by u_l^2, which maintains scaling.

Residual blocks

Consider a residual block of the form:

$$
x_{l+1} = x_l + R(x_l),
$$

for some operation R. Suppose that $E[R(x)^2] = s^2 E[x^2]$ for some constant s. The forward signal second moment gets multiplied by $(1+s^2)$ after the residual block:

$$
E[x_{l+1}^2] = E[x_l^2](1+s^2).
$$

The backwards signal second moment is also multiplied by s:

$$
E[\Delta x_l^2] = (1+s^2)E[\Delta x_{l+1}^2].
$$

So:

$$
\begin{aligned}
\varsigma_{l+1} &= E[\Delta x_{l+1}^2]E[x_{l+1}^2] \\
&= \frac{1}{(1+s^2)}E[\Delta x_l^2]E[x_l^2](1+s^2) \\
&= \varsigma_l.
\end{aligned}
$$

Appendix C: Extrinsic and intrinsic form equivalence

Recall the extrinsic definition of γ_l:

$$
\varsigma_l = n_l\rho_l^2 E[\Delta x_l^2]E[x_l^2].
$$

We rewrite this as:

$$
n_l\rho_l^2 E[x_l^2] = \frac{\varsigma_l}{E[\Delta x_l^2]}.
$$

Then by using the forward and backward relations Eqs. 9 and 10:

$$
\begin{aligned}
\gamma_l &= n_l^{\mathrm{in}}k_l^2\rho_{l+1}^2 E[x_l^2]^2\frac{E[\Delta x_{l+1}^2]}{E[x_{l+1}^2]} \\
&= k_l^2 E[x_l^2]\frac{\rho_{l+1}^2\varsigma_l E[\Delta x_{l+1}^2]}{\rho_l^2 E[\Delta x_l^2]E[x_{l+1}^2]} \quad (\varsigma \text{ substitution}) \\
&= k_l^2 E[x_l^2]\frac{\rho_{l+1}^2\varsigma_l E[\Delta x_{l+1}^2]}{\rho_l^2 E[\Delta x_l^2]n_l k_l^2 E[W_l^2]E[x_l^2]} \quad (\text{forward}) \\
&= E[\Delta x_{l+1}^2]\cdot\frac{\rho_{l+1}^2\varsigma_l}{\rho_l^2 E[\Delta x_l^2]n_l E[W_l^2]} \\
&= \frac{\rho_l^2 E[\Delta x_l^2]}{\rho_{l+1}^2 n_{l+1}k^2 E[W_l^2]}\cdot\frac{\rho_{l+1}^2\varsigma_l}{\rho_l^2 E[\Delta x_l^2]n_l E[W_l^2]} \quad (\text{back}) \\
&= \frac{\varsigma_l}{n_{l+1}n_l k^2 E[W_l^2]^2}.
\end{aligned}
$$

Appendix D: The weight gradient ratio is equal to GR scaling for MLP models

Theorem 3 The weight-gradient ratio v_l is equal to the scaling γ_l factor under the assumptions of Sect. 3.

Proof First, we rewrite Eq. 9 to express $E[W^2]$ in terms of forward moments:

$$E[x_{l+1}^2] = n_l k_l^2 E[W^2] E[x_l^2],$$
$$\therefore E[W^2] = \frac{E[x_{l+1}^2]}{n_l k_l^2 E[x_l^2]}.$$

For the gradient w.r.t to weights of a convolutional layer, we have:

$$E[\Delta W_l^2] = \rho_l^2 E[x_l^2] E[\Delta x_{l+1}^2].$$

Therefore:

$$v_l = \frac{E[\Delta W_l^2]}{E[W_l^2]}$$
$$= n_l^{\text{in}} k_l^2 E[x_l^2]^2 \frac{E[\Delta x_{l+1}^2]}{E[x_{l+1}^2]}$$
$$= \gamma_l.$$

Appendix E: Scaling of scalar multipliers

Consider the layer:

$$x_{l+1} = u_l x_l$$

with a single learnable scalar u_l. Using :

$$E[x_{l+1}^2] = E[u_l^2] E[x_l^2],$$

we have:

$$E[u_l^2] = \frac{E[x_{l+1}^2]}{E[x_l^2]}.$$

Likewise from the equations of back-prop we have:

$$\Delta u = \sum_c^{n_l} \sum_i^{\rho_l} \sum_j^{\rho_l} \Delta x_{l+1,c,i,j} x_{l,c,i,j},$$

so

$$E[\Delta u^2] = n_l \rho_l^2 E[\Delta x_{l+1}^2] E[x_l^2].$$

Therefore:

$$v_l = \frac{E[\Delta u_l^2]}{E[u_l^2]} = n_l \rho_l^2 E[x_l^2]^2 \frac{E[\Delta x_{l+1}^2]}{E[x_{l+1}^2]}. \tag{11}$$

This equation is the same for a 1x1 convolutional layer. We can write the scaling factor in terms of the weight u more directly, by rearranging the scaling rule as:

$$\frac{\varsigma_l}{E[\Delta x_l^2] E[x_l^2]} = n_l \rho_l^2,$$

and substituting it into Eq. 11:

$$n_l \rho_l^2 E[x_l^2]^2 \frac{E[\Delta x_{l+1}^2]}{E[x_{l+1}^2]} = \frac{\varsigma_l}{E[\Delta x_l^2] E[x_l^2]} E[x_l^2]^2 \frac{E[\Delta x_{l+1}^2]}{E[x_{l+1}^2]}$$
$$= \frac{\varsigma_l}{E[\Delta x_l^2]} E[x_l^2] \frac{E[\Delta x_{l+1}^2]}{E[x_{l+1}^2]}$$
$$= \frac{\varsigma_l}{E[\Delta x_l^2]} E[x_l^2] \frac{E[\Delta x_l^2]}{E[u_l^2]^2 E[x_l^2]}$$
$$= \frac{\varsigma_l}{E[u_l^2]^2}.$$

To ensure that the scaling factor matches that of convolutions used in the network, suppose that geometric initialization is used with global constant c, then each convolution has $\gamma_l = \varsigma_l/c^2$, so we need:

$$E[u_l^2] = c,$$

or just $u_l = \sqrt{c}$. Notice that if we had used a per-channel scalar instead, then the layer scaling would not match the scaling of convolution weights, which would result in uneven layer scaling in the network. This motivates using scalar rather than channel-wise scaling factors.

Appendix F: The Gauss–Newton matrix

Standard ReLU classification and regression networks have a particularly simple structure for the Hessian with respect to the input, as the network's output is a piecewise-linear function g feed into a final layer consisting of a convex log-softmax operation, or a least-squares loss. This structure results in the Hessian with respect to the input being equivalent to its *Gauss–Newton* approximation. The Gauss–Newton matrix can be written in a factored form, which is used in the analysis we perform in this work. We emphasize that this is just used as a convenience when working with diagonal blocks, the GN representation is not an approximation in this case.

The (Generalized) Gauss-Newton matrix G is a positive semi-definite approximation of the Hessian of a non-convex function f, given by factoring f into the composition of two functions $f(x) = h(g(x))$ where h is convex, and g is approximated by its Jacobian matrix J at x, for the purpose of computing G:

$$G = J^T (\nabla^2 h(g(x))) J.$$

The GN matrix also has close ties to the Fisher information matrix [23], providing another justification for its use.

Surprisingly, the Gauss-Newton decomposition can be used to compute diagonal blocks of the Hessian with respect to the weights W_l as well as the inputs [23]. To see this, note that for any activation y_l, the layers above may be treated in a combined fashion as the h in a $f(W_l) =$

$h(g(W_l))$ decomposition of the network structure, as they are the composition of a (locally) linear function and a convex function and thus convex. In this decomposition $g(W_l) = W_l x_l + b_l$ is a function of W_l with x_l fixed, and as this is linear in W_l, the Gauss-Newton approximation to the block is thus not an approximation.

Appendix G: GR scaling derivation

Our quantity of interest is the average squared singular value of G_l, which is simply equal to the (element-wise) non-central second moment of the product of G with a i.i.d normal random vector r:

$$E[(G_l r)^2] = E[(J_l^T R_l J_l r)^2].$$

Recall that our notation $E[X^2]$ refers to the element-wise non-central second moment of the vector. To compute the second moment of the elements of $G_l r$, we can calculate the second moment of matrix-random-vector products against J_l, R_l and J_l^T separately since R is uncorrelated with J_l, and the back-propagated gradient Δy_l is uncorrelated with y_l (Assumption A3).

Jacobian products J_l and J_l^T

Note that each row of J_l has n_l^{in} non-zero elements, each containing a value from x_l. This structure can be written as a block matrix,

$$J_l = \begin{bmatrix} x_l & 0 & 0 \\ 0 & x_l & 0 \\ 0 & 0 & \ddots \end{bmatrix}, \tag{12}$$

Where each x_l is a $1 \times n_l^{\text{in}}$ row vector. This can also be written as a Kronecker product with an identity matrix as $I_{n_l^{\text{out}}} \otimes x_l$. The value x_l is i.i.d random at the bottom layer of the network. For layers further up, the multiplication by a random weight matrix from the previous layer ensures that the entries of x_l are identically distributed. So we have:

$$E\left[(J_l r)^2\right] = n_l^{\text{in}} E[r^2] E[x_l^2] = n_l^{\text{in}} E[x_l^2]. \tag{13}$$

Note that we didn't assume that the input x_l is mean zero, so $Var[x_l] \neq E[x_l^2]$. This is needed as often the input to a layer is the output from a ReLU operation, which will not be mean zero.

For the transposed case, we have a single entry per column, so when multiplying by an i.i.d random vector u we have:

$$E\left[(J_l^T u)^2\right] = E[u^2] E[x_l^2]. \tag{14}$$

Upper Hessian R_l product

Instead of using $R_l u$, for any arbitrary random u, we will instead compute it for $u = y_l / E[y_l^2]$, it will have the same expectation since both $J_l r$ and y_l are uncorrelated with R_l. The piecewise linear structure of the network above y_l with respect to the y_l makes the structure of R_l particularly simple. It is a least-squares problem $g(y_l) = \frac{1}{2}\|\Phi y_l - t\|^2$ for some Φ that is the linearization of the remainder of the network. The gradient is $\Delta y = \Phi^T(\Phi y - t)$ and the Hessian is simply $R = \Phi^T \Phi$. So we have that

$$\begin{aligned} E[\Delta y_l^2] &= E\left[\frac{1}{n_l^{\text{out}}}\|\Phi^T(\Phi y - t)\|^2\right] \\ &= E\left[\frac{1}{n_l^{\text{out}}}\|\Phi^T \Phi y\|^2\right] + E\left[\frac{1}{n_l^{\text{out}}}\|\Phi^T t\|^2\right] \\ &= E\left[\frac{1}{n_l^{\text{out}}}\|\Phi^T \Phi y\|^2\right] + O\left(\frac{1}{n_l^{\text{out}}}\right). \\ &= E\left[(R_l y_l)^2\right] + O\left(\frac{1}{n_l^{\text{out}}}\right). \end{aligned}$$

Applying this gives:

$$E(R_l u)^2 = E[u^2] E[(R_l y_l)^2] / E[y_l^2] \tag{15}$$

$$= E[u^2] E[\Delta y_l^2] / E[y_l^2] + O\left(\frac{E[u^2]}{n_l^{\text{out}} E[y_l^2]}\right) \tag{16}$$

Combining

To compute $E[(G_l r)^2] = E[(J_l^T R_l J_l r)^2]$ we then combine the simplifications from Eqs. 13, 14 and 15 to give:

$$E[(G_l r)^2] = n_l^{\text{in}} E[x_l^2]^2 \frac{E[\Delta y_l^2]}{E[y_l^2]} + O\left(\frac{n_l^{\text{in}} E[x_l^2]^2}{n_l^{\text{out}} E[y_l^2]}\right).$$

Appendix H: Details of LibSVM dataset input/output scaling

To prevent the results from being skewed by the number of classes and the number of inputs affecting the output variance, the logit output of the network was scaled to have standard deviation 0.05 after the first minibatch evaluation for every method, with the scaling constant fixed thereafter. LayerNorm was used on the input to whiten the data. Weight decay of 0.00001 was used for every dataset. To aggregate the losses across datasets we divided by the worst loss across the initializations before averaging.

Declarations

Conflict of interest The authors declare no conflicts of interest.

Code availability An implementation of the modified rescale net is available upon request.

References

1. Bottou L (1988) Reconnaissance de la parole par reseaux connexionnistes. In: Proceedings of Neuro Nimes 88
2. Chan A, Tay Y, Ong YS, Fu J (2020) Jacobian adversarially regularized networks for robustness. In: Eighth international conference on learning representations (ICLR2020)
3. Dua D, Graff C (2017) UCI machine learning repository. http://archive.ics.uci.edu/ml
4. Duarte M, Hu YH (2004) Vehicle classification in distributed sensor networks. J Parallel Distrib Comput
5. Feng C, Sutherland A, King S, Muggleton S, Henery R (1993) Comparison of machine learning classifiers to statistics and neural networks. AI & Stats Conf. 93
6. Glorot X, Bengio Y (2010) Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the thirteenth international conference on artificial intelligence and statistics
7. Hanin B (2018) Which neural net architectures give rise to exploding and vanishing gradients? In: Advances in neural information processing systems 31, pp. 582–591. Curran Associates, Inc
8. Hanin B, Nica M (2020) Products of many large random matrices and gradients in deep neural networks. Commun Math Phys
9. Hanin B, Paouris G: Non-asymptotic results for singular values of gaussian matrix products (2020)
10. Hanin B, Rolnick D (2018) How to start training: the effect of initialization and architecture. In: Advances in Neural Information Processing Systems, vol. 31
11. He K, Zhang X, Ren S, Sun J (2015) Delving deep into rectifiers: surpassing human-level performance on imagenet classification. In: Proceedings of the 2015 ieee international conference on computer vision (ICCV) (2015)
12. Hoffman J, Roberts DA, Yaida S (2019) Robust learning with jacobian regularization
13. Hsu CW, Chang CC, Lin CJ (2003) A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University (2003)
14. Hull JJ (1994) A database for handwritten text recognition research. IEEE Transactions on Pattern Analysis and Machine Intelligence
15. Jakubovitz D, Giryes R (2018) Improving dnn robustness to adversarial attacks using jacobian regularization. In: Computer Vision – ECCV 2018. Springer International Publishing
16. Krizhevsky A (2009) Learning multiple layers of features from tiny images
17. Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp 1097–1105
18. Lang K (1995) Newsweeder: Learning to filter netnews. In Proceedings of the twelfth international conference on machine learning
19. LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proceedings of the IEEE
20. LeCun Y, Huang F, Bottou L (2004) Learning methods for generic object recognition with invariance to pose and lighting. IEEE computer society conference on computer vision and pattern recognition (CVPR)
21. LeCun YA, Bottou L, Orr GB, Müller KR (2012) Neural Networks: Tricks of the Trade, chap. Springer, Efficient BackProp
22. Li L, Talwalkar AS (2019) Random search and reproducibility for neural architecture search. CoRR
23. Martens J (2014) New insights and perspectives on the natural gradient method. In: ArXiv e-prints
24. Mishkin D, Matas J (2016) All you need is a good init
25. Netzer Y, Wang T, Coates A, Bissacco A, Wu B, Ng AY (2011) Reading digits in natural images with unsupervised feature learning. NIPS Workshop on Deep Learning and Unsupervised Feature Learning
26. Nie W, Patel A (2019) Towards a better understanding and regularization of gan training dynamics. In: UAI
27. Pennington J, Schoenholz SS, Ganguli S (2018) The emergence of spectral universality in deep networks. Proceedings of the 21st international conference on artificial intelligence and statistics (AISTATS)
28. Rocha A, Goldenstein S (2014) Multiclass from binary: expanding one-vs-all, one-vs-one and ecoc-based approaches. IEEE transactions on neural networks and learning systems
29. Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M, Berg AC, Fei-Fei L (2015) ImageNet large scale visual recognition challenge. Int J Comput Vis (IJCV) 115(3):211–252. https://doi.org/10.1007/s11263-015-0816-y
30. Saxe AM, McClelland JL, Ganguli S (2014) Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. 2nd international conference on learning representations (ICLR 2014)
31. Shao J, Hu K, Wang C, Xue X, Raj B (2020) Is normalization indispensable for training deep neural networks?. In: 34th conference on neural information processing systems (NeurIPS 2020)
32. Szegedy C, Ioffe S, Vanhoucke V, Alemi AA (2017) Inception-v4, inception-resnet and the impact of residual connections on learning. In: Proceedings of the Thirty-First AAAI conference on artificial intelligence
33. Varga D, Csiszárik A, Zombori Z (2018) Gradient regularization improves accuracy of discriminative models
34. Wang JY (2002) Application of support vector machines in bioinformatics. Master's thesis, Department of Computer Science and Information Engineering, National Taiwan University
35. Xiao L, Bahri Y, Sohl-Dickstein J, Schoenholz SS, Pennington J (2018) Dynamical isometry and a mean field theory of cnns: How

to train 10,000-layer vanilla convolutional neural networks. ICML

36. Zhang H, Dauphin YN, Ma T (2019) Residual learning without normalization via better initialization. In: International conference on learning representations

37. Zoph B, Le QV (2016) Neural architecture search with reinforcement learning. CoRR