DEEP LEARNING WITHOUT SHORTCUTS: SHAPING THE KERNEL WITH TAILORED RECTIFIERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training very deep neural networks is still an extremely challenging task. The common solution to this is to add shortcut connections and normalization layers, which are both crucial ingredients in the ResNet architecture. However, there is strong evidence to suggest that ResNets behave more like ensembles of shallower networks than truly deep ones. Recently, it was shown that deep vanilla networks (i.e. networks without normalization layers or shortcut connections) can be trained as fast as ResNets by applying certain transformations to their activation functions. However, this method (called Deep Kernel Shaping) isn't fully compatible with ReLUs, and produces networks that exhibit significantly more overfitting than ResNets of similar size on ImageNet. In this work, we rectify this situation by developing a new type of transformation which is perfectly compatible with a variant of ReLUs – Leaky ReLUs. We show in experiments that our method, which introduces negligible extra computational cost, achieves tests accuracies with vanilla deep networks that are competitive with ResNets (of the same width/depth), and significantly higher than those obtained with the Edge of Chaos (EOC) method. And unlike with EOC, the test accuracies we obtain do not get worse with depth.

1 INTRODUCTION

Thanks to many architectural and algorithmic innovations, the recent decade has witnessed the unprecedented success of deep learning in various high-profile challenges, e.g., the ImageNet recognition task (Krizhevsky et al., 2012), the challenging board game of Go (Silver et al., 2017) and human-like text generation (Brown et al., 2020). Among them, shortcut connections (He et al., 2016a; Srivastava et al., 2015) and normalization layers (Ioffe & Szegedy, 2015; Ba et al., 2016) are two architectural components of modern networks that are critically important for achieving fast training at very high depths, and feature prominently in the ubiquitous ResNet architecture of He et al. (2016b).

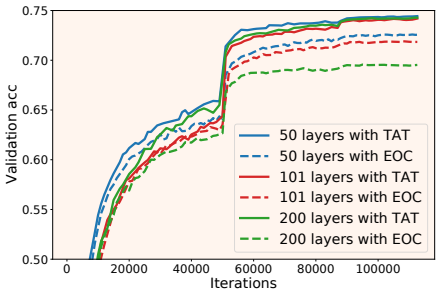

Figure 1: Top-1 ImageNet validation accuracy of vanilla deep networks with ReLU activation function initialized using either EOC or TAT (ours) and trained with K-FAC.

Despite the success of ResNets, there is significant evidence to suggest that the primary reason they work so well is that they reduce the effective depth and resemble ensembles of shallower networks during training (Veit et al., 2016), which lets them avoid the common pathologies associated with very deep networks (e.g. Hochreiter et al., 2001). In this sense, the question of whether truly deep networks can be efficient and effectively trained on challenging tasks remains an open one.

As argued by Oyedotun et al. (2020) and Ding et al. (2021), the multi-branch topology of ResNets has certain drawbacks. First, it is memory-inefficient at inference time, as the input to every residual block has to be kept in memory until the final addition. Second, it imposes constraints on the architectural specification. Particularly, ResNets require the last conv layer of every residual block to produce tensors of the same shape. Third, the classical interpretation of why deep networks perform well – because of the hierarchical feature representations they produce – does not strictly apply to ResNets, due to their aforementioned tendency to behave like ensembles of shallower networks. Beyond the drawbacks of ResNets, training vanilla deep neural networks (which we define as networks without shortcut connections or normalization layers) is an interesting research problem in its own right,

and finding a solution could open the path to discovering new model architectures. However, recent progress in this direction has not fully succeeded in matching the performance of ResNets.

Schoenholz et al. (2017) used a mean-field analysis of deep MLPs to choose initialization variances for the weights and bias parameters, and showed that the resulting method – called Edge of Chaos (EOC) – allowed vanilla networks to be trained at very high depths on small datasets. Building on EOC, and incorporating dynamical isometry theory, Xiao et al. (2018) was able to train vanilla networks with Tanh units[1] at depths of up to 10,000. While impressive, these EOC-initialized networks trained significantly slower than standard ResNets of the same depth, and also exhibited significantly worse generalization performance. While Oyedotun et al. (2020) was able to narrow the generalization gap between vanilla networks and ResNets, their experiments were limited to networks with only 30 layers, and their networks required many times more parameters. More recently, Martens et al. (2021) introduced a method called Deep Kernel Shaping (DKS) for initializing and transforming networks (and their activation functions) based on an analysis of their initialization-time kernel properties. They showed that their approach enabled vanilla networks to train faster than previous methods, even matching the speed of similarly sized ResNets when combined with stronger optimizers like K-FAC (Martens & Grosse, 2015; Grosse & Martens, 2016) or Shampoo (Anil et al., 2020). However, their method isn't fully compatible with ReLUs, and in their experiments (which focused on training speed) their networks exhibited significantly more overfitting than ResNets.

Inspired by both DKS and the line of work using mean-field theory, we propose a new method called Tailored Activation Transformation (TAT). TAT inherits the main advantages of DKS, while working particularly well with the "Leaky ReLU" activation function. Importantly, TAT, together with Leaky ReLU, enjoys certain favourable kernel properties at initialization time that are not achieved by DKS. While being easy to implement and introducing negligible extra computational cost, TAT enables very deep vanilla neural networks to be trained on ImageNet without the use of any additional architectural tricks. Using TAT, we demonstrate for the first time that a 50-layer vanilla deep network can nearly match the test accuracy of its ResNet counterpart when trained on Imagenet. And unlike the EOC method, the test accuracy does not decrease with depth using TAT (see Figure 1). Furthermore, TAT can also be applied to ResNets without normalization layers, allowing them to match or even exceed the test accuracy of standard ResNets of the same width/depth.

2 Background

Our main tool of analysis will be kernel functions for neural networks (Neal, 1996; Cho & Saul, 2009; Daniely et al., 2016) and the related Q/C maps (Saxe et al., 2013; Poole et al., 2016; Martens et al., 2021). In this section, we introduce our notation and some key concepts used throughout.

2.1 Kernel Function Approximation for Wide Networks

For simplicity, we start with the kernel function approximation for feedforward fully-connected networks, and discuss its extensions to convolutional networks and non-feedforward networks later. In particular, we will assume a network that is defined by a sequence of L *combined layers* (each of which is an affine transformation followed by the elementwise activation function ϕ) as follows:

$$x^{l+1} = \phi\left(W_l x^l + b_l\right) \in \mathbb{R}^{d_{l+1}}, \tag{1}$$

with weights $W_l \in \mathbb{R}^{d_{l+1} \times d_l}$ initialized as $W_l \overset{\text{iid}}{\sim} \mathcal{N}(0, 1/d_l)$ (or scale-corrected uniform orthogonal matrices (Martens et al., 2021)), and biases $b_l \in \mathbb{R}^{d_{l+1}}$ initialized to zero. Due to the randomness of the initial parameters θ, the network can be viewed as random feature model $f_\theta^l(x) \triangleq x^\ell$ at each layer l (with $x^0 = x$) at initialization time. This induces a random kernel defined as follows:

$$\kappa_f^l(x_1, x_2) = \frac{1}{d_l} f_\theta^l(x_1)^\top f_\theta^l(x_2). \tag{2}$$

Given these assumptions, as the width of each layer goes to infinity, $\kappa_f^l(x_1, x_2)$ converges in probability (see Theorem 3) to a deterministic kernel $\tilde{\kappa}_f^l(x_1, x_2)$ that has a simple form, and can be computed layer by layer as follows:

$$\Sigma^{l+1} = \mathbb{E}_{z \sim \mathcal{N}(0, \Sigma^l)}\left[\phi(z)\phi(z)^\top\right], \text{ with } \Sigma^l = \begin{bmatrix} \tilde{\kappa}_f^l(x_1, x_1) & \tilde{\kappa}_f^l(x_1, x_2) \\ \tilde{\kappa}_f^l(x_1, x_2) & \tilde{\kappa}_f^l(x_2, x_2) \end{bmatrix}, \tag{3}$$

where $\tilde{\kappa}_f^0(x_1, x_2) = \kappa_f^0(x_1, x_2) = x_1^\top x_2 / d_0$.

[1] Dynamical isometry is unavailable for ReLU (Pennington et al., 2017), even with orthogonal weighs.

2.2 LOCAL Q/C MAPS

By equation 3, any diagonal entry q_i^{l+1} of Σ^{l+1} only depends on the corresponding diagonal entry q_i^l of Σ^l. Hence, we obtain the following recursion for these diagonal entries, which we call *q values*:

$$q_i^{l+1} = \mathcal{Q}(q_i^l) = \mathbb{E}_{z\sim\mathcal{N}(0,q_i^l)}[\phi(z)^2] = \mathbb{E}_{z\sim\mathcal{N}(0,1)}\left[\phi(\sqrt{q_i^l}z)^2\right], \text{ with } q_i^0 = \|x_i\|^2/d_0 \tag{4}$$

where $\mathcal{Q}$ is the *local Q map*. We note that q_i^l is an approximation of $\kappa_f^l(x_i, x_i)$. Analogously, one can write the recursion for the normalized off-diagonal entries, which we call *c values*, as:

$$c^{l+1} = \mathcal{C}(c^l, q_1^l, q_2^l) = \frac{\mathbb{E}_{\left[\begin{smallmatrix} z_1 \\ z_2 \end{smallmatrix}\right]\sim\mathcal{N}(0,\Sigma^l)}\left[\phi(z_1)\phi(z_2)\right]}{\sqrt{\mathcal{Q}(q_1^l)\mathcal{Q}(q_2^l)}}, \text{ with } \Sigma^l = \begin{bmatrix} q_1^l & \sqrt{q_1^l q_2^l}c^l \\ \sqrt{q_1^l q_2^l}c^l & q_2^l \end{bmatrix}, \tag{5}$$

where $\mathcal{C}$ is the *local C map* and $c^0 = x_1^\top x_2/d_0$. We note that c^l is an approximation of the cosine similarity between $f_\theta^l(x_1)$ and $f_\theta^l(x_2)$. Because $\mathcal{C}$ is a three dimensional function, it is difficult to analyze, as the associated q values can vary wildly for distinct inputs. However, by scaling the inputs to have norm $\sqrt{d_0}$, and rescaling ϕ so that $\mathcal{Q}(1) = 1$, it follows that $q_i^l = 1$ for all l. This allows us to treat $\mathcal{C}$ as a one dimensional function from $[-1, 1]$ to $[-1, 1]$ satisfying $\mathcal{C}(1) = 1$. Additionally, it can be shown that $\mathcal{C}$ possesses special structure as a *positive definite function* (see Appendix A.3 for details). Going forward, we will thus assume that $q_i^0 = 1$, and that ϕ is scaled so that $\mathcal{Q}(1) = 1$.

2.3 EXTENSIONS TO CONVOLUTIONAL NETWORKS AND MORE COMPLEX TOPOLOGIES

As argued in Martens et al. (2021), Q/C maps can also be defined for convolutional networks if one adopts a Delta initialization (Balduzzi et al., 2017; Xiao et al., 2018), in which all weights but those in the center of the filter are initialized to zero. Intuitively, this makes convolutional networks behave like a collection of fully-connected networks operating independently over feature map locations. As such, the Q/C map computations for a feed-forward convolutional network are the same as above. Martens et al. (2021) also gives formula to compute q and c values for weighted sum operations between the outputs of multiple layers (without nonlinearities), thus allowing more complex network topologies. In particular, the sum operation's output q value is given by $q = \sum_{i=1}^n w_i^2 q_i$, and its output c value is given by $\frac{1}{q}\sum_{i=1}^n w_i^2 q_i c_i$. In order to maintain the property that all q values are 1 in the network, we will assume that sum operations are *normalized* in the sense that $\sum_{i=1}^n w_i^2 = 1$.

Following Martens et al. (2021), we will extend the definition of Q/C maps to include *global Q/C maps*, which describe the behavior of entire networks. Global maps, denoted by $\mathcal{Q}_f$ and $\mathcal{C}_f$ for a given network f, can be computed by applying the above rules for each layer in f. For example, the global C map of a three-layer network f is simply $\mathcal{C}_f(c) = \mathcal{C} \circ \mathcal{C} \circ \mathcal{C}(c)$. Like the local C map, global C maps are positive definite functions (see Appendix A.3). In this work, we restrict our attention to the family of networks comprising of combined layers, and normalized sums between the output of multiple affine layers, for which we can compute global Q/C maps. For all the theorems in this work, we assume this family of networks.

2.4 Q/C MAPS FOR RESCALED RESNETS

ResNets consist of a sequence of residual blocks, each of which computes the sum of a residual branch (which consists of a small multi-layer convolutional network) and a shortcut branch (which copies the block's input). In order to analyze ResNets we will consider the modified version used in Shao et al. (2020) and Martens et al. (2021) which **removes the normalization layers** found in the residual branches, and replaces the sum at the end of each block with a normalized sum. These networks, which we will call *rescaled ResNets*, are defined by the following recursion:

$$x^{l+1} = wx^l + \sqrt{1 - w^2}\mathcal{R}(x^l), \tag{6}$$

where $\mathcal{R}$ is the residual branch, and w is the *shortcut weight* (which must be in $[-1, 1]$). Applying the previously discussed rules for computing Q/C maps, we get $q_i^l = 1$ for all l and

$$c^{l+1} = w^2 c^l + (1 - w^2)\mathcal{C}_\mathcal{R}(c^l). \tag{7}$$

3 EXISTING SOLUTIONS AND THEIR LIMITATIONS

Global Q/C maps can be intuitively understood as a way of characterizing signal propagation through the network f at initialization time. The q value approximates the squared magnitude of the activation

vector, so that $\mathcal{Q}_f$ describe the contraction or expansion of this magnitude through the action of f. On the other hand, the c value approximates the cosine similarity of the function values for different inputs, so that $\mathcal{C}_f$ describes how well f preserves this cosine similarity from its input to its output.

Standard initializations methods (LeCun et al., 1998; Glorot & Bengio, 2010; He et al., 2015) are motivated through an analysis of how the variance of the activations evolves throughout the network. This can be viewed as a primitive form of Q map analysis, and from that perspective, these methods are trying to ensure that q values remain stable throughout the network by controlling the local Q map. This is necessary for trainability, since large or tiny q values can cause numerical issues, saturated activation functions (which have implications for C maps), and problems with scale-sensitive losses. However, as was first observed by Schoenholz et al. (2017), a well-behaved C map is also necessary for trainability. When the global C map is close to a constant function (i.e. degenerate) on $(-1, 1)$, which easily happens in deep networks (including those with stable q values), this means that the network's output will appear either constant or random looking, and won't convey any useful information about the input. Xiao et al. (2020) and Martens et al. (2021) give more formal arguments for why this leads to slow optimization and/or poor generalization under gradient descent.

Several previous works (Schoenholz et al., 2017; Yang & Schoenholz, 2017; Hayou et al., 2019) attempt to achieve a well-behaved global C map by choosing the variance of the initial weights and biases in each layer such that $\mathcal{C}'(1) = 1$ – a procedure which is referred to as *Edge of Chaos* (EOC). However, this approach only slows down the convergence (with depth) of the c values from exponential to sublinear (Hayou et al., 2019), and does not solve the fundamental issue of degenerate global C maps for very deep networks. In particular, the global C map of a deep network with ReLU and EOC initialization rapidly concentrates around 1 as depth increases (see Figure 2). While EOC allows very deep vanilla networks to be trained, the training speed and generalization performance is typically much worse than for comparable ResNets.

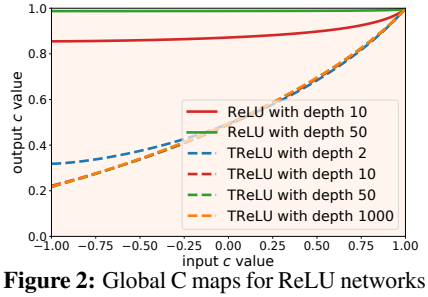

Figure 2: Global C maps for ReLU networks (EOC) and TReLU networks ($C_f(0) = 0.5$). The global C map of a TReLU network converges to a nice function as depth increases (proved in Proposition 2) while that of a ReLU network collapses.

To address these problems, Martens et al. (2021) introduced DKS, which enforces the conditions $\mathcal{C}_f(0) = 0$ and $\mathcal{C}_f'(1) = \zeta$ (for some modest constant $\zeta > 1$) directly on the network's global C map $\mathcal{C}_f$. They show that these conditions, along with the positive definiteness of C maps, cause $\mathcal{C}_f$ to be close to the identity and thus well-behaved. In addition to these C map conditions, DKS enforces that $\mathcal{Q}(1) = 1$ and $\mathcal{Q}'(1) = 1$, which lead to constant q values of 1 in the network, and lower kernel approximation error (respectively). Overall DKS enforces four conditions on $\mathcal{Q}$ and $\mathcal{C}$, which it does by applying a model class-preserving transformation $\hat{\phi}(x) = \gamma(\phi(\alpha x + \beta) + \delta)$. with non-trainable parameters α, β, γ and δ. The hyperparameter ζ is chosen to be suffiently greater than 1 (e.g. 1.5) in order to prevent the transformed activation functions from looking *too linear* (as they would be exactly linear if $\zeta = 1$), which Martens et al. (2021) argue makes it hard for the network to achieve nonlinear behavior during training. Using DKS, they were able to match the training speed of ResNets on ImageNet with vanilla networks using K-FAC. However, DKS is not fully compatible with ReLUs, and their networks fell substantially short of ResNets in terms of generalization performance.

4 Tailored Activation Transformation (TAT)

The reason why DKS is not fully compatible with ReLUs is that they are positive homogeneous, i.e. $\phi(\alpha x) = \alpha\phi(x)$ for $\alpha \geq 0$. This makes the γ parameter of the transformed activation function redundant, thus reducing the degrees of freedom with which to enforce DKS's four Q/C map conditions. Martens et al. (2021) attempt to circumvent this issue by dropping the condition $\mathcal{Q}'(1) = 1$, which leads to vanilla deep networks that are trainable, but slower to optimize compared to using DKS with other activation functions. This is a significant drawback for DKS, as the best generalizing deep models often use ReLU-family activations. We therefore set out to investigate other possible remedies – either in the form of different activation functions, new Q/C map conditions, or both. To this end, we adopt a ReLU-family activation function with an extra degree of freedom (known as "Leaky ReLU"), and modify the Q/C map conditions in order to preserve certain desirable properties of this choice. The resulting method, which we name Tailored Activation Transformation (TAT) achieves competitive generalization performance with ResNets in our experiments.

Table 1: Comparison of different methods applied to a network f.

EOC (smooth)	EOC (LReLU)	DKS	TAT (smooth)	TAT (LReLU)
q^∞ exists $\mathcal{C}'(1, q^\infty, q^\infty) = 1$	$\mathcal{Q}(q) = q$ $\mathcal{C}'(1) = 1$	$\mathcal{Q}(1) = 1$ $\mathcal{Q}'(1) = 1$ $\mathcal{C}_f(0) = 0$ $\mathcal{C}'_f(1) = \zeta$	$\mathcal{Q}(1) = 1$ $\mathcal{Q}'(1) = 1$ $\mathcal{C}'_f(1) = 1$ $\mathcal{C}''_f(1) = \chi$	$\mathcal{Q}(q) = q$ $\mathcal{C}'_f(1) = 1$ $\mathcal{C}_f(0) = \eta$

4.1 Tailored Activation Transformation for Leaky ReLUs

One way of addressing the issue of DKS's partial incompatibility with ReLUs is to consider a slightly different activation function – namely the Leaky ReLU (LReLU) (Maas et al., 2013):

$$\phi_\alpha(x) = \max\{x, 0\} + \alpha \min\{x, 0\}, \tag{8}$$

where α is the *negative slope parameter*. While using LReLUs with $\alpha \neq 0$ in place of ReLUs changes the model class, it doesn't limit the model's expressive capabilities compared to ReLU, as assuming $\alpha \neq \pm 1$, one can simulate a ReLU network with a LReLU network of the same depth by doubling the number of neurons (see Proposition 3). Rather than using a fixed value for α, we will use it as an extra parameter to satisfy our desired Q/C map conditions. Define $\tilde{\phi}_\alpha(x) = \sqrt{\frac{2}{1+\alpha^2}}\phi_\alpha(x)$. By Lemma 1, the local Q and C maps for this choice of activation function are:

$$\mathcal{Q}(q) = q \quad \text{and} \quad \mathcal{C}(c) = c + \tfrac{(1-\alpha)^2}{\pi(1+\alpha^2)}\left(\sqrt{1-c^2} - c\cos^{-1}(c)\right). \tag{9}$$

Note that the condition $\mathcal{Q}(q) = q$ is actually *stronger* than DKS's Q map conditions ($\mathcal{Q}(1) = 1$ and $\mathcal{Q}'(1) = 1$), and has the potential to reduce kernel approximation errors in finite width networks compared to DKS, as it provides a better guarantee on the stability of $\mathcal{Q}_f$ w.r.t. small random perturbations of the q values at each layer. Additionally, because the form of $\mathcal{C}$ does not depend on the layer's input q value, random perturbations in the q values won't affect the accuracy of the c value computations. Notably, if one uses the negative slope parameter to transform LReLUs with DKS (as we do in our experiments with $\hat{\phi}(x) = \gamma(\phi_\alpha(x+\beta) + \delta)$), these properties will not be achieved.

Another consequence of using $\tilde{\phi}_\alpha(x)$ for our activation function is that we have $\mathcal{C}'(1) = 1$ as in EOC. If combined with the condition $\mathcal{C}(0) = 0$ (which is used to achieve $\mathcal{C}_f(0) = 0$ in DKS) this would imply by Theorem 1 that $\mathcal{C}$ is the identity function, which by equation 9 is only true when $\alpha = 1$, thus resulting in a linear network. In order to avoid this situation, and to prevent $\tilde{\phi}_\alpha$ appearing "too linear" more generally, we instead set the value of α so that $\mathcal{C}_f(0) = \eta$ for a hyperparameter $0 \leq \eta \leq 1$. As shown in the following theorem, η controls how close $\mathcal{C}_f$ is to the identity, thus allowing us to achieve a well-behaved global C map without making $\tilde{\phi}_\alpha$ too linear:

Theorem 1. *For a network f with $\tilde{\phi}_\alpha(x)$ as its activation function (for $\alpha \geq 0$), we have*

$$\max_{c\in[-1,1]} |\mathcal{C}_f(c) - c| \leq \min\{4\mathcal{C}_f(0), 1 + \mathcal{C}_f(0)\}, \quad \max_{c\in[-1,1]} \left|\mathcal{C}'_f(c) - 1\right| \leq \min\{4\mathcal{C}_f(0), 1\} \tag{10}$$

Another motivation for using $\tilde{\phi}_\alpha(x)$ as an activation function is given by the following proposition:

Proposition 1. *The global C map of a feedforward network with $\tilde{\phi}_\alpha(x)$ as its activation function is equal to that of a rescaled ResNet of the same depth (see Section 2.4) with normalized ReLU activation $\phi(x) = \sqrt{2}\max(x, 0)$, shortcut weight $\sqrt{\frac{\alpha}{1+\alpha^2}}$, and residual branch $\mathcal{R}$ consisting of a combined layer (or just a normalized ReLU activation) followed by an affine layer.*

This result implies that at initialization, a vanilla network using $\tilde{\phi}_\alpha$ behaves similarly to a ResNet[2], a property that is quite desirable given the huge success that ResNets have already demonstrated in many machine learning tasks.

In summary we have the following three conditions:

$$\mathcal{Q}(q) = q, \quad \mathcal{C}'_f(1) = 1, \quad \mathcal{C}_f(0) = \eta, \tag{11}$$

which we achieve by picking the negative slope parameter α so that $C_f(0) = \eta$. We define the Tailored Rectifier (`TReLU`) to be $\tilde{\phi}_\alpha$ with α chosen in this way. Note that the first two conditions are

[2]That's only at initialization time. During training, they can exhibit very different behaviors. At initialization, even a shallow network can achieve the same C map as deep ones with a carefully designed activation function.

also true when applying the EOC method to LReLUs, and its only the third which sets `TReLU` apart. While this might seem like a minor difference, it actually matters a lot to the behavior of the global C map. This can be seen in Figure 2 where the c value quickly converges towards 1 with depth under EOC, resulting in a degenerate global C map. By contrast, the global C map of `TReLU` for a fixed η converges rapidly to a nice function, suggesting a very deep vanilla network with `TReLU` has the same well-behaved global C map as a shallow network. We prove this in Proposition 2 by showing the local C map in equation 9 converges to an ODE as we increase the depth. For direct comparison of all Q/C map conditions, we refer the readers to Table 1.

For the hyperparameter $0 \leq \eta \leq 1$, we note that a value very close to 0 will produce a network that is "too linear", while a value very close to 1 will give rise to a degenerate C map. In practice we use $\eta = 0.9$ or 0.95, which seems to work well in most settings. Once we decide on η, we can solve the value α using binary search by exploiting the closed-form form of $\mathcal{C}$ in equation 9 to efficiently compute $\mathcal{C}_f(0)$. For instance, if f is a 100 layer vanilla network, one can compute $\mathcal{C}_f(0)$ as follows:

$$\mathcal{C}_f(0) = \overbrace{\mathcal{C} \circ \mathcal{C} \cdots \mathcal{C} \circ \mathcal{C}}^{\text{100 times}}(0), \tag{12}$$

which is a function of α. The same basic procedure also applies to rescaled ResNets.

4.2 Tailored Activation Transformation for Smooth Activation Functions

Unlike LReLU, most activation functions don't have closed-form formulas for their local C maps. As a result, the computation of $\mathcal{C}_f(0)$ involves the numerical approximation of many two-dimensional integrals to high precision (as in equation 5), which can be quite expensive. One alternative way to control how close $\mathcal{C}_f$ is to the identity, while maintaining the condition $\mathcal{C}_f'(1) = 1$, is to modulate its second derivative $\mathcal{C}_f''(1)$. The validity of this approach is established by the following theorem:

Theorem 2. *Suppose f is a network with a smooth activation function. If $\mathcal{C}_f'(1) = 1$, then we have*

$$\max_{c \in [-1,1]} |\mathcal{C}_f(c) - c| \leq 2\mathcal{C}_f''(1), \quad \max_{c \in [-1,1]} \left|\mathcal{C}_f'(c) - 1\right| \leq 2\mathcal{C}_f''(1) \tag{13}$$

Given $\mathcal{C}(1) = 1$ and $\mathcal{C}'(1) = 1$, a straightforward computation shows that $\mathcal{C}_f''(1) = L\mathcal{C}''(1)$ if f is a L-layer vanilla network. From this we obtain the following four local Q/C map conditions:

$$\mathcal{Q}(1) = 1, \quad \mathcal{Q}'(1) = 1, \quad \mathcal{C}''(1) = \chi/L, \quad \mathcal{C}'(1) = 1. \tag{14}$$

To achieve these we adopt the same activation transformation as DKS (see Section 3). We emphasize that these conditions cannot be used with LReLU, as LReLU networks have $\mathcal{C}''(1) = \infty$. By equation 4 and basic properties of expectations, we have

$$1 = \mathcal{Q}(1) = \mathbb{E}_{z \sim \mathcal{N}(0,1)} \left[\hat{\phi}(z)^2\right] = \gamma^2 \mathbb{E}_{z \sim \mathcal{N}(0,1)} \left[(\phi(\alpha z + \beta) + \delta)^2\right] \tag{15}$$

so that $\gamma = \mathbb{E}_{z \sim \mathcal{N}(0,1)} \left[(\phi(\alpha z + \beta) + \delta)^2\right]^{-1/2}$. To obtain the values for α, β and δ, we can treat the remaining three conditions as a three dimensional nonlinear system, which can be written as follows:

$$\begin{aligned} &\mathcal{Q}'(1) = \mathbb{E}_{z \sim \mathcal{N}(0,1)} \left[\hat{\phi}(z)\hat{\phi}'(z)z\right] = 1, \\ &\mathcal{C}''(1) = \mathbb{E}_{z \sim \mathcal{N}(0,1)} \left[\hat{\phi}''(z)^2\right] = \chi/L, \ \mathcal{C}'(1) = \mathbb{E}_{z \sim \mathcal{N}(0,1)} \left[\hat{\phi}'(z)^2\right] = 1. \end{aligned} \tag{16}$$

We are not aware of any closed-form solution of this system. However, each expectation is a one dimensional integral, and so can be quickly evaluated to high precision using Gaussian quadrature. One can then use black-box nonlinear equation solvers, such as modified Powell's method (Powell, 1964), to obtain a solution. See Appendix E.2 for an example implementation in NumPy.

5 Experiments

We experimentally evaluate our proposed method TAT in training deep vanilla networks on ImageNet (Deng et al., 2009). The main goal we have is *not* to achieve state-of-the-art, but to compare TAT as fairly as possible with existing methods. In all of our experiments we use ResNet V2 (He et al., 2016b) as the main reference architecture, which uses ReLU by default. For rescaled ResNets

(as described in equation 6), we remove all normalization layers and weight the shortcut branch by a fixed w. For networks without batch normalization, we add dropout to the penultimate layer to regularize the networks as was done in Brock et al. (2021b). We train the models with 90 epochs and a batch size of 1024, unless stated otherwise. For `TReLU`, we conduct grid search over η in $\{0.9, 0.95\}$. For all other details, please see Appendix C.

5.1 Towards removing batch normalization

Two crucial components for the successful training of very deep neural networks are shortcut connections and batch normalization (BN). As argued in De & Smith (2020) and Shao et al. (2020), BN implicitly biases the residual blocks toward the identity function, which makes the network better behaved at initialization time, and thus easier to train. This suggests that one can compensate for the removal of BN layers, at least in terms of their effect on the behaviour of the network at initialization time, by down-scaling the residual branch of each residual block. Arguably, almost all recent work on training deep networks without normalization layers has adopted this idea by introducing multipliers on the residual branches, which may or not be optimized during training (Zhang et al., 2018; Shao et al., 2020; Bachlechner et al., 2020; Brock et al., 2021a;b).

In Table 2, we demonstrate that one can close most of the gap with standard ResNets by simply adopting the modification in equation 6 without using BN layers. By further replacing ReLU with `TReLU`, we can exactly match the performance of standard ResNets. With K-FAC as the optimizer, the rescaled ResNet with shortcut weight $w = 0.9$ is only 0.5 shy of the validation accuracy (76.4) of the standard ResNet. Further taking $\eta = 0.9$ with TAT, we match the performance of standard ResNet with shortcut weight $w = 0.8$. Here we note that the negative slope chosen for `TReLU` for the case of $w = 0.9$ is close to 0, so using `TReLU` makes little difference (vs standard ReLU).

Table 2: Top-1 validation accuracy of rescaled ResNet50 with varying shortcut weights. We set $\eta = 0.9$ for `TReLU`.

Optimizer	Standard ResNet	Activation	Rescaled ResNet (w)			
			0.0	0.5	0.8	0.9
K-FAC	76.4	ReLU	72.6	74.5	75.6	75.9
		TReLU	74.6	75.5	**76.4**	75.9
SGD	76.3	ReLU	63.7	72.4	73.9	75.0
		TReLU	71.0	72.6	**76.0**	74.8

5.2 The difficulty of removing shortcut connections

Although the aforementioned works have shown that it is possible to achieve competitive results without normalization layers, they all rely on the use of shortcut connections to make the network look more linear at initialization. It remains an open challenge to train deep neural networks without shortcuts. Given that normalization layers make networks better behaved, it is natural to conjecture that they might be sufficient to enable fast training of networks without shortcuts. We test this hypothesis by running shortcut-free networks with either BN or LN layers. As shown in Table 3, these changes do not seem to make a significant difference, especially with strong optimizers like K-FAC. These findings are in agreement those of Yang et al. (2019), who showed that shortcut-free networks with BN still suffer from exploding gradients.

Table 3: ImageNet top-1 validation accuracies of shortcut-free Networks on ImageNet. **BN** and **LN** stand for batch and layer normalization respectively.

Networks	Optimizers	vanilla	BN	LN
ResNet50	K-FAC	72.6	72.8	72.7
	SGD	63.7	72.6	58.1
ResNet101	K-FAC	71.8	67.6	72.0
	SGD	41.6	43.4	28.6

5.3 Training Deep Neural Networks without Shortcuts

The main motivation for developing the TAT is to be able to train deep vanilla networks to achieve generalization performance similar to standard ResNets. In our investigations we also include rescaled ResNets with a shortcut weight of either 0.0 (i.e. vanilla networks) or 0.8. In Table 4 we can see that with a strong optimizer like K-FAC, we can close to gap on the 50 layer network to only 1.8% accuracy when training for 90 epochs, and further down to 0.6% when training for 180 epochs. For ResNet-101, the gaps are 3.6% and 1.7% respectively, which we show can be further reduced with wider networks (see Table 9). To our knowledge, this is the first time that a deep vanilla network has been trained to such a high test accuracy on ImageNet. In addition, our networks have fewer parameters and run faster than standard ResNets, and use less memory at inference time due to the removal of shortcuts and BN layer. The gap when using SGD as the optimizer is noticeably larger, which we further discuss in Section 5.5. Lastly, using rescaled ResNets with a shortcut weight of 0.8 and `TReLU`, we can exactly match or even surpass the standard ResNet results.

Table 4: ImageNet top-1 validation accuracy. For rescaled ResNets ($w = 0.0$ or $w = 0.8$), we do not include any normalization layer. For standard ResNets, batch normalization is included. By default, ReLU activation is used for standard ResNet while we use `TReLU` for rescaled networks.

Network	Optimizer	90 epochs			180 epochs		
		standard	$w = 0.0$	$w = 0.8$	standard	$w = 0.0$	$w = 0.8$
ResNet50	K-FAC	76.4	74.6	76.4	76.6	76.0	77.0
	SGD	76.3	71.0	76.0	76.6	74.0	76.8
ResNet101	K-FAC	77.8	74.2	77.8	77.6	75.9	78.4
	SGD	77.9	70.0	77.3	77.6	73.8	77.4

5.4 Comparisons with existing approaches

Comparison with EOC. We first compare TAT to EOC on vanilla deep networks. For EOC with ReLUs we simply use He initialization (He et al., 2015), as ReLUs always satisfy $\mathcal{C}'(1) = 1$ when the bias variance is zero. For Tanh, a direct comparison with EOC is more difficult since there are in theory infinitely many choices of initialization. Here, we use $(\sigma_w, \sigma_b) = (1.302, 0.2)$[3], as suggested in Hayou et al. (2019). In Table 5, we can see that in all the settings, networks constructed with TAT outperform EOC by a significant margin, especially with SGD. Another observation is that the performance of EOC-networks drops with the increasing depth. In the case of ReLU activation function, we notice, by comparing Figure 1 to 6, that EOC-networks tends to suffer from worse overfitting as depth increases.

Table 5: ImageNet top-1 validation accuracy comparison between EOC and TAT on deep vanilla networks.

Network	Optimizer	Method	(L)ReLU	Tanh
ResNet50	K-FAC	EOC	72.6	70.6
		TAT	**74.6**	**73.1**
	SGD	EOC	63.7	55.7
		TAT	**71.0**	**69.5**
ResNet101	K-FAC	EOC	71.8	69.2
		TAT	**74.2**	**72.8**
	SGD	EOC	41.6	54.0
		TAT	**70.0**	**69.0**

Comparison with DKS. The closest approach to TAT in the existing literature is DKS, whose similarity and drawbacks we discussed in Section 4. We compare our approach and DKS on both positive homogeneous functions like the LReLU[4], as well as on smooth functions like the SoftPlus and Tanh. For the smooth activations we perform a grid search over three representative values $\{0.2, 0.3, 0.5\}$ for χ in TAT and $\{1.5, 10.0, 100.0\}$ for ζ in DKS and report only the best performing one. In addition with From the results are show in Table 7 we observe that TAT together with LReLU (`TReLU`) performs best over all combinations in most settings, and the gaps become larger when we remove dropout. In particular, we find that the gains of using `TReLU` mostly come from improved generalization. Here, one key difference between `TReLU` and the other activations is that it automatically satisfies the condition $\mathcal{Q}(q) = q$, and its local C map is independent of the input q value (see equation 9), which likely reduces the associated kernel approximation error (as discussed in Section 4.1). One might speculate that this is responsible for the improved generalization performance we see from `TReLU` versus the other activation functions.

Comparison with PReLU. The Parametric ReLU (PReLU) was introduced in He et al. (2015), which generalizes the classical rectified unit. Specifically, PReLU adaptively learns the negative slope parameter of the LReLU along with all other trainable parameters during training. At the first glance, this seems to be similar to what we're doing with `TReLU`. However, this is not the case, since in TAT we only adjust the negative slope parameter before training and then fix it throughout training.

Table 6: Comparison with PReLU (last two columns).

Network	Optimizer	TReLU	0.0	0.25
ResNet50	K-FAC	74.6	72.5	73.6
	SGD	71.1	66.7	67.9
ResNet101	K-FAC	74.2	71.9	72.8
	SGD	70.0	54.3	66.3

Here, we compare our `TReLU` to PReLU on ImageNet. For PReLU, we include two different initializations: one with 0 negative slope (effectively ReLU) and another with 0.25 negative slope (which was used in He et al. (2015)). We report the results on deep vanilla networks in Table 6 (see Appendix D.5 for the results of rescaled ResNets). In all these settings, our method outperforms PReLU by a large margin, suggesting a better-initialized negative slope could be crucial. In principle,

[3]We also run experiments with $(\sigma_w, \sigma_b) = (1.0, 0.0)$ and the scheme in Pennington et al. (2017); Xiao et al. (2018) for dynamical isometry, the results are worse than what we reported in the table.

[4]For DKS, we set the negative slope as a parameter and adopt the transformation $\hat{\phi}(x) = \gamma(\phi_\alpha(x + \beta) + \delta)$.

Table 7: Comparisons between TAT and DKS. The numbers on the right hand of / are results without dropout. The methods with $*$ are introduced in this paper.

Network	Optimizer	Shortcut Weight	TAT			DKS		
			LReLU*	SoftPlus*	Tanh*	LReLU*	SoftPlus	Tanh
ResNet50	K-FAC	$w = 0.0$	**74.6/74.2**	74.4/**74.2**	73.1/72.9	74.3/74.3	74.3/73.7	72.9/72.9
		$w = 0.8$	**76.4**/75.9	**76.4**/75.0	74.8/74.4	76.2/**76.2**	76.3/75.1	74.7/74.5
	SGD	$w = 0.0$	71.1/71.1	70.2/70.0	69.5/69.5	70.4/70.4	**71.8/71.4**	69.2/69.2
		$w = 0.8$	**76.0/75.8**	74.3/73.8	72.4/72.2	73.4/73.0	75.2/74.1	72.8/72.8
ResNet101	K-FAC	$w = 0.0$	**74.2/74.2**	74.1/73.4	72.8/72.5	73.5/73.5	73.9/73.1	72.5/72.4
		$w = 0.8$	**77.8/77.0**	76.6/75.7	75.8/75.1	76.8/76.7	76.8/75.6	75.9/75.7
	SGD	$w = 0.0$	70.0/**70.0**	**70.3**/68.8	69.0/67.8	68.3/68.3	68.3/68.3	69.8/69.8
		$w = 0.8$	**77.3/76.0**	75.3/75.3	73.8/73.5	74.9/74.6	76.3/75.1	74.6/74.6

these two can be even combined together, i.e., we first initialize the negative slope parameter with TAT and then optimize it during training. However, we do not see any benefit to doing this in practice.

5.5 The role of strong optimzier

Table 8: Batch size scaling.

Optimizer	Batch size					
	128	256	512	1024	2048	4096
K-FAC	74.5	74.4	74.5	74.6	74.2	72.0
SGD	72.7	72.6	72.7	71.0	69.3	62.0
LARS	72.4	72.3	72.6	71.8	71.3	70.2

One interesting phenomenon we observed in our experiments, which echoes the findings of Martens et al. (2021), is that strong optimizer K-FAC performs significantly better than SGD on vanilla deep networks, given the same training epochs and batch size. One plausible explanation would be that K-FAC works better than SGD in the large-batch setting and the batch size of 1024 is already over the "critical batch size" of SGD (at which scaling efficiency begins to drop). Indeed, it was shown by Zhang et al. (2019) that optimization algorithms that employ preconditioning, specifically Adam and K-FAC, result in much larger critical batch sizes.

To investigate this further, we tested the effect of varying the batch size between 128 to 4096. As shown in Table 8, K-FAC performs equally well over all different batch sizes except 4096 (due to increased overfitting) while the performance of SGD starts to drop when we increase the batch size past 512. Surprisingly, we observe similar trend for the LARS optimizer (You et al., 2019) which was designed for large-batch training. Even at the smallest batch size we tested (128) K-FAC still outperforms SGD by a gap of 1.8%. We conjecture the reason behind this to be that vanilla networks without normalization and shortcuts give rise to ill-conditioned loss landscapes which slow down simpler optimizers like SGD. To investigate further, we also ran SGD at batch size 512 for 360 epochs, and with a cosine learning rate schedule as in (Loshchilov & Hutter, 2016). As shown in Figure 3, SGD indeed catches up eventually with K-FAC towards the end of the run. One may argue that K-FAC is more expensive for each step, but this can amortized significantly by updating the inverse Fisher less frequently, leading to an almost negligible overhead compared to SGD. Our experiments show that doing this does not appear to significantly affect per-iteration optimization performance (see Figure 5 in the Appendix).

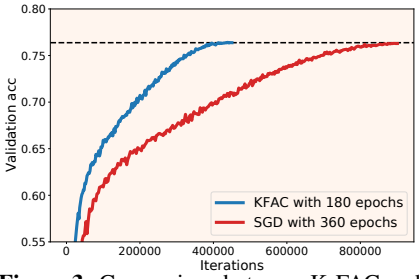

Figure 3: Comparison between K-FAC and SGD on ImageNet.

6 Conclusions

In this work we considered the problem of training and generalization in vanilla deep neural networks (i.e. those without shortcut connections and normalization layers). To address this we developed a novel method that modifies the activation functions in a way tailored to the specific architecture, and which enables us to achieve generalization performance on par with standard ResNets of the same width/depth. Unlike the most closely related approach (DKS), our method is fully compatible with ReLU-family activation functions, and in fact achieves its best performance with them. By removing the need for shortcut connections, we believe our method could enable further research into deep models and their representations. In addition, our method may enable new architectures to be trained for which existing techniques (like shortcuts and normalization layers) are insufficient.

REPRODUCIBILITY STATEMENT

Here we discuss our efforts to facilitate the reproducibility of the paper. Firstly, we present `JAX` example code for the proposed TAT in Appendix E.1 (for LReLU) and Appendix E.2 (for smooth activations). Moreover, we give the full implementation details in Appendix C, including how do we tune hyperparameters, algorithm details, initialization details and model details.

REFERENCES

Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.

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. PMLR, 2017.

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

Andrew Brock, Soham De, and Samuel L Smith. Characterizing signal propagation to close the performance gap in unnormalized resnets. In *International Conference on Learning Representations*, 2021a.

Andrew Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. *arXiv preprint arXiv:2102.06171*, 2021b.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Youngmin Cho and Lawrence Saul. Kernel methods for deep learning. *Advances in Neural Information Processing Systems*, 22:342–350, 2009.

Amit Daniely, Roy Frostig, and Yoram Singer. Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity. *Advances In Neural Information Processing Systems*, 29:2253–2261, 2016.

Soham De and Sam Smith. Batch normalization biases residual blocks towards the identity function in deep networks. *Advances in Neural Information Processing Systems*, 33, 2020.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. Repvgg: Making vgg-style convnets great again. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13733–13742, 2021.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010.

Roger Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers. In *International Conference on Machine Learning*, pp. 573–582. PMLR, 2016.

Soufiane Hayou, Arnaud Doucet, and Judith Rousseau. On the impact of the activation function on deep neural networks training. In *International conference on machine learning*, pp. 2672–2680. PMLR, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016a.

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, 2016b.

Tom Hennigan, Trevor Cai, Tamara Norman, and Igor Babuschkin. Haiku: Sonnet for JAX, 2020. URL http://github.com/deepmind/dm-haiku.

Matteo Hessel, David Budden, Fabio Viola, Mihaela Rosca, Eren Sezener, and Tom Hennigan. Optax: composable gradient transformation and optimisation, in jax!, 2020. URL http://github.com/deepmind/optax.

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015.

Hassan K. Khalil. Nonlinear systems third edition. 2008.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

Alex Krizhevsky et al. Learning multiple layers of features from tiny images. 2009.

Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*. Springer, 1998.

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

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning*, 2013.

James Martens. On the validity of kernel approximations for orthogonally-initialized neural networks. *arXiv preprint arXiv:2104.05878*, 2021.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pp. 2408–2417. PMLR, 2015.

James Martens, Andy Ballard, Guillaume Desjardins, Grzegorz Swirszcz, Valentin Dalibard, Jascha Sohl-Dickstein, and Samuel S Schoenholz. Rapid training of deep neural networks without skip connections or normalization layers using deep kernel shaping. *ArXiv*, 2021.

Elizabeth S Meckes. *The random matrix theory of the classical compact groups*, volume 218. Cambridge University Press, 2019.

Radford M Neal. Bayesian learning for neural networks. *Lecture notes in statistics*, 118, 1996.

Oyebade K Oyedotun, Djamila Aouada, Björn Ottersten, et al. Going deeper with neural networks without skip connections. In *2020 IEEE International Conference on Image Processing (ICIP)*, pp. 1756–1760. IEEE, 2020.

Jeffrey Pennington, Samuel S Schoenholz, and Surya Ganguli. Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 4788–4798, 2017.

Ben Poole, Subhaneil Lahiri, Maithra Raghu, Jascha Sohl-Dickstein, and Surya Ganguli. Exponential expressivity in deep neural networks through transient chaos. *Advances in neural information processing systems*, 29:3360–3368, 2016.

Michael JD Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal*, 7(2):155–162, 1964.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

Samuel S Schoenholz, Justin Gilmer, Surya Ganguli, and Jascha Sohl-Dickstein. Deep information propagation. In *International Conference on Learning Representations*, 2017.

Jie Shao, Kai Hu, Changhu Wang, Xiangyang Xue, and Bhiksha Raj. Is normalization indispensable for training deep neural network? *Advances in Neural Information Processing Systems*, 33, 2020.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

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

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

Andreas Veit, Michael J Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks. *Advances in neural information processing systems*, 29:550–558, 2016.

Lechao Xiao, Yasaman Bahri, Jascha Sohl-Dickstein, Samuel Schoenholz, and Jeffrey Pennington. Dynamical isometry and a mean field theory of cnns: How to train 10,000-layer vanilla convolutional neural networks. In *International Conference on Machine Learning*, pp. 5393–5402. PMLR, 2018.

Lechao Xiao, Jeffrey Pennington, and Samuel Schoenholz. Disentangling trainability and generalization in deep neural networks. In *International Conference on Machine Learning*, pp. 10462–10472. PMLR, 2020.

Ge Yang and Samuel Schoenholz. Mean field residual networks: On the edge of chaos. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

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

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *British Machine Vision Conference 2016*. British Machine Vision Association, 2016.

Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George Dahl, Chris Shallue, and Roger B Grosse. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. *Advances in neural information processing systems*, 32:8196–8207, 2019.

Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. In *International Conference on Learning Representations*, 2018.

A BACKGROUND

A.1 KERNEL FUNCTION APPROXIMATION

In section 2.1, we claimed that the kernel defined in equation 2 would converge to a deterministic kernel, as the width of each layer goes to infinity. To be specific, we have the following result bounding the kernel approximation error.

Theorem 3 (Adapted from Theorem 2 of Daniely et al. (2016))**.** *Consider a fully-connected network of depth L with weights initialized independently using a standard Gaussian fan-in initialization. Further suppose that the activation function ϕ is C-bounded (i.e. $\|\phi\|_\infty \leq C$, $\|\phi'\|_\infty \leq C$ and $\|\phi''\|_\infty \leq C$ for some constant C) and satisfies $\mathbb{E}_{z\sim\mathcal{N}(0,1)}[\phi(z)^2] = 1$, and that the width of each layer is greater than or equal to $(4C^4)^L \log(8L/\delta)/\epsilon^2$. Then at initialization time, for inputs x_1 and x_2 satisfying $\|x_1\|^2 = \|x_2\|^2 = \dim(x_1)$, we have that*

$$\left|\kappa_f^L(x_1, x_2) - \tilde{\kappa}_f^L(x_1, x_2)\right| \leq \epsilon$$

with probability at least $1 - \delta$.

The bound in Theorem 3 predicts an exponential dependence on the depth L of the minimum required width of each layer. However, for a network with ReLU activations, this dependence is only quadratic in L, as established in the following theorem:

Theorem 4 (Adapted from Theorem 3 of Daniely et al. (2016))**.** *Consider a fully-connected network of depth L with ReLU activations and weights initialized independently using a He initialization (He et al., 2015), and suppose that the width of each layer is greater than or equal to $L^2 \log(L/\delta)/\epsilon^2$. Then at initialization time, for inputs x_1 and x_2 satisfying $\|x_1\|^2 = \|x_2\|^2 = \dim(x_1)$, and $\epsilon \lesssim \frac{1}{L}$, we have that*

$$\left|\kappa_f^L(x_1, x_2) - \tilde{\kappa}_f^L(x_1, x_2)\right| \leq \epsilon$$

with probability at least $1 - \delta$.

Although Theorems 3 and 4 are only applicable to Gaussian initializations, a similar bound has been given by Martens (2021) for scaled uniform orthogonal initializations in the case that $L = 1$. Moreover, Martens (2021) conjectures that their result could be extended to general values of L.

A.2 C MAP DERIVATIVE

Poole et al. (2016) gave the following nice formula for the derivative of C map of a combined layer with activation function ϕ:

$$\mathcal{C}'(c, q_1, q_2) = \frac{\sqrt{q_1 q_2}}{\sqrt{\mathcal{Q}(q_1)\mathcal{Q}(q_2)}} \mathbb{E}_{z_1, z_2 \sim \mathcal{N}(0,1)} \left[\phi'\left(\sqrt{q_1} z_1\right) \phi'\left(\sqrt{q_2}\left(c z_1 + \sqrt{1 - c^2} z_2\right)\right)\right]. \quad (17)$$

For a rigorous proof of this result we refer the reader to Martens et al. (2021).

One can iterate this formula to obtain a similar equation for higher-order derivatives:

$$\mathcal{C}^{(i)}(c, q_1, q_2) = \frac{(q_1 q_2)^{(i/2)}}{\sqrt{\mathcal{Q}(q_1)\mathcal{Q}(q_2)}} \mathbb{E}_{z_1, z_2 \sim \mathcal{N}(0,1)} \left[\phi^{(i)}\left(\sqrt{q_1} z_1\right) \phi^{(i)}\left(\sqrt{q_2}\left(c z_1 + \sqrt{1 - c^2} z_2\right)\right)\right]. \quad (18)$$

A.3 SOME USEFUL PROPERTIES OF C MAPS

In this section we will assume that $q_1 = q_2 = 1$.

Observe that $\mathcal{C}(1) = \mathbb{E}_{z\sim\mathcal{N}(0,1)}\left[\phi(z)^2\right] = 1$ and that $\mathcal{C}$ maps $[-1, 1]$ to $[-1, 1]$ (which follows from its interpretation as computing cosine similarities for infinitely wide networks). Moreover, $\mathcal{C}$ is a positive definite function, which means that it can be written as $\sum_{n=0}^{\infty} b_n c^n$ for $b_n \geq 0$ (Daniely et al., 2016; Martens et al., 2021). Note that for smooth activation functions, positive definiteness can be easily verified by Taylor-expanding $\mathcal{C}(c)$ about $c = 0$ and using

$$\mathcal{C}^{(i)}(0) = \mathbb{E}_{z\sim\mathcal{N}(0,1)}\left[\phi^{(i)}(z)\right]^2 \geq 0. \quad (19)$$

As discussed in Section 2.3, global C maps are computed by recursively taking compositions and weighted averages (with non-negative weights), starting from $\mathcal{C}$. Because all of the above properties are preserved under these operations, it follows that global C maps inherit them from $\mathcal{C}$.

B TECHNICAL RESULTS AND PROOFS

Lemma 1. *For networks using the activation function $\tilde{\phi}_\alpha(x) = \sqrt{\frac{2}{1+\alpha^2}}\phi_\alpha(x)$, the local Q and C maps are given by*

$$\mathcal{Q}(q) = q \quad \textit{and} \quad \mathcal{C}(c) = c + \frac{(1-\alpha)^2}{\pi(1+\alpha^2)}\left(\sqrt{1-c^2} - c\cos^{-1}(c)\right). \tag{20}$$

Proof. In this proof we will use the notation $\mathcal{Q}_\phi$ and $\mathcal{C}_\phi$ to denote the local Q and C maps for networks that use a given activation function ϕ.

First, we note that LReLU is basically the weighted sum of identity and ReLU. In particular, we have the following equation:

$$\phi_\alpha(x) = \alpha x + (1-\alpha)\phi_0(x) = \max\{x, 0\} + \alpha\min\{x, 0\}.$$

Second, it is easy to show that $\mathcal{Q}_{\phi_\alpha}(q) = \frac{1+\alpha^2}{2}q$ (from which $\mathcal{Q}(q)_{\tilde{\phi}_\alpha} = q$ immediately follows).

It then follows from equation 5, and the fact that local C maps are invariant to multiplication of the activation function by a constant, that

$$\begin{aligned}\mathcal{C}_{\tilde{\phi}_\alpha}(c) = \mathcal{C}_{\phi_\alpha}(c) &= \frac{2}{1+\alpha^2}\mathbb{E}_{z_1,z_2\sim\mathcal{N}(0,1)}\left[\phi_\alpha(z_1)\,\phi_\alpha\left(cz_1 + \sqrt{1-c^2}z_2\right)\right] \\ &= \frac{2}{1+\alpha^2}\left[\alpha^2 c + (1-\alpha)^2\mathcal{C}_{\phi_0}(c)Q_{\phi_0}(1)\right] \\ &\quad + \frac{2}{1+\alpha^2}\left[2\alpha(1-\alpha)\mathbb{E}_{z_1,z_2\sim\mathcal{N}(0,1)}\left[(cz_1 + \sqrt{1-c^2}z_2)\phi_0(z_1)\right]\right]\end{aligned} \tag{21}$$

From Daniely et al. (2016) we have that

$$\mathcal{C}_{\phi_0}(c) = \frac{\sqrt{1-c^2} + (\pi - \cos^{-1}(c))c}{\pi}, \tag{22}$$

and for the last part of equation 21 we have

$$\mathbb{E}_{z_1,z_2\sim\mathcal{N}(0,1)}\left[(cz_1 + \sqrt{1-c^2}z_2)\phi_0(z_1)\right] = \mathbb{E}_{z_1\sim\mathcal{N}(0,1)}\left[cz_1^2\mathbb{1}_{z_1>0}\right] = \frac{c}{2}. \tag{23}$$

Plugging equation 22 and equation 23 back into equation 21, we get

$$\begin{aligned}\mathcal{C}_{\tilde{\phi}_\alpha}(c) &= \frac{2}{1+\alpha^2}\left[\alpha^2 c + \frac{(1-\alpha)^2}{2}\frac{\sqrt{1-c^2} + (\pi - \cos^{-1}(c))c}{\pi} + \alpha(1-\alpha)c\right] \\ &= \frac{(1-\alpha)^2\left(\sqrt{1-c^2} + c(\pi - \cos^{-1}(c))\right) + 2\pi\alpha c}{(1+\alpha^2)\pi}.\end{aligned} \tag{24}$$

Rearranging this gives the claimed formula. □

Proposition 1. *The global C map of a feedforward network with $\tilde{\phi}_\alpha(x)$ as its activation function is equal to that of a rescaled ResNet of the same depth (see Section 2.4) with normalized ReLU activation $\phi(x) = \sqrt{2}\max(x, 0)$, shortcut weight $\sqrt{\frac{\alpha}{1+\alpha^2}}$, and residual branch $\mathcal{R}$ consisting of a combined layer (or just a normalized ReLU activation) followed by an affine layer.*

Proof. By equation 7, the C map for a residual block $\mathcal{B}$ of the hypothesized rescaled ResNet is given by

$$\mathcal{C}_{\mathcal{B}}(c) = w^2 c + (1-w^2)\mathcal{C}_{\phi_0}(c). \tag{25}$$

The global C map of this network is given by L compositions of this function, while the global C map of the hypothesized feedforward network is given by L compositions of $\mathcal{C}_{\tilde{\phi}_\alpha}(c)$. So to prove the claim it suffices to show that $\mathcal{C}_{\mathcal{B}}(c) = \mathcal{C}_{\tilde{\phi}_\alpha}(c)$.

Taking $w = \sqrt{\frac{2\alpha}{1+\alpha^2}}$, one obtains the following

$$\mathcal{C}_{\mathcal{B}}(c) = \frac{2\alpha}{1+\alpha^2} + \frac{(1-\alpha^2)}{1+\alpha^2}\frac{\sqrt{1-c^2} + c(\pi - \cos^{-1}(c))}{\pi}, \tag{26}$$

which is exactly the same as $\mathcal{C}_{\tilde{\phi}_\alpha}(c)$ as given in Lemma 1. This concludes the proof. □

Proposition 2. *For a vanilla network f consisting of L combined layers with the* `TReLU` *activation function, if we choose a fixed $\mathcal{C}_f^L(0) = \eta \in (0,1)$, then the global map $\mathcal{C}_f$ of the network converges to a limiting map on $(-1,1)$ as L goes to infinity. In particular,*

$$\lim_{L\to\infty} \mathcal{C}_f^L(c) = \psi(c,T) \tag{27}$$

where T is such that $\psi(0,T) = \eta$, and where ψ is the solution of the following ordinary differential equation (ODE) with the first argument is the initial condition (i.e. $\psi(c,0) = c$), and the second argument is time:

$$\frac{dx(t)}{dt} = \sqrt{1-x(t)^2} - x(t)\cos^{-1}(x(t)). \tag{28}$$

Proof. First, we notice that the local C map of a `TReLU` combined layer can be written as a difference equation:

$$\mathcal{C}(c) = c + \frac{(1-\alpha)^2}{\pi(1+\alpha^2)}\left(\sqrt{1-c^2} - c\cos^{-1}(c)\right). \tag{29}$$

Importantly, this C map is a monotonically increasing function and the speed of increasing is inversely associated with the value of $\alpha \in [0,1]$. So in order to guarantee $\mathcal{C}_f^L(0) = \eta$, the "step size" $\frac{(1-\alpha)^2}{\pi(1+\alpha^2)}$ has to be inversely proportional to L and goes to 0 as $L \to \infty$. In this case, the difference equation converges to the ODE in equation 28.

Because the function $\sqrt{1-x^2} - x\cos^{-1}(x)$ is continuously differentiable in $[-1,1]$ and its derivative with the form of $-\cos^{-1}(x)$ is bounded, one can immediately show that it is globally Lipschitz and the ODE has a unique solution $\psi(c_0,t)$ according to Theorem 3.2 of Khalil (2008). Now, we are only left to find the time T such that $\mathcal{C}_f^\infty(0) = \psi(0,T) = \eta$. To that end, we notice that

$$g(x) = \sqrt{1-x^2} - x\cos^{-1}(x) > 0, ; \text{for } x \in (-1,1) \tag{30}$$

because $g(1) = 0$ and $g'(x) = -\cos^{-1}(x) < 0$ on $(-1,1)$. This implies that the $\psi(0,t)$ is a monotonically increasing function of t. Besides, it has been shown by Hayou et al. (2019) that the c value of LReLU networks with a fixed negative slope α converges to 1 as $L \to \infty$, suggesting $\psi(0,\infty) = 1$ (we also verified that numerically), hence T could be uniquely decided as long as $\eta \in (0,1)$. This completes the proof. □

Theorem 1. *For a network f with $\tilde{\phi}_\alpha(x)$ as its activation function (for $\alpha \geq 0$), we have*

$$\max_{c\in[-1,1]} |\mathcal{C}_f(c) - c| \leq \min\{4\mathcal{C}_f(0), 1 + \mathcal{C}_f(0)\}, \quad \max_{c\in[-1,1]} \left|\mathcal{C}_f'(c) - 1\right| \leq \min\{4\mathcal{C}_f(0), 1\} \tag{10}$$

Proof. Because $\mathcal{C}_f$ is a positive definite function (by Section A.3) we have that it can be written as $\mathcal{C}_f(c) = \sum_n^\infty b_n c^n$ for $b_n \geq 0$. Given $\mathcal{C}_f(1) = \mathcal{C}_f'(1) = 1$, we have

$$\sum_{n=0}^\infty b_n = \sum_{n=1}^\infty n b_n = 1 \quad \Longrightarrow \quad b_0 = \sum_{n=2}^\infty (n-1) b_n \quad \Longrightarrow \quad 2b_0 + b_1 \geq 1. \tag{31}$$

Hence, $1 - \mathcal{C}_f'(0) = 1 - b_1 \leq 2b_0 = 2\mathcal{C}_f(0)$. Now we are ready to bound the deviation of $\mathcal{C}_f(c)$ from identity:

$$\begin{aligned}
\max_{c\in[-1,1]} |\mathcal{C}_f(c) - c| &= \max_{c\in[-1,1]} \left| b_0 + \sum_{n=2}^{\infty} b_n c^n - (1-b_1)c \right| \\
&\leq \max_{c\in[-1,1]} \left[b_0 + \sum_{n=2}^{\infty} b_n |c|^n + (1-b_1)|c| \right] \\
&= b_0 + \sum_{n=2}^{\infty} b_n + 1 - b_1 = 2(1-b_1) \\
&= 2(1 - \mathcal{C}_f'(0)) \leq 4\mathcal{C}_f(0).
\end{aligned} \tag{32}$$

Using equation 20 we have that

$$\mathcal{C}'(c) = 1 - \frac{(1-\alpha)^2}{(1+\alpha^2)\pi} \cos^{-1}(c).$$

From our assumption that $\alpha \geq 0$ it follows that $0 \leq \mathcal{C}'(c) \leq 1$ for all $c \in [-1, 1]$. Since the property of having a derivative bounded between 0 and 1 is closed under functional composition and positive weighted averages, it thus follows that $0 \leq \mathcal{C}_f(c) \leq 1$ for all $c \in [-1, 1]$. An immediate consequence of this is that $\mathcal{C}_f(c)$ is non-decreasing, and that

$$\max_{c\in[-1,1]} |\mathcal{C}_f(c) - c| = \mathcal{C}_f(-1) + 1 \leq \mathcal{C}_f(0) + 1. \tag{33}$$

Next, we bound the deviation of $\mathcal{C}_f'(c)$ from 1:

$$\begin{aligned}
\max_{c\in[-1,1]} \left|\mathcal{C}_f'(c) - 1\right| &= \max_{c\in[-1,1]} \left| \sum_{n=2}^{\infty} n b_n c^{n-1} - (1-b_1) \right| \\
&\leq \max_{c\in[-1,1]} \left[\sum_{n=2}^{\infty} n b_n |c|^{n-1} + (1-b_1) \right] \\
&= \sum_{n=2}^{\infty} n b_n + 1 - b_1 = 2(1-b_1) \\
&= 2(1 - \mathcal{C}_f'(0)) \leq 4\mathcal{C}_f(0).
\end{aligned} \tag{34}$$

From the previous fact that $0 \leq \mathcal{C}'(c) \leq 1$ for all $c \in [-1, 1]$ we also have that $\max_{c\in[-1,1]} \left|\mathcal{C}_f'(c) - 1\right| \leq 1$. This completes the proof. □

Theorem 2. *Suppose f is a network with a smooth activation function. If $\mathcal{C}_f'(1) = 1$, then we have*

$$\max_{c\in[-1,1]} |\mathcal{C}_f(c) - c| \leq 2\mathcal{C}_f''(1), \quad \max_{c\in[-1,1]} \left|\mathcal{C}_f'(c) - 1\right| \leq 2\mathcal{C}_f''(1) \tag{13}$$

Proof. $\mathcal{C}_f$ is a positive definite function by Section A.3. So by the fact that positive definite functions are non-negative, non-decreasing, and convex on the non-negative part of their domain, we obtain that $\mathcal{C}_f'(0) \geq \mathcal{C}_f'(1) - C_f''(1) = 1 - C_f''(1)$. By equation 32, we have

$$\max_{c\in[-1,1]} |\mathcal{C}_f(c) - c| \leq 2(1 - \mathcal{C}_f'(0)) \leq 2\mathcal{C}_f''(1). \tag{35}$$

Further by equation 34, we also have

$$\max_{c\in[-1,1]} \left|\mathcal{C}_f'(c) - 1\right| \leq 2(1 - \mathcal{C}_f'(0)) \leq 2\mathcal{C}_f''(1). \tag{36}$$

This completes the proof. □

Algorithm 1 K-FAC. Subscript l denotes layers, $\mathbf{w}_l = \mathrm{vec}(\mathbf{W}_l)$. In this algorithm box, we assume zero momentum (in practice we use a momentum value of 0.9) and skip the trust region learning rate rescaling (in practice we use trust region norm constraint 0.001).

Require: α, β: stepsize, exponential moving average parameter (default: 0.99)
Require: γ, λ : damping term (default: 0.001), L_2 regularization coefficient
Require: stats and inverse update intervals T_{stats} and T_{inv}
  $k \leftarrow 0$ and initialize $\{\mathbf{S}_l\}_{l=1}^{L}, \{\mathbf{A}_l\}_{l=0}^{L-1}$
  while stopping criterion not met **do**
    $k \leftarrow k + 1$
    if $k \equiv 0 \pmod{T_{\text{stats}}}$ **then**
      Update the factors $\{\mathbf{S}_l\}_{l=1}^{L}, \{\mathbf{A}_l\}_{l=0}^{L-1}$ using exponential moving averages with parameter β
    end if
    if $k \equiv 0 \pmod{T_{\text{inv}}}$ **then**
      Calculate the inverses $\{[\mathbf{S}_l^{\gamma}]^{-1}\}_{l=1}^{L}, \{[\mathbf{A}_l^{\gamma}]^{-1}\}_{l=0}^{L-1}$.
    end if
    $\mathbf{V}_l = \nabla_{\mathbf{W}_l} \log p(\mathbf{y}|\mathbf{x}, \mathbf{w}) - \lambda \cdot \mathbf{W}_l$
    $\mathbf{W}_l \leftarrow \mathbf{W}_l + \alpha[\mathbf{A}_l^{\gamma}]^{-1}\mathbf{V}_l[\mathbf{S}_l^{\gamma}]^{-1}$
  end while

Proposition 3. *Suppose f is some function computed by a neural network with the ReLU activation. Then for any negative slope parameter $\alpha \neq \pm 1$, we can compute f using an LReLU neural network of the same structure and double the width of the original network.*

Proof. The basic intuition behind this proof is that a ReLU unit can always be "simulated" by two LReLU units as long as $\alpha \neq \pm 1$, due to the following formula:

$$\phi_0(x) = \frac{1}{1-\alpha^2}\left(\phi_\alpha(x) + \alpha\phi_\alpha(-x)\right).$$

We will begin by proving the claim in the case of a network with one hidden layer. In particular, we assume the ReLU network has m hidden units:

$$f(w, b, a, x) = \sum_{r=1}^{m} a_r \phi_0(w_r^\top x + b_r), \tag{37}$$

where $x \in \mathbb{R}^d$ is the input, and $w \in \mathbb{R}^{md}$, $b \in \mathbb{R}^m$ and $a \in \mathbb{R}^m$ are weights, biases of the input layer and weights of output layer, respectively. For LReLU with negative slope α, one can construct the following network

$$f(w', b', a', x) = \sum_{r=1}^{2m} a'_r \phi_\alpha({w'_r}^\top x + b'_r). \tag{38}$$

If we choose $w'_r = w_r = -w'_{r+m}$, $b'_r = b_r = -b'_{r+m}$, $a'_r = \frac{1}{1-\alpha^2}a_r$ and $a'_{r+m} = \frac{\alpha}{1-\alpha^2}a_r$, we have

$$\begin{aligned} &a'_r\phi_\alpha({w'_r}^\top x + b'_r) + a'_{r+m}\phi_\alpha({w'_{r+m}}^\top x + b'_{r+m}) \\ &\qquad = \frac{1}{1-\alpha^2}a_r\phi_\alpha(w_r^\top x + b_r) - \frac{\alpha^2}{1-\alpha^2}a_r\phi_{\frac{1}{\alpha}}(w_r^\top x + b_r) = a_r\phi(w_r^\top x + b_r), \end{aligned} \tag{39}$$

This immediately suggests that $f(w', b', a', x) = f(w, b, a, x)$.

Since deeper networks and one with more complex topologies can be constructed by composing and summing shallower ones, the general claim follows. □

C Implementation Details

For data preprocessing we perform a random crop of size 224×224 to each image and apply a random horizontal flip. In all experiments, we only applied L_2 regularization to weights (both biases and batch normalization parameters are excluded). In addition, we used label smoothing (Szegedy

et al., 2017) with a value of 0.1. For networks using batch normalization, we set a decay rate value of 0.9. In all optimization algorithms we set the momentum value to 0.9 and tune the initial learning rate, L_2 regularization value and dropout value with grid searches. The L_2 value is selected from $\{0.00005, 0.00002, 0.0\}$ and the dropout rate from $\{0.2, 0.0\}$ respectively. We emphasis that dropout is only used for networks without batch normalization. For K-FAC, we used a fixed damping value of 0.001 and a norm constraint value of 0.001. We updated the Fisher matrix approximation every iteration and compute the inverse every 50 iterations (i.e., $T_{\text{stats}} = 1$ and $T_{\text{inv}} = 50$ in Algorithm 1), unless stated otherwise. For LARS, we set the trust region coefficient of 0.001. The best learning rate for SGD was selected from $\{1.0, 0.3, 0.1, 0.03, 0.01\}$, for K-FAC from $\{0.003, 0.001, 0.0003, 0.0001, 0.00003\}$ and for LARS from $\{10.0, 3.0, 1.0, 0.3, 0.1\}$. If all of the runs fail for any of the optimizers, we select a smaller learning rate. Throughout training we used a standard learning rate warmup scheme which linearly increases from 0 to the initial learning rate in the first 5 epochs, and then decays the learning rate by a factor of 10 at $4/9$ and $7/9$ of the total epoch budget[5], unless specified otherwise. All of the models, are implemented with `JAX` (Bradbury et al., 2018) and `Haiku` (Hennigan et al., 2020). We used the implementation in `Optax` (Hessel et al., 2020) for both SGD and LARS.

For initialization of the weights by default we used the scale-corrected uniform orthogonal (SUO) distribution (Martens et al., 2021). For a $m \times k$ matrix, samples from this distribution can be generated by computing $\left(XX^\top\right)^{-1/2} X$, where X is an $m \times k$ matrix with entries sampled independently from $\mathcal{N}(0, 1)$. When $m > k$, we may apply the same procedure but with k and m reversed, and then transpose the result. The resulting distribution is given by the well-known Haar measure on orthogonal matrices (e.g. Meckes (2019)), and is also sometimes called the uniform distribution.. The resulting sample is further multiplied by the scaling factor $\max\{\sqrt{m/k}, 1\}$, which will have an effect only when $k \leq m$. For convolutional networks, we combine this procedure with the Delta initializations (Balduzzi et al., 2017; Xiao et al., 2018), which we refer to as Orthogonal Delta initializations.

D Additional Results

D.1 Results on CIFAR-10

In addition to our main results on the ImageNet dataset, we also compare our method to EOC on the CIFAR-10 (Krizhevsky et al., 2009) dataset with a rescaled Wide ResNets (Zagoruyko & Komodakis, 2016). In particular, we use a widening factor of 2 and remove all batch normalization layers and shortcut connections. We train both networks with the K-FAC optimizer for 200 epochs and use a standard piecewise constant learning rate schedule. To be specific, we decay the learning rate by a factor of 10 at 75 and 150 epochs. For K-FAC, we set the damping value to 0.01 and norm constraint value to 0.0001. For data preprocessing we include basic data augmentation such as random crop and horizontal flip during training. As shown in Figure 4, our method outperforms EOC significantly. As we increase the depth from 100 to 304, the accuracy of EOC network drops dramatically while the accuracy of the network constructed using TAT remains roughly unchanged.

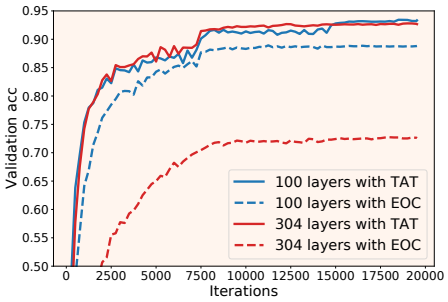

Figure 4: CIFAR validation accuracy with ReLU activation function initialized using either EOC or TAT (ours).

D.2 Reducing K-FAC overhead

With the learning rate schedule and fixed momentum value used in our experiments K-FAC wall-clock time is roughly $2.5\times$ than that of SGD. However, this gap can be decreased significantly by reducing the frequency of the updates of the approximate curvature matrix and its inverses. For example if we

[5]We later found that cosine learning rate annealing (Loshchilov & Hutter, 2016) is slightly better for most settings, but this did not change our conclusions.

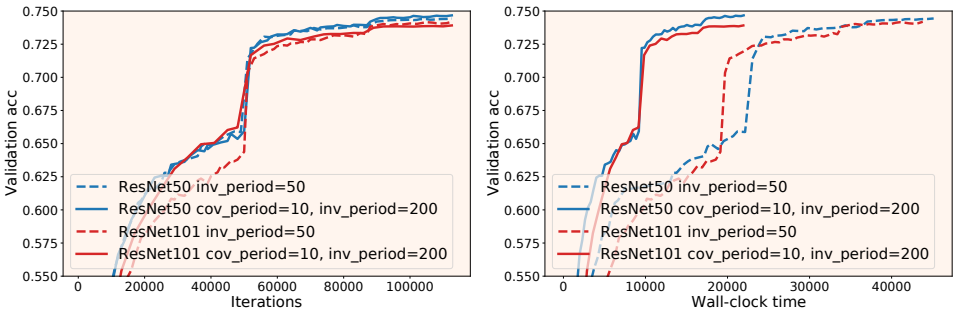

Figure 5: Top-1 validation accuracy as a function of number of iterations (**left**) or wall-clock time (**right**) with K-FAC optimizer. One can reduce the computational overhead significantly by update the Fisher covariance blocks and inverse less frequently.

update the curvature approximation every 10 steps and the inverses every 200 steps the wall-clock time of K-FAC reduces by half to a mere $1.25\times$ that of SGD. Importantly, as can be seen on Figure 5, this does not appear to affect per-iteration optimization performance significantly.

D.3 Disentangling Training and Generalization

So far, we only reported validation accuracies on ImageNet, so it is hard to tell whether the poor performance of EOC-initialized networks is due to underfitting or overfitting. Here, we compare training accuracies of EOC-initialized networks (with ReLU) and networks with `TReLU` with exactly the same setting as Figure 1. We train each network on ImageNet with 90 epochs using K-FAC. For each setting, we report the training accuracy of the model with highest validation accuracy over all sets of hyperparameters. As shown in Figure 6, the EOC-initialized networks achieve competitive (if not any better) training performance, suggesting that the use of `TReLU` improves the generalization performance rather than training speed.

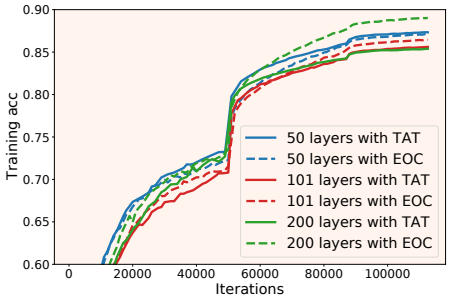

Figure 6: Top-1 ImageNet training accuracy of deep vanilla networks with ReLU activation initialized using either EOC or TAT (ours).

D.4 Closing the Remaining Gap with Wider Networks

For all the experiments we've done, we used networks with the same width as standard ResNets. A natural question to ask is how do wider networks perform. It is possible that our method would benefit more from the increasing width, since it makes the kernel approximation more accurate. With wider neural networks, it becomes too expensive to store and invert Kronecker factors in K-FAC. Therefore, we only train wider neural networks with SGD. In order to mitigate non-convergent issues of SGD (see section 5.5), we train vanilla networks with 360 epochs and a batch size of 512. For fair comparison, we train standard ResNets with both 90 epochs and 360 epochs, and report the higher numbers.

Table 9: The effect of increasing width. We use vanilla networks for EOC and TAT (ours).

Network	Width	EOC	TAT	ResNets
ResNet50	$1\times$	72.0	76.0	76.7
	$2\times$	73.5	77.3	77.9
ResNet101	$1\times$	62.4	76.5	77.9
	$2\times$	66.5	77.6	78.6

As shown in Table 9, we observe that doubling the width indeed further closes the remaining gap. For standard ResNets, the improvements are relatively small after increasing the width, whereas we obtain more than 1% gain in validation accuracy with our method. The EOC method also benefits from the increasing width, but that's likely due to the use of SGD (after doubling the width, the EOC accuracies still fall short of the results in Table 5 with K-FAC optimizer.).

D.5 Comparison with PReLU on Rescaled ResNets

In Table 6, we report the comparison between PReLU and TReLU on deep vanilla networks. Here we further compare our TReLU to PReLU on rescaled ResNets with shortcut weight $w = 0.8$. For PReLU, we again include two different initializations: one with 0 negative slope (effectively ReLU) and another with 0.25 negative slope (this was used in He et al. (2015)). We report the full results in Table 10. For all the settings, our method once again outperforms PReLU by a big margin, suggesting a better-initialized negative slope could also be crucial for rescaled ResNets.

Table 10: Comparison with PReLU (last two columns) with rescaled ResNets ($w = 0.8$).

Network	Optimizer	TReLU	0.0	0.25
ResNet50	K-FAC	76.4	75.7	73.6
	SGD	76.0	71.5	71.5
ResNet101	K-FAC	77.8	76.4	76.8
	SGD	77.3	73.1	73.4

D.6 Comparison of different initializations

For all the results we have reported so far, we used orthogonal delta initialization introduced in Xiao et al. (2018). A natural question to ask is whether this initialization is better than the standard initialization which draws samples from Gaussian distribution. Here we conduct a head-to-head comparison for all different architectures including standard ResNet, normalizer-free shortcut-free networks with either EOC or TAT, with results reported in Table 11. For all the networks, we used ReLU activation except TAT for which it is essentially LReLU with adjusted negative slope. Surprisingly, it turns out that orthogonal delta initialization does not have any obvious advantage over standard Gaussian initialization.

Table 11: Ablation study on comparing orthogonal delta and He initialization (sampled from Gaussian).

Network	Optimizer	Init	ResNet	EOC	TAT
ResNet50	K-FAC	Orth Delta	76.4	72.6	74.6
		Gaussian	76.5	72.5	74.8
	SGD	Orth Delta	76.3	63.7	71.0
		Gaussian	76.6	63.1	68.7
ResNet101	K-FAC	Orth Delta	77.8	71.8	74.2
		Gaussian	77.8	72.3	74.1
	SGD	Orth Delta	77.9	41.6	70.0
		Gaussian	78.0	41.1	68.7

E Code for activation transformation

E.1 Code for LReLU

```
import jax.numpy as np
import jax
import math
import functools

CONFIGS = {
      50: {
          "blocks_per_group": (3, 4, 6, 3),
          "bottleneck": True,
          "channels_per_group": (256, 512, 1024, 2048),
          "use_projection": (True, True, True, True),
      },
      101: {
          "blocks_per_group": (3, 4, 23, 3),
          "bottleneck": True,
          "channels_per_group": (256, 512, 1024, 2048),
          "use_projection": (True, True, True, True),
      },
      152: {
          "blocks_per_group": (3, 8, 36, 3),
          "bottleneck": True,
          "channels_per_group": (256, 512, 1024, 2048),
          "use_projection": (True, True, True, True),
      },
      200: {
```

```
          "blocks_per_group": (3, 24, 36, 3),
          "bottleneck": True,
          "channels_per_group": (256, 512, 1024, 2048),
          "use_projection": (True, True, True, True),
      },
}

def lrelu_kernel(c, alpha=0.0):
  return ((1 - alpha) ** 2 *
          (np.sqrt(1 - c ** 2) + (math.pi - np.arccos(c)) * c) / math.pi
          + 2 * alpha * c) / (1 + alpha ** 2)

def global_cmap_fn(local_cmap_fn, depth, c_init=0.0, shortcut_weight=0.0):
    blocks_per_group = CONFIGS[depth]["blocks_per_group"]
    bottleneck = CONFIGS[depth]["bottleneck"]
    use_projection = CONFIGS[depth]["use_projection"]
    c = c_init
    for i in range(4):
      for j in range(blocks_per_group[i]):
        if bottleneck:
          main_c = local_cmap_fn(local_cmap_fn(local_cmap_fn(c)))
        else:
          main_c = local_cmap_fn(local_cmap_fn(c))
        res_c = local_cmap_fn(c) if (j == 0 and use_projection[i]) else c
        c = shortcut_weight**2 * res_c + (1.0 - shortcut_weight**2) * main_c
    return local_cmap_fn(c)

def binary_search(fn, target, input_=0.0, min_=-1.0, max_=1.0, tol=1e-6):
  value = fn(input_)

  if np.abs(value - target) < tol:
    return input_

  if value < target:
    new_input = 0.5 * (input_ + min_)
    max_ = input_
  elif value > target:
    if np.isinf(max_):
      new_input = input_ * 2
    else:
      new_input = 0.5 * (input_ + max_)
    min_ = input_

  return binary_search(fn, target, new_input, min_, max_, tol=tol)

# the key function getting transformed activation
def get_transformed_lrelu(depth, shortcut_weight, target_value):
  global_lrelu_cmap_fn = lambda alpha: global_cmap_fn(
      functools.partial(lrelu_kernel, alpha=alpha),
      depth, shortcut_weight=shortcut_weight)

  negative_slope = binary_search(global_lrelu_cmap_fn, target_value)
  transformed_lrelu = (lambda x: math.sqrt(2.0 / (1 + negative_slope**2)) *
                       jax.nn.leaky_relu(x, negative_slope=negative_slope))
  return transformed_lrelu
```

E.2 Code for other smooth activations

```
import jax
import scipy as osp
from autograd import elementwise_grad as egrad
from autograd import numpy as np
from autograd import scipy as sp
import scipy.integrate as sp_int
```

```
from scipy.integrate.quadrature import _cached_roots_legendre
import scipy.optimize as sp_opt

SELU_LAMBDA = 1.0507
SELU_ALPHA = 1.67326
STEP = 0.0005
NUM_POINTS = 100000
SIGMOID = sp.special.expit
roots = osp.special.roots_legendre(NUM_POINTS)

def swish(x):
  return x * SIGMOID(x)


def swish_der(x):
  return SIGMOID(x) * (1. + x * (1 - SIGMOID(x)))


def swish_der2(x):
  return SIGMOID(x) * (1. - SIGMOID(x)) * (2. + x * (1. - 2. * SIGMOID(x)))


def safe_softplus(x):
  """Numerically-stable softplus.
  """
  return np.log(1. + np.exp(-np.abs(x))) + np.maximum(x, 0)


# Definitions for basic activation functions and their derivatives:
# (could compute the latter using `egrad` probably)
NONLINEARITIES = {
    "tanh": {
        "fn": np.tanh,
        "der": lambda x: 1. - (np.tanh(x)**2),
        "curv": egrad(egrad(np.tanh)),
    },
    "selu": {
        "fn": lambda x: elu(x, SELU_ALPHA, SELU_LAMBDA),
        "der": lambda x: elu_der(x, SELU_ALPHA, SELU_LAMBDA),
        "curv": lambda x: elu_curv(x, SELU_ALPHA, SELU_LAMBDA),
    },
    "softplus": {
        "fn": safe_softplus,
        "der": SIGMOID,
        "curv": egrad(SIGMOID),
    },
    "swish": {
        "fn": lambda x: x * SIGMOID(x),
        "der": swish_der,
        "curv": egrad(swish_der),
    },
}

class ParameterizedNonlinearity(object):
  """A class for determining nonlinearity parameters."""

  def __init__(self, nonlin_str, input_scale, input_shift,
               output_shift, output_scale=1.0,):

    self.nonlin_str = nonlin_str

    nl = NONLINEARITIES[nonlin_str]
    self.phi, self.phi_der, self.phi_curv = nl["fn"], nl["der"], nl["curv"]

    self.params = {}
```

```
    self.params["input_scale"] = input_scale
    self.params["input_shift"] = input_shift
    self.params["output_scale"] = output_scale
    self.params["output_shift"] = output_shift

    # first output scale
    q = _calc_output_q(self)
    self.params["output_scale"] = np.sqrt(1. / q) * self.params["output_scale"]

    # curv1 is the second derivative the C map at c=1
    self.curv1 = _calc_curv1(self, q_output=1.0)

    # chi0 is the slope the C map at c=0
    self.chi0 = _calc_chi(self, target="chi0", q_output=1.0)

    # chi1 is the slope the C map at c=1
    self.chi1 = _calc_chi(self, target="chi1", q_output=1.0)

    self.q_output = _calc_output_q(self)

    # qslope1 is the slope the Q map at c=1
    self.qslope1 = _calc_qslope1(self)

  def fn(self, x):
    b = self.params["input_shift"]
    f = self.params["output_shift"]
    g = self.params["output_scale"]
    s = self.params["input_scale"]

    return g * (self.phi(s * x + b) + f)

  def der(self, x):
    b = self.params["input_shift"]
    g = self.params["output_scale"]
    s = self.params["input_scale"]
    return g * (self.phi_der(s * x + b)) * s

  def curv(self, x):
    b = self.params["input_shift"]
    g = self.params["output_scale"]
    s = self.params["input_scale"]
    return g * (self.phi_curv(s * x + b)) * s ** 2

  @property
  def chi_ratio(self):
    return self.chi1 / self.chi0


def _estimate_gaussian_mean(fn, n=NUM_POINTS):
  """Estimate the mean of a function fn(x) where x ~ N(0,1)."""
  _cached_roots_legendre.cache[n] = roots
  fn_weighted = lambda x: np.exp(-x**2 / 2) * fn(x)
  integral, _ = sp_int.fixed_quad(fn_weighted, -10., 10., n=n)

  return integral / np.sqrt(2 * np.pi)


def _calc_chi(nl, target="chi1", q_output=None):
  # Estimate result using numerical integration:

  if target == "chi0":
    d_int = _estimate_gaussian_mean(nl.der)**2
  elif target == "chi1":
    fn = lambda x: nl.der(x)**2
    d_int = _estimate_gaussian_mean(fn)
```

```
  if q_output is None:
    q_output = _calc_output_q(nl)

  chi = d_int / q_output

  return chi


def _calc_curv1(nl, q_output=None):
  fn = lambda x: nl.curv(x)**2
  int_ = _estimate_gaussian_mean(fn)
  if q_output is None:
    q_output = _calc_output_q(nl)

  return int_ / q_output


def _calc_qslope1(nl):
  # this assumes qin = 1

  fn = lambda x: nl.fn(x) * nl.der(x) * x
  int_ = _estimate_gaussian_mean(fn)
  return int_


def _calc_output_q(nl):
  fn = lambda x: nl.fn(x)**2
  int_ = _estimate_gaussian_mean(fn)
  return int_


def _eval_nonlin_properties(nonlin_str, input_scale, input_shift, output_shift):

  return ParameterizedNonlinearity(
      nonlin_str, input_scale=input_scale,
      input_shift=input_shift, output_shift=output_shift)

def _match_x_match_y_match_z(nonlin_str,
                             x_target,
                             y_target,
                             z_target,
                             x_string,
                             y_string,
                             z_string,
                             method="hybr",
                             options=None,
                             starting_point=(1.0, -0.5, 0.0)):

  # We are searching over input_scale and input_shift here.

  def func(v):
    input_scale = v[0]
    input_shift = v[1]
    output_shift = v[2]

    nl_prop = _eval_nonlin_properties(nonlin_str,
                                      input_scale=input_scale,
                                      input_shift=input_shift,
                                      output_shift=output_shift)

    return np.asarray([
        getattr(nl_prop, x_string) - x_target,
        getattr(nl_prop, y_string) - y_target,
        getattr(nl_prop, z_string) - z_target,
```

```
    ])

  sol = sp_opt.root(
      func,
      np.asarray(starting_point),
      method=method,
      jac=False,
      options=options)
  input_scale_match = sol.x[0]
  input_shift_match = sol.x[1]
  output_shift_match = sol.x[2]

  if not sol.success:
    raise ValueError("Root finding failed for given arguments: "
                     "nonlin_str={}, x_target={}, x_string={}, "
                     "y_target={}, y_string={}, z_target={}, z_string={}."
                     "".format(nonlin_str, x_target, x_string, y_target,
                               y_string, z_target, z_string))

  return _eval_nonlin_properties(
      nonlin_str,
      input_scale=input_scale_match,
      input_shift=input_shift_match,
      output_shift=output_shift_match)


def compute_nonlinearity_properties(nonlinearity, per_nl_curv1):

  always_try = (
      (1.0, 0.0, 0.0),
      (1.0, 1.0, 0.0),
      (1.0, -1.0, 0.0),
      (1.0, 0.0, 1.0),
      (1.0, 1.0, 1.0),
      (1.0, -1.0, 1.0),
      (1.0, 0.0, -1.0),
      (1.0, 1.0, -1.0),
      (1.0, -1.0, -1.0),
      (0.1, 0.0, 0.0),
      (0.1, 1.0, 0.0),
      (0.1, -1.0, 0.0),
      (0.1, 0.0, 1.0),
      (0.1, 1.0, 1.0),
      (0.1, -1.0, 1.0),
      (0.1, 0.0, -1.0),
      (0.1, 1.0, -1.0),
      (0.1, -1.0, -1.0),
  )
  nl_prop = None

  for i in range(50):

    if i < len(always_try):
      starting_point = always_try[i]
    else:
      starting_point = (np.random.uniform(low=0.0, high=2.0),
                        np.random.uniform(low=-3.0, high=3.0),
                        np.random.uniform(low=-3.0, high=3.0))
    try:
      nl_prop = _match_x_match_y_match_z(
          nonlinearity,
          1.0,
          1.0,
          per_nl_curv1,
          "chi1",
```

```
          "qslope1",
          "curv1",
          method="hybr",
          starting_point=starting_point)

      print("Found parameters for nonlinearity {} using starting "
            "point {}.".format(nonlinearity, starting_point))
      if nl_prop.params["input_scale"] * nl_prop.params["output_scale"] > 2.0:
        print("The solution is not good enough. Keep searching.")
        continue
      break
    except ValueError:
      print("Failed to find parameters for nonlinearity {} using "
            "starting point {}.".format(nonlinearity, starting_point))

  if nl_prop is None:
    raise ValueError("Failed to find parameters for "
                     "nonlinearity {}.".format(nonlinearity))

  return nl_prop.params

def get_transformed_activation(act_fn, depth, shortcut_weight, target_value):
  w = shortcut_weight**2
  per_nl_curv1 = target_value / (5 + (1 - w) * (depth - 6))
  params = compute_nonlinearity_properties(act_fn, per_nl_curv1)
  # transforming the activation
  if act_fn == "tanh":
    activation = jnp.tanh
  else:
    activation = getattr(jax.nn, act_fn)
  transformed_activation = (lambda x: params["output_scale"] *
                            (activation(params["input_scale"] * x +
                                        params["input_shift"]) +
                             params["output_shift"]))
```