# FAST FINITE WIDTH NEURAL TANGENT KERNEL

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

The Neural Tangent Kernel (NTK), defined as the outer product of the neural network (NN) Jacobians, $\Theta_\theta(x_1, x_2) = \left[\partial f(\theta, x_1)/\partial\theta\right]\left[\partial f(\theta, x_2)/\partial\theta\right]^T$, has emerged as a central object of study in deep learning. In the infinite width limit, the NTK can sometimes be computed analytically and is useful for understanding training and generalization of NN architectures. At finite widths, the NTK is also used to better initialize NNs, compare the conditioning across models, perform architecture search, and do meta-learning. Unfortunately, the finite-width NTK is notoriously expensive to compute, which severely limits its practical utility.

We perform the first in-depth analysis of the compute and memory requirements for NTK computation in finite width networks. Leveraging the structure of neural networks, we further propose two novel algorithms that change the *exponent* of the compute and memory requirements of the finite width NTK, dramatically improving efficiency.

We open-source [github.com/iclr2022anon/fast_finite_width_ntk] our two algorithms as general-purpose JAX function transformations that apply to any differentiable computation (convolutions, attention, recurrence, etc.) and introduce no new hyper-parameters.

## 1 INTRODUCTION

The past few years have seen significant progress towards a theoretical foundation for deep learning. Much of this work has focused on understanding the properties of random functions in high dimensions. One significant line of work (Neal, 1994; Lee et al., 2018; Matthews et al., 2018; Novak et al., 2019; Garriga-Alonso et al., 2019; Hron et al., 2020; Yang, 2019) established that in the limit of infinite width, randomly initialized Neural Networks (NNs) are Gaussian Processes (called the NNGP). Building on this development, (Jacot et al., 2018) showed that in function space the dynamics under gradient descent could be computed analytically using the so-called Neural Tangent Kernel (NTK) and (Lee et al., 2019) showed that wide neural networks reduce to their linearization in weight space throughout training. A related set of results (Belkin et al., 2019; Spigler et al., 2019) showed that the ubiquitous bias-variance decomposition breaks down as high-dimensional models enter the so-called interpolating regime. Together these results describe learning in the infinite-width limit and help explain the impressive generalization capabilities of NNs.

Insights from the wide network limit have had significant practical impact. The conditioning of the NTK has been shown to significantly impact trainability and generalization in NNs (Schoenholz et al., 2017; Xiao et al., 2018; 2020). This notion inspired initialization schemes like Fixup (Zhang et al., 2019), MetaInit (Dauphin & Schoenholz, 2019), and Normalizer Free networks (Brock et al., 2021a;b) and has enabled efficient neural architecture search (Park et al., 2020; Chen et al., 2021b). The NTK has additionally given insight into a wide range of phenomena such as: the behavior of Generative Adversarial Networks (Franceschi et al., 2021), neural scaling laws (Bahri et al., 2021), and neural irradiance fields (Tancik et al., 2020). Kernel regression using the NTK has further enabled strong performance on small datasets (Arora et al., 2020), and applications such as dataset distillation (Nguyen et al., 2020; 2021) and uncertainty prediction (He et al., 2020; Adlam et al., 2020).

Despite the significant promise of theory based on the NTK, computing the NTK in practice is challenging. In the infinite-width limit, the NTK can sometimes be computed analytically. However, it remains intractable for many architectures, and finite-width corrections can be important to de-

scribe actual NNs used in practice. The NTK can be computed for finite-width networks as the outer-product of Jacobians using forward- or reverse-mode automatic differentiation,

$$\Theta_\theta(x_1, x_2) = \left[\partial f(\theta, x_1)/\partial\theta\right]\left[\partial f(\theta, x_2)/\partial\theta\right]^T. \tag{1}$$

However, this is often infeasible due to computational and memory requirements.

In this paper, we perform the first in-depth analysis of the compute and memory requirements for the NTK as in Equation 1. Noting that forward- and reverse-mode automatic differentiation are two extremes of a wide range of Automatic Differentiation (AD) strategies (Naumann, 2004; 2008), we explore other methods for computing the NTK leveraging the structure of NNs in practice. We propose two novel methods for computing the NTK that exploit different orderings of the computation. We describe the compute and memory requirements of our techniques in fully-connected and convolutional settings, and show that one is asymptotically more efficient in both settings. We compute the NTK over a wide range of NN architectures and demonstrate that these improvements are robust in practice. We open-source implementations of both methods as JAX function transformations.

## 2 Related Work

The finite-width NTK (denoted as simply NTK throughout this work) has been used extensively in many recent works, but to our knowledge implementation details and compute costs were rarely made public. Below we draw comparison to some of these works, but we stress that it only serves as a sanity check to make sure our contribution is valuable relative to the scale of problems that have been attempted (none of these works had efficient NTK computation as their central goal).

In order to compare performance of models based on the NTK and the infinite-width NTK, Arora et al. (2019, Table 2) compute the NTK of up to 20-layer, 128-channel CNN in a binary CIFAR-2 classification setting. In an equivalent setting with the same hardware (NVIDIA V100), we are able to compute the NTK of a 2048-channel CNN, i.e. a network with at least 256 times more parameters.

To demonstrate the stability of the NTK during training for wide networks, Lee et al. (2019, Figure S6) compute the NTK of up to 3-layer $2^{12}$-wide or 1-layer $2^{14}$-wide Fully Connected Networks (FCNs). In the same setting with the same hardware (NVIDIA V100), we can reach widths of at least $2^{14}$ and $2^{18}$ respectively, i.e. handle networks with at least 16 times more parameters.

To investigate convergence of a WideResNet WRN-28-$k$ (Zagoruyko & Komodakis, 2016) to its infinite-width limit, Novak et al. (2020, Figure 2) evaluate the NTK of this model with widening factor $k$ up to 32. In matching setting and hardware, we are able to reach the widening factor of at least 64, i.e. work with models at least 4 times larger.

To meta-learn NN parameters for transfer learning in a MAML-like (Finn et al., 2017) setting, Zhou et al. (2021, Table 7) replace the inner training loop with NTK-based inference. They use up to 5-layer, 200-channel CNNs on MiniImageNet (Oreshkin et al., 2018) with scalar outputs and batch size 25. In same setting we achieve at least 512 channels, i.e. support models at least 6 times larger.

Park et al. (2020, §4.1) use the NTK to predict the generalization performance of architectures in the context of Neural Architecture Search (Zoph & Le, 2017, NAS); however, the authors comment on its high computational burden and ultimately use a different proxy. In another NAS setting, Chen et al. (2021a, §3.1.1) use the condition number of NTK to predict a model's trainability. Chen et al. (2021b, Table 1) also use the NTK to evaluate the trainability of several ImageNet (Deng et al., 2009) models such as ResNet 50/152 (He et al., 2016), Vision Transformer (Dosovitskiy et al., 2021) and MLP-Mixer (Tolstikhin et al., 2021). However, in all of the above cases the authors only evaluate a pseudo-NTK, i.e. an NTK of a scalar-valued function[1], which impacts the quality of the respective trainability/generalization proxy. In this work we can compute the full $1000 \times 1000$ NTK on the same models, i.e. perform a task 1000 times more costly.

Finally, we remark that in all of the above settings, scaling up by increasing width or by working with the true NTK (vs the pseudo-NTK) should lead to improved downstream task performance

[1]Precisely, computing the Jacobian only for a single logit or the sum of all 1000 class logits. The result is not the full NTK, but rather a single diagonal block or the sum of its 1000 diagonal blocks (finite-width NTK is a dense matrix, not block-diagonal).

| Method | Time | Memory | Use when |
|---|---|---|---|
| Jacobian contraction | $\mathbf{N}^2\mathbf{L}\mathbf{O}^2\mathbf{W}^2$ | $\mathbf{N}\mathbf{O}\mathbf{W}^2 + \mathbf{N}^2\mathbf{O}^2 + \mathbf{N}\mathbf{L}\mathbf{W} + \mathbf{L}\mathbf{W}^2$ | Don't |
| NTK-vector products | $\mathbf{N}^2\mathbf{O}^2\mathbf{W} + \mathbf{N}^2\mathbf{L}\mathbf{O}\mathbf{W}^2$ | $\mathbf{N}\mathbf{O}\mathbf{W}^2 + \mathbf{N}^2\mathbf{O}^2 + \mathbf{N}\mathbf{L}\mathbf{W} + \mathbf{L}\mathbf{W}^2$ | $\mathbf{O} > \mathbf{W}$ or $\mathbf{N} = 1$ |
| Structured derivatives | $\mathbf{N}^2\mathbf{L}\mathbf{O}^2\mathbf{W} + \mathbf{N}\mathbf{L}\mathbf{O}\mathbf{W}^2$ | $\mathbf{N}\mathbf{O}\mathbf{W} + \mathbf{N}^2\mathbf{O}^2 + \mathbf{N}\mathbf{L}\mathbf{W} + \mathbf{L}\mathbf{W}^2$ | $\mathbf{O} < \mathbf{W}$ or $\mathbf{L} = 1$ |

Table 1: **Time and memory cost of computing the NTK for an FCN.** Costs are for a pair of batches of inputs of size $\mathbf{N}$ each, and for $\mathbf{L}$-deep, $\mathbf{W}$ -wide FCN with $\mathbf{O}$ outputs. Resulting NTK has shape $\mathbf{NO} \times \mathbf{NO}$. NTK-vector products allow a reduction of the time complexity, while Structured derivatives reduce both time and memory complexity. See §3.6 for discussion, Table 7 for CNN, and Table 2 for more generic cost analysis.

| Method | Time | Memory | Use when |
|---|---|---|---|
| Jacobian contraction | $\mathbf{N}\,\mathbf{O}\,[\mathbf{FP}] + \mathbf{N}^2\mathbf{O}^2\mathbf{P}$ | $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\left(\mathbf{Y}^l + \mathbf{P}^l\right) + \mathbf{P}$ | $\mathbf{P} \ll \mathbf{Y}$, small $\mathbf{O}$, exotic primitives |
| NTK-vector products | $\mathbf{N}^2\mathbf{O}\,[\mathbf{FP}]$ | $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\left(\mathbf{Y}^l + \mathbf{P}^l\right) + \mathbf{P}$ | Cheap **FP**, large $\mathbf{O}$, small $\mathbf{N}$ |
| Structured derivatives | $\mathbf{N}\,\mathbf{O}\,[g(\mathbf{Y}, \mathbf{P})] + \mathbf{N}^2\mathbf{O}^2 \min(\mathbf{Y}, \mathbf{P})$ | $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\,\mathbf{Y}^l + \mathbf{P}$ | Costly **FP**, small $\mathbf{O}$, large $\mathbf{N}$ |

Table 2: **Time and memory cost estimates of computing the NTK for a generic function.** $\mathbf{P}$ stands for the number of all parameters in the network, $\mathbf{Y}$ stands for size of all pre-activations in the network, **FP** stands for forward pass. "Cheap" and "costly" mean relative to $\mathbf{P}$. For example, FCNs have a cheap **FP**, as it consists of a single matrix multiply with the parameter matrix. CNNs, notably when the number of output pixels $\mathbf{D}$ is large, have a costly **FP**, since it amounts to $\mathbf{D}$ matrix multiplies with the parameters. For precise analysis, see Table 1 for FCN and Table 7 for CNN.

due to better infinite-width/linearization approximation or higher-quality trainability/generalization proxy respectively, which makes our work especially relevant to modern research.

## 3 Efficient finite-width NTKs in a simplified setting

To gain intuition for the problem, we start by analyzing and improving the cost of computing the NTK for a simple Fully-Connected Network (FCN). See §F for an equivalent analysis of Convolutional Neural Networks (CNNs). We summarize the resulting time and memory complexities for FCN in Table 1. (see also CNN in Table 7, and a more general takeaway in Table 2).

**Setting.** Consider an $\mathbf{L}$-layer FCN $f(\theta, x) = \theta^{\mathbf{L}}\, \phi\left(\theta^{\mathbf{L}-1} \dots \theta^1\, \phi\left(\theta^0 x\right) \dots\right) \in \mathbb{R}^{\mathbf{O}}$, where $\mathbf{O}$ is the number of logits. We denote individual weight matrices as $\theta^l$ with shapes $\mathbf{W} \times \mathbf{W}$ (except for top-layer $\theta^{\mathbf{L}}$ of shape $\mathbf{O} \times \mathbf{W}$), where $\mathbf{W}$ is the width of the network, and write the set of all parameters as $\theta = \mathrm{vec}\left[\theta^0, \dots, \theta^{\mathbf{L}}\right] \in \mathbb{R}^{\mathbf{L}\mathbf{W}^2 + \mathbf{O}\mathbf{W}}$. We further define $x^l := \phi\left(y^{l-1}\right)$ as post-activations (with $x^0 := x$), and $y^l := \theta^l x^l$ as pre-activations with $y^{\mathbf{L}} = f(\theta, x)$. See Figure 5 for a visual schematic of these quantities. For simplicity, we assume that inputs $x$ also have width $\mathbf{W}$, and $\mathbf{O} = \mathcal{O}(\mathbf{LW})$, i.e. the number of logits is dominated by the product of width and depth.

The NTK of $f$ evaluated at two inputs $x_1$ and $x_2$ is an $\mathbf{O} \times \mathbf{O}$ matrix defined as

$$\Theta_\theta := \frac{\partial f(\theta, x_1)}{\partial \theta} \frac{\partial f(\theta, x_2)}{\partial \theta}^T = \sum_{l=0}^{\mathbf{L}} \frac{\partial f(\theta, x_1)}{\partial \theta^l} \frac{\partial f(\theta, x_2)}{\partial \theta^l}^T =: \sum_{l=0}^{\mathbf{L}} \Theta_\theta^l \in \mathbb{R}^{\mathbf{O} \times \mathbf{O}}, \tag{2}$$

where we have defined $\Theta_\theta^l$ to be the summands. We omit dependence on $x_1$, $x_2$, and $f$ for brevity.

In §3.1 and §3.2 we describe the cost of several fundamental AD operations that we will use as building blocks throughout the text. We borrow the nomenclature introduced by Autograd (Maclaurin et al.) and describe Jacobian-vector products (JVP), vector-Jacobian products (VJP), as well as the cost of computing the Jacobian $\partial f(\theta, x) / \partial \theta$.

In §3.3, we describe the baseline complexity of evaluating the NTK, by computing two Jacobians and contracting them. This approach is used in most (likely all) prior works, and scales poorly with the NN width $\mathbf{W}$ and output size $\mathbf{O}$.

In §3.4 we present our first contribution, that consists in observing that many intermediate operations on weights performed by NNs possess a certain structure, that can allow linear algebra simplifications of the NTK expression, leading to a cheaper contraction and smaller memory footprint.

In §3.5 we present our second contribution, where we rephrase the NTK computation as instantiating itself row-by-row by applying the NTK-vector product function to columns of an identity matrix. As we will show, this trades off Jacobian contraction for more forward passes, which proves beneficial in many (but not all) settings.

### 3.1 Jacobian-vector products and vector-Jacobian products

We begin by defining Jacobian-vector products and vector-Jacobian products:

$$\text{JVP}_{(f,\theta,x)} : \theta_t \in \mathbb{R}^{\mathbf{LW}^2+\mathbf{OW}} \mapsto \frac{\partial f(\theta, x)}{\partial \theta}\theta_t \in \mathbb{R}^{\mathbf{O}}, \tag{3}$$

$$\text{VJP}_{(f,\theta,x)} : f_c \in \mathbb{R}^{\mathbf{O}} \mapsto \frac{\partial f(\theta, x)}{\partial \theta}^T f_c \in \mathbb{R}^{\mathbf{LW}^2+\mathbf{OW}}. \tag{4}$$

The JVP can be understood as pushing forward a tangent vector in weight-space to a tangent vector in the space of outputs; by contrast the VJP pulls back a cotangent vector in the space of outputs to a cotangent vector in weight-space. These elementary operations correspond to forward- and reverse-mode AD respectively and serve as a basis for typical AD computations such as gradients, Jacobians, Hessians, etc. The time cost[2] of both operations is comparable to the forward pass (**FP**), i.e. $[\textbf{FP}] = [\text{cost of all intermediate layers}] + [\text{cost of the top layer}] = \left[\mathbf{LW}^2\right] + [\mathbf{OW}] \sim \mathbf{LW}^2$.

For a single input, the memory cost of computing both the JVP and the VJP are respectively,

$$[\text{size of all weights}] + [\text{size of activations at a single layer}] = \left[\mathbf{LW}^2 + \mathbf{OW}\right] + [\mathbf{W} + \mathbf{O}] \sim \mathbf{LW}^2,$$
$$[\text{size of all weights}] + [\text{size of activations in all layers}] = \left[\mathbf{LW}^2 + \mathbf{OW}\right] + [\mathbf{LW} + \mathbf{O}] \sim \mathbf{LW}^2.$$

Despite the fact that the VJP requires more memory to store intermediate activations (which is necessary for efficient backpropagation), we see that both computations are dominated by the cost of storing the weights.

**Batched inputs.** If $x$ is a batch of inputs of size $\mathbf{N}$, the time cost of JVP and VJP increases linearly to $\mathbf{NLW}^2$. The memory cost is slightly more nuanced. Since weights can be shared across inputs, the memory cost of the JVP and VJP are respectively,

$$\begin{aligned}
&[\text{size of all weights}] + \mathbf{N}\,[\text{size of activations at a single layer}] \\
&\qquad = \left[\mathbf{LW}^2 + \mathbf{OW}\right] + \mathbf{N}\,[\mathbf{W} + \mathbf{O}] \sim \mathbf{LW}^2 + \mathbf{NW} + \mathbf{NO}, \\
&[\text{size of all weights}] + \mathbf{N}\,[\text{size of activations in all layers}] + \mathbf{N}\,[\text{size of all weight matrices}] \\
&\qquad = \left[\mathbf{LW}^2 + \mathbf{OW}\right] + \mathbf{N}\,[\mathbf{LW} + \mathbf{O}] + \mathbf{N}\left[\mathbf{LW}^2 + \mathbf{OW}\right] \sim \mathbf{NLW}^2.
\end{aligned}$$

The cost of the VJP is dominated by the cost of storing the cotangents in weight-space. For the purposes of computing the NTK, we will be contracting Jacobians layerwise and so we will only need to store one cotangent weight matrix, $\partial f / \partial \theta^l$, at a time. Thus, for the purposes of this work we end up with the following costs:

- JVP costs $\mathbf{NLW}^2$ time and $\mathbf{LW}^2 + \mathbf{NW} + \mathbf{NO}$ memory.
- VJP costs $\mathbf{NLW}^2$ time and $\mathbf{LW}^2 + \mathbf{NLW} + \mathbf{NW}^2 + \mathbf{NOW}$ memory.

### 3.2 Jacobian computation

For neural networks, the Jacobian is most often computed by evaluating the VJP on rows of the identity matrix $I_{\mathbf{O}}$, i.e.

$$\left[\partial f(\theta, x) / \partial \theta\right]^T = \left[\partial f(\theta, x) / \partial \theta\right]^T I_{\mathbf{O}} \in \mathbb{R}^{(\mathbf{LW}^2+\mathbf{OW}) \times \mathbf{O}}. \tag{5}$$

It follows that computing the Jacobian takes $\mathbf{O}$ evaluations of the VJP. However, as above we only need to store one $\partial f / \partial \theta^l$ at a time and the weights and intermediate activations are reused across

[2] To declutter notation, we omit the $\mathcal{O}$ symbol to indicate asymptotic complexity in this work.

evaluations. Thus, the time and memory costs to compute the Jacobian are respectively,

$$\mathbf{ON}\left([\text{cost of all intermediate layers}] + [\text{cost of the top layer}]\right)$$
$$= \mathbf{ON}\left(\left[\mathbf{LW}^2\right] + [\mathbf{OW}]\right) \sim \mathbf{NLOW}^2 + \mathbf{NO}^2\mathbf{W},$$
$$[\text{size of all weights}] + \mathbf{N}\,[\text{size of activations in all layers}] + \mathbf{ON}\,[\text{size of a single weight matrix}]$$
$$= \left[\mathbf{LW}^2 + \mathbf{OW}\right] + \mathbf{N}\left[\mathbf{LW} + \mathbf{O}\right] + \mathbf{ON}\left[\mathbf{W}^2 + \mathbf{OW}\right] \sim \mathbf{LW}^2 + \mathbf{NLW} + \mathbf{NOW}^2 + \mathbf{NO}^2\mathbf{W}.$$

Therefore, asymptotically,

> Jacobian costs $\mathbf{NLOW}^2 + \mathbf{NO}^2\mathbf{W}$ time and $\mathbf{LW}^2 + \mathbf{NLW} + \mathbf{NOW}^2 + \mathbf{NO}^2\mathbf{W}$ memory.

### 3.3 Jacobian contraction

We now analyze the cost of computing the NTK, starting with the direct computation as the product of two Jacobians. Consider a single summand from Equation 2:

$$\underbrace{\Theta_\theta^l}_{\mathbf{O}\times\mathbf{O}} = \underbrace{\frac{\partial f(\theta, x_1)}{\partial \theta^l}}_{\mathbf{O}\times(\mathbf{W}\times\mathbf{W})} \underbrace{\frac{\partial f(\theta, x_2)}{\partial \theta^l}^T}_{(\mathbf{W}\times\mathbf{W})\times\mathbf{O}}. \tag{6}$$

The time cost of this contraction is $\mathbf{O}^2\mathbf{W}^2$, and the memory necessary to instantiate each factor and the result is $\mathbf{OW}^2 + \mathbf{O}^2$. Repeating the above operation for each $\theta^l$, we arrive at $\mathbf{LO}^2\mathbf{W}^2$ time cost and unchanged memory, due to being able to process summands sequentially.

**Batched inputs.** If we consider $x_1$ and $x_2$ to be input batches of size $\mathbf{N}$, then the resulting NTK is a matrix of shape $\mathbf{NO} \times \mathbf{NO}$, and the time cost becomes $\mathbf{N}^2\mathbf{LO}^2\mathbf{W}^2$, while memory grows to $[\text{NTK matrix size}] + [\text{factors size}] = \mathbf{N}^2\mathbf{O}^2 + \mathbf{NOW}^2$.

What remains is to account for the cost of computing and storing individual derivatives $\partial f / \partial \theta^l$, which is exactly the cost of computing the Jacobian described in §3.2. Adding the costs up we obtain

> Jacobian contraction costs $\mathbf{N}^2\mathbf{LO}^2\mathbf{W}^2$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOW}^2 + \mathbf{NO}^2\mathbf{W} + \mathbf{LW}^2 + \mathbf{NLW}$ memory.

### 3.4 Leveraging structured derivatives for computing the NTK

We can rewrite ${\Theta^l}_\theta$ in Equation 6 using the chain rule and our pre- and post-activation notation as:

$$\Theta_\theta^l = \left[\frac{\partial f(\theta, x_1)}{\partial y_{x_1}^l}\frac{\partial y_{x_1}^l}{\partial \theta^l}\right]\left[\frac{\partial f(\theta, x_2)}{\partial y_{x_2}^l}\frac{\partial y_{x_2}^l}{\partial \theta^l}\right]^T = \underbrace{\frac{\partial f(\theta, x_1)}{\partial y_{x_1}^l}}_{\mathbf{O}\times\mathbf{W}} \underbrace{\frac{\partial y_{x_1}^l}{\partial \theta^l}}_{\mathbf{W}\times(\mathbf{W}\times\mathbf{W})} \underbrace{\frac{\partial y_{x_2}^l}{\partial \theta^l}^T}_{(\mathbf{W}\times\mathbf{W})\times\mathbf{W}} \underbrace{\frac{\partial f(\theta, x_2)}{\partial y_{x_2}^l}^T}_{\mathbf{W}\times\mathbf{O}}. \tag{7}$$

At face value, rewriting Equation 6 in this way is unhelpful as it appears to have introduced additional costly contractions. However, recall that $y^l = \theta^l x^l$, and therefore

$$\frac{\partial y_{x_1}^l}{\partial \theta^l} = I_{\mathbf{W}} \otimes {x_1^l}^T, \quad \frac{\partial y_{x_2}^l}{\partial \theta^l} = I_{\mathbf{W}} \otimes {x_2^l}^T, \tag{8}$$

where $\otimes$ is the Kronecker product. Plugging Equation 8 into Equation 7 we obtain (see §G)

$$\Theta_\theta^l = \left(\underbrace{{x_1^l}^T}_{1\times\mathbf{W}} \quad \underbrace{x_2^l}_{\mathbf{W}\times 1}\right)\left[\underbrace{\frac{\partial f(\theta, x_1)}{\partial y_{x_1}^l}}_{\mathbf{O}\times\mathbf{W}} \quad \underbrace{\frac{\partial f(\theta, x_2)}{\partial y_{x_2}^l}^T}_{\mathbf{W}\times\mathbf{O}}\right], \tag{9}$$

and observe that it takes only $\mathbf{O}^2\mathbf{W}$ time and $\mathbf{OW} + \mathbf{O}^2$ memory. Accounting for depth, time cost increases by a factor of depth $\mathbf{L}$ and becomes $\mathbf{LO}^2\mathbf{W}$, while memory does not change since the summands can be processed sequentially.

**Batched inputs.** In the batched setting, the time cost grows quadratically with the size of the NTK to $\mathbf{N}^2\mathbf{LO}^2\mathbf{W}$, while the memory cost increases to $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOW}$ to store the result, $\Theta_\theta^l(x_1, x_2)$, and factors, $\partial f(\theta, x) / \partial y_x^l$, respectively.

Finally, we need to account for the cost of computing the derivatives, $\partial f / \partial y^l$, and post-activations, $x^l$. Notice that both $x^l$ and $\partial f / \partial y^l$ arises naturally when computing the Jacobian as the primals and cotangents in layer $l$ respectively. However, since we do not need to compute the weight-space cotangents explicitly (in other words, we cut the backpropagation algorithm short) the memory cost will be,

$$\begin{aligned}&[\text{size of all weights}] + \mathbf{N}\,[\text{size of activations in all layers}]\\ &\quad = \left[\mathbf{LW}^2 + \mathbf{OW}\right] + \mathbf{N}\left[\mathbf{LW} + \mathbf{O}\right] \sim \mathbf{LW}^2 + \mathbf{NLW}.\end{aligned}$$

The extra time cost is asymptotically the cost of $\mathbf{O}$ forward-passes, $\mathbf{NLOW}^2$ which is the same as the Jacobian. However, as we will see in experiments, in practice we'll often compute the NTK faster than the Jacobian. Putting everything together we find the following costs,

> By leveraging Structured derivatives in NN computations, we have reduced the cost of NTK to $\mathbf{N}^2\mathbf{LO}^2\mathbf{W} + \mathbf{NLOW}^2$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOW} + \mathbf{LW}^2 + \mathbf{NLW}$ memory.

The key insight was to leverage the constant block-diagonal structure of the pre-activation derivatives $\partial y^l / \partial \theta^l$. This idea is quite general; as we discuss in §4 and detail in the appendix, similar structure exists for many common operations such as convolutions, pooling, and arithmetic. However, the improvements discussed in this section do not emerge automatically in AD. While JAX and other libraries leverage structures analogous to Equation 8 to efficiently compute single evaluations of the VJP and JVP, this structure is lost once the (structureless) Jacobian is instantiated (e.g. by composing the VJP with vectorization and contraction). We will discuss how we impose this structure to compute the NTK for general neural networks in §4.

### 3.5 NTK VIA NTK-VECTOR PRODUCTS

Computing the Jacobian using Jacobian contraction first instantiates the Jacobian using using VJPs and then performs a contraction. Structured derivatives use a similar strategy, but speed-up the contraction and avoid explicitly instantiating the weight-space cotangents. In this section we avoid performing a contraction altogether at the cost of extra VJP/JVP calls; this ends up being beneficial for FCNs.

We introduce the linear function performing the NTK-vector product: $\Theta\text{VP} : v \in \mathbb{R}^{\mathbf{O}} \mapsto \Theta_\theta v \in \mathbb{R}^{\mathbf{O}}$. Applying this function to $\mathbf{O}$ columns of the identity matrix $I_{\mathbf{O}}$ allows us to compute the NTK, i.e. $\Theta_\theta I_{\mathbf{O}} = \Theta_\theta$. The cost of evaluating the NTK in this fashion is equal to $\mathbf{O}$ times the cost of a single NTK-vector product evaluation $\Theta\text{VP}(v)$. We now expand $\Theta\text{VP}(v) = \Theta_\theta v$ as

$$\frac{\partial f(\theta, x_1)}{\partial \theta} \frac{\partial f(\theta, x_2)}{\partial \theta}^T v = \frac{\partial f(\theta, x_1)}{\partial \theta} \text{VJP}_{(f,\theta,x_2)}(v) = \text{JVP}_{(f,\theta,x_1)}\left[\text{VJP}_{(f,\theta,x_2)}(v)\right], \tag{10}$$

where we have observed that, if contracted from right to left, the NTK-vector product can be expressed as a composition of a JVP and VJP of the underlying function $f$. The cost of this operation is asymptotically equivalent to the cost of Jacobian, since it consists of $\mathbf{O}$ VJPs followed by $\mathbf{O}$ (cheaper) JVPs. Therefore it costs $\mathbf{LOW}^2 + \mathbf{O}^2\mathbf{W}$ time and $\mathbf{LW}^2 + \mathbf{OW}^2 + \mathbf{O}^2\mathbf{W}$ memory.

**Batched inputs.** In the batched setting Equation 10 is repeated for each pair of inputs, and therefore time increases by a factor of $\mathbf{N}^2$ to become $\mathbf{N}^2\mathbf{LOW}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{W}$. However, the memory cost grows linearly in $\mathbf{N}$ (except for the cost of storing the NTK of size $\mathbf{N}^2\mathbf{O}^2$), since intermediate activations and derivatives necessary to compute the JVP and VJP can be computed for each batch $x_1$ and $x_2$ separately; these quantities are then reused for every pairwise combination resulting in a memory cost equal to the cost of computing the Jacobian over a batch, i.e. $\mathbf{N}^2\mathbf{O}^2 + \left(\mathbf{LW}^2 + \mathbf{NOW}^2 + \mathbf{NO}^2\mathbf{W} + \mathbf{NLW}\right)$.

NTK computation as a sequence of NTK-vector products costs $\mathbf{N}^2\mathbf{LOW}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{W}$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOW}^2 + \mathbf{LW}^2 + \mathbf{NLW}$ memory.

### 3.6 Summary

NTK-vector products and Structured derivatives allow a reduction in the time cost of NTK computation in different ways, and the latter also reduces memory requirements. NTK-vector products are beneficial for networks with large outputs $\mathbf{O}$, while Structured derivatives are beneficial for wide networks, with large $\mathbf{W}$. We confirm our predictions with FLOPs measurements in Figure 1 (left).

We further confirm our methods can provide orders of magnitude speed-ups and memory savings on all major hardware platforms in Figure 1 (right) and Figure 3. However, we notice that our wall-clock time measurements often deviate from predictions due to unaccounted constant overheads of various methods, hardware specifics, padding, and the (largely black-box) behavior of the XLA compiler. Notably, in practice, we find Structured derivatives almost always outperform NTK-vector products.

Finally, we evaluate our methods in the wild, and confirm computational benefits on full ImageNet models in Figure 2 and Figure 4.

## 4 Structured derivatives for generic functions

In this section we generalize the idea of leveraging structure in subexpression derivatives presented in §3.4. This section (and our implementation) is not specific to NNs and applies to any differentiable real-valued functions.

Consider two differentiable functions defined on a common input domain (but potentially different output codomains):

$$f_i : \left(\theta^0, \ldots, \theta^{\mathbf{L}}\right) \in \mathbb{R}^{\mathbf{P}_0 \times \cdots \times \mathbf{P}_{\mathbf{L}}} \mapsto f_i\left(\theta^0, \ldots, \theta^{\mathbf{L}}\right) \in \mathbb{R}^{\mathbf{O}_i} \quad (i \in \{1, 2\}).$$

For NNs, typically $\left(\theta^0, \ldots, \theta^{\mathbf{L}}\right)$ correspond to trainable parameters in layers $0, \ldots, \mathbf{L}$, and $f_i\left(\theta^0, \ldots, \theta^{\mathbf{L}}\right) = f\left(\theta^0, \ldots, \theta^{\mathbf{L}}, x_i\right)$, $x_i$ being network inputs, $\mathbf{O}_i = \mathbf{O}$ being the number of outputs (logits, classes). The NTK is defined as

$$\Theta_\theta\left(f_1, f_2\right) := \sum_{l=0}^{\mathbf{L}} \frac{\partial f_1}{\partial \theta^l} \frac{\partial f_2}{\partial \theta^l}^T \in \mathbb{R}^{\mathbf{O}_1 \times \mathbf{O}_2}. \tag{11}$$

Assume, without loss of generality, the following decomposition of $f_1$ and $f_2$ into a computational graph made of primitives $y_1$ and $y_2$:

$$f_i\left(\theta^0, \ldots, \theta^{\mathbf{L}}\right) = \tilde{f}_i\left(y_i^1(\theta^0, \ldots, \theta^{\mathbf{L}}), \ldots, y_i^{\mathbf{K}_i}(\theta^0, \ldots, \theta^{\mathbf{L}})\right) \quad (i \in \{1, 2\}). \tag{12}$$

with $y_i^k\left(\theta^0, \ldots, \theta^{\mathbf{L}}\right) \in \mathbb{R}^{\mathbf{Y}_i^k}$. In common NNs, $y_i^{k_i}$ would correspond to pre-activations evaluated on inputs $x_i$ in layer $k_i$, and, without weight sharing, typically $\mathbf{K}_1 = \mathbf{K}_2 = \mathbf{L}$. However, here we do not impose any relationship between the number of parameter variables $\mathbf{L}$ and number of primitive subexpressions $\mathbf{K}_1$ and $\mathbf{K}_2$, and allow arbitrary weight sharing. We can then use the chain rule in Equation 2 to obtain:

$$\Theta_\theta\left(f_1, f_2\right) = \sum_{l, k_1, k_2}^{\mathbf{L}, \mathbf{K}_1, \mathbf{K}_2} \left(\frac{\partial \tilde{f}_1}{\partial y_1^{k_1}} \frac{\partial y_1^{k_1}}{\partial \theta^l}\right) \left(\frac{\partial \tilde{f}_2}{\partial y_2^{k_2}} \frac{\partial y_2^{k_2}}{\partial \theta^l}\right)^T = \sum_{l, k_1, k_2}^{\mathbf{L}, \mathbf{K}_1, \mathbf{K}_2} \frac{\partial \tilde{f}_1}{\partial y_1^{k_1}} \frac{\partial y_1^{k_1}}{\partial \theta^l} \frac{\partial y_2^{k_2}}{\partial \theta^l}^T \frac{\partial \tilde{f}_2}{\partial y_2^{k_2}}^T. \tag{13}$$

All methods from §3 perform the sum of contractions in Equation 13 one way or another. Jacobian contraction uses VJPs to implicitly contract each summand "outside-in", i.e. it first computes $\partial f_i / \partial \theta^l$ terms with VJPs followed by their contraction. As discussed in §3.3, this costs (in a batched setting) $\mathbf{NO}\left[\mathbf{FP}\right] + \mathbf{N}^2\mathbf{O}^2\mathbf{P}$, where $\mathbf{FP}$ is the cost of the forward pass.

NTK-vector products uses both JVPs and VJPs to contract "Right-to-left", i.e. first compute $\partial f_2 / \partial \theta^l$ as an implicit contraction of $\partial f_2 / \partial y_2$ with $\partial y_2 / \partial \theta^l$ via VJP, followed by an implicit contraction

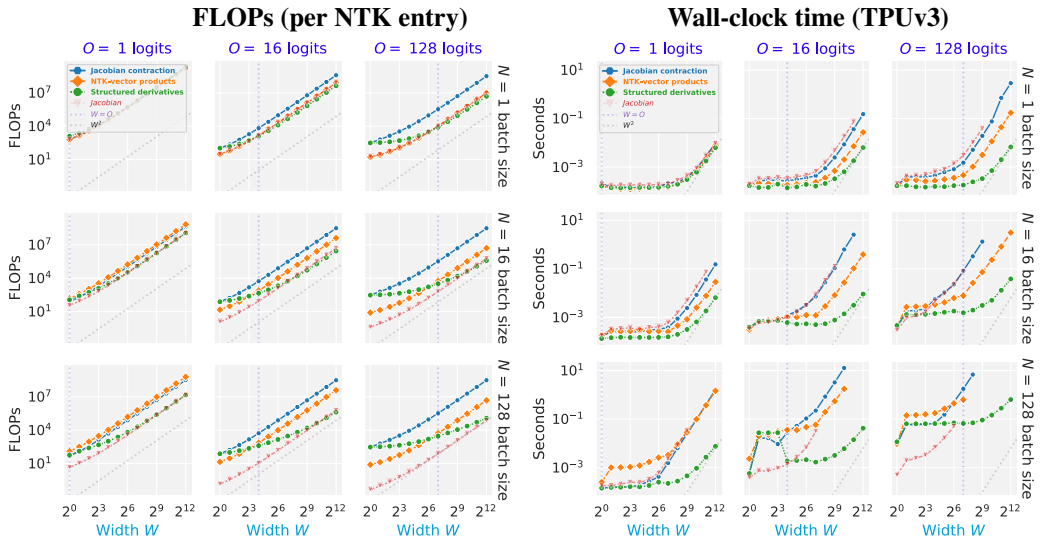


Figure 1: **FLOPs (left) and wall-clock time (right) of computing the NTK for a 10-layer ReLU FCN.** As predicted by Table 1, our methods almost always outperform Jacobian contraction, allowing orders of magnitude speed-ups and memory improvements for realistic problem sizes. **FLOPs per NTK entry:** We confirm several specific predictions: (1) NTK-vector products is the best performing method for $\mathbf{N} = 1$, and has cost equivalent to Jacobian for any width $\mathbf{W}$ or output size $\mathbf{O}$ (top row); (2) NTK-vector products offer an $\mathbf{O}$-fold improvement over Jacobian contraction (left to right columns); (3) NTK-vector products are equivalent to Jacobian contraction for $\mathbf{O} = 1$ (leftmost column); (4) Structured derivatives outperform NTK-vector products iff $\mathbf{O} < \mathbf{W}$ ($\mathbf{O} = \mathbf{W}$ are plotted as pale vertical lines, which is where Structured derivatives and NTK-vector products intersect); (5) Structured derivatives approach the cost of Jacobian in the limit of large width $\mathbf{W}$ (left to right). (6) All methods, as expected, scale quadratically with width $\mathbf{W}$. **Wall-clock runtime:** In real applications, given hardware-specific constraints, padding, and delicate interplay with the XLA compiler, we observe that: (1) NTK-vector products improve upon Jacobian contraction for $\mathbf{O} > 1$, but the effect is not perfectly robust (see bottom row for small $\mathbf{W}$ and Figure 3, notably GPU platforms); (2) Structured derivatives robustly outperform all other methods, including simply computing the Jacobian, as discussed in §3.4; (3) Structured derivatives has lower memory footprint, and reaches up to 8x larger widths (bottom right; missing points indicate out-of-memory errors), i.e. can process models up to 64x larger than other methods, as discussed in §3.4; (4) All methods have a smaller memory footprint than Jacobian, as discussed in §3.1. **More:** see Figure 3 for time measurements on other hardware platforms, and §H for details.

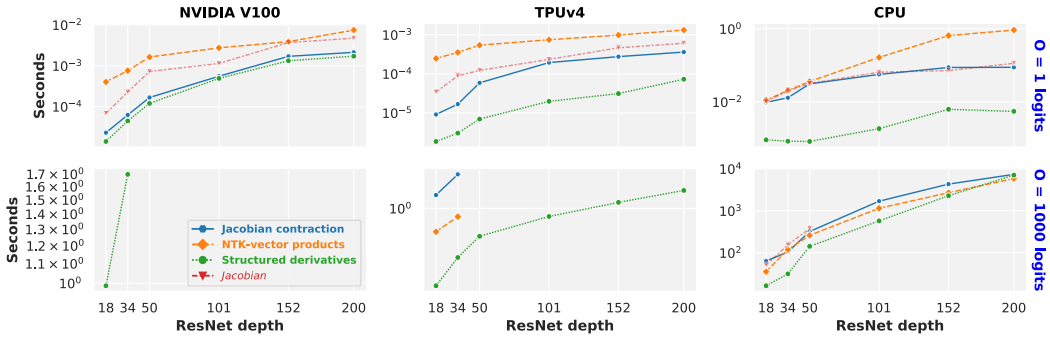


Figure 2: **Wall-clock time cost of computing an NTK for several ResNet sizes on a pair of ImageNet inputs**. Structured derivatives allow the NTK to be computed faster and for larger models (see bottom row – missing points indicate out-of-memory error). NTK-vector products, as predicted in §3.6 and Table 2, are advantageous for large $\mathbf{O}$ (bottom row), but are suboptimal when the cost of the forward pass is large relative to the number of parameters, e.g. when there is a lot of weight sharing (see Table 7 and Table 2). For this reason in practice they rarely performs best on convolutional models. See Figure 4 for more ImageNet models, and §H for experimental details, and §F for analysis of CNN rewrite rules.

of the result with $\partial y_1 / \partial \theta^l$ via a JVP, followed by another implicit contraction with $\partial f_1 / \partial y_1$ with another JVP. Per §3.5 this costs $\mathbf{N}^2\mathbf{O}\,[\mathbf{FP}]$.

The idea of Structured derivatives is to design rules for efficient computation of such contractions, similarly to how JAX and other packages have rules for efficient JVPs and VJPs. From Equation 13, in the general case this requires hand-made rules for all pairwise combinations of primitives $y_1$ and $y_2$. Due to quadratic scaling in the number of primitives, we restrict the current implementation to rules that operate on individual primitives $y$. This still provide substantial computational benefit.

Specifically, our rules identify a few simple types of structure (e.g. block diagonal, constant-block diagonal, tiling) in $\partial y / \partial \theta^l$, that allow us to simplify the contraction in Equation 13. In practice this amounts to replacing the inner terms $\partial y_1^{k_1} / \partial \theta^l$ and $\partial y_2^{k_2} / \partial \theta^l$ with their (much) smaller subarrays, and modifying the instructions passed to `np.einsum` that contracts all 4 terms. In §C we provide specific descriptions of our rules and their impact on the computational complexity of Equation 13.

In Table 1 and Table 7 we show that our rules are asymptotically better than Jacobian contraction for matrix multiplications and convolutions, and verify that they are practically beneficial in a much wider set of operations used by contemporary ImageNet models in Figure 2 and Figure 4.

For both Structured derivatives and NTK-vector products a fully general and rigorous comparison of complexities is not feasible since it will rely upon specifics of the model architecture and the pairs of primitives, $y_1$ and $y_2$, present in the network. Nonetheless, we can offer heuristics that suggest when each method will be beneficial. The time-complexity of Structured derivatives is at most $\mathbf{NO}\,[g(\mathbf{Y}, \mathbf{P})] + \mathbf{N}^2\mathbf{O}^2 \min(\mathbf{Y}, \mathbf{P})$, where $\max(\mathbf{Y}, \mathbf{P}) \leq g(\mathbf{Y}, \mathbf{P}) \leq \mathbf{YP}$ (Table 3), and the exact complexity depends on the structure present in $y_1$ and $y_2$. This is guaranteed to be no worse than Jacobian contraction if $g(\mathbf{Y}, \mathbf{P}) \leq [\mathbf{FP}]$, which is the case for FCNs and CNNs. From Table 2, the performance of NTK-vector products relative to Jacobian contraction ultimately depends on the cost of the forward pass through the network, $[\mathbf{FP}]$, relative to $\mathbf{OP}$. In practice this amounts to best performance on models without weight sharing like FCNs.

Owing to the nuanced trade-offs between different computational methods in the general case, we release all our implementations as a single function that allows the user to manually select the desired implementation. For convenience, we include an automated setting which will perform FLOPs analysis for each method at compilation time and automatically select the most efficient one.

## 5 Implementation

Both algorithms are implemented in JAX (Bradbury et al., 2018) as the following function transformation `ntk_fn` : $[f : (\theta, x) \mapsto f(\theta, x)] \mapsto [\Theta : (x_1, x_2, \theta) \mapsto \Theta_\theta(x_1, x_2)]$, i.e. our functions accept any function $f$ with the above signature and returns the efficient NTK kernel function operating on inputs $x_1$ and $x_2$ and parameterized by $\theta$. Inputs $x$, parameters $\theta$, and outputs $f(\theta, x)$ can be arbitrary PyTrees. We rely on many utilities from JAX and Neural Tangents (Novak et al., 2020).

**NTK-vector products** algorithm is implemented by using JAX core operations such as `vjp`, `jvp`, and `vmap` to map the NTK-vp function to the $I_{\mathbf{O}}$ matrix and to parallelize the computation over pairwise combinations of $\mathbf{N}$ inputs in each batch $x_1$ and $x_2$.

**Structured derivatives** algorithm is implemented as a Jaxpr interpreter, built on top of the default JAX reverse-mode AD interpreter. On a high level, the algorithm performs the sum in Equation 13. Each summand is a contraction of 4 factors: $\partial \tilde{f}_1 / \partial y_1, \partial y_1 / \partial \theta, \partial y_2 / \partial \theta, \partial \tilde{f}_2 / \partial y_2$.

First, we linearize $f$ to obtain a computational graph constructed out of a limited set (54,[3] see Table 5) of linear primitives $y^1, \ldots, y^{\mathbf{K}}$. Then, we can obtain two factors $\partial \tilde{f}_1 / \partial y_1, \partial \tilde{f}_2 / \partial y_2$ as part of a backward pass almost identical to calling `jax.jacobian`$(f)(\theta, x)$. To contract these terms with $\partial y_1 / \partial \theta$ and $\partial y_2 / \partial \theta$, as described above, we query a dictionary of rules which map primitives to a structural description (§C.7); for a given pair of primitives, these rules allow us to analytically simplify the contraction and avoid explicitly instantiating the derivatives.

[3] JAX leverages a similar approach to implement only 54 transpose rules for linear primitives for reverse-mode differentiation instead of 131 VJP rules (Frostig et al., 2021).

## REFERENCES

Ben Adlam, Jaehoon Lee, Lechao Xiao, Jeffrey Pennington, and Jasper Snoek. Exploring the uncertainty properties of neural networks' implicit priors in the infinite-width limit. In *International Conference on Learning Representations*, 2020.

Sanjeev Arora, Simon S Du, Wei Hu, Zhiyuan Li, Russ R Salakhutdinov, and Ruosong Wang. On exact computation with an infinitely wide neural net. In *Advances in Neural Information Processing Systems*, pp. 8141–8150. Curran Associates, Inc., 2019.

Sanjeev Arora, Simon S. Du, Zhiyuan Li, Ruslan Salakhutdinov, Ruosong Wang, and Dingli Yu. Harnessing the power of infinitely wide deep nets on small-data tasks. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkl8sJBYvH.

Yasaman Bahri, Ethan Dyer, Jared Kaplan, Jaehoon Lee, and Utkarsh Sharma. Explaining neural scaling laws. *arXiv preprint arXiv:2102.06701*, 2021.

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

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

Andrew Brock, Soham De, and Samuel L Smith. Characterizing signal propagation to close the performance gap in unnormalized resnets. *arXiv preprint arXiv:2101.08692*, 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.

Wuyang Chen, Xinyu Gong, and Zhangyang Wang. Neural architecture search on imagenet in four gpu hours: A theoretically inspired perspective. In *International Conference on Learning Representations*, 2021a.

Xiangning Chen, Cho-Jui Hsieh, and Boqing Gong. When vision transformers outperform resnets without pretraining or strong data augmentations, 2021b.

Yann Dauphin and Samuel S Schoenholz. Metainit: Initializing learning by learning to initialize. 2019.

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.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1126–1135. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/finn17a.html.

Jean-Yves Franceschi, Emmanuel de Bézenac, Ibrahim Ayed, Mickaël Chen, Sylvain Lamprier, and Patrick Gallinari. A neural tangent kernel perspective of gans. *arXiv preprint arXiv:2106.05566*, 2021.

Roy Frostig, Matthew J Johnson, Dougal Maclaurin, Adam Paszke, and Alexey Radul. Decomposing reverse-mode automatic differentiation. *arXiv preprint arXiv:2105.09469*, 2021.

Adrià Garriga-Alonso, Laurence Aitchison, and Carl Edward Rasmussen. Deep convolutional networks as shallow gaussian processes. In *International Conference on Learning Representations*, 2019.

Bobby He, Balaji Lakshminarayanan, and Yee Whye Teh. Bayesian deep ensembles via the neural tangent kernel. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/0b1ec366924b26fc98fa7b71a9c249cf-Abstract.html.

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, 2016.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2020. URL http://github.com/google/flax.

Jiri Hron, Yasaman Bahri, Jascha Sohl-Dickstein, and Roman Novak. Infinite attention: NNGP and NTK for deep attention networks. In *International Conference on Machine Learning*, 2020.

Arthur Jacot, Franck Gabriel, and Clement Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in Neural Information Processing Systems*, 2018.

Jaehoon Lee, Yasaman Bahri, Roman Novak, Sam Schoenholz, Jeffrey Pennington, and Jascha Sohl-dickstein. Deep neural networks as gaussian processes. In *International Conference on Learning Representations*, 2018.

Jaehoon Lee, Lechao Xiao, Samuel S. Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. In *Advances in Neural Information Processing Systems*, 2019.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy.

Alexander G. de G. Matthews, Jiri Hron, Mark Rowland, Richard E. Turner, and Zoubin Ghahramani. Gaussian process behaviour in wide deep neural networks. In *International Conference on Learning Representations*, 2018.

Uwe Naumann. Optimal accumulation of jacobian matrices by elimination methods on the dual computational graph. *Mathematical Programming*, 99(3):399–421, 2004.

Uwe Naumann. Optimal jacobian accumulation is np-complete. *Mathematical Programming*, 112 (2):427–441, 2008.

Radford M. Neal. Priors for infinite networks (tech. rep. no. crg-tr-94-1). *University of Toronto*, 1994.

Timothy Nguyen, Zhourong Chen, and Jaehoon Lee. Dataset meta-learning from kernel ridge-regression. *arXiv preprint arXiv:2011.00050*, 2020.

Timothy Nguyen, Roman Novak, Lechao Xiao, and Jaehoon Lee. Dataset distillation with infinitely wide convolutional networks. *arXiv preprint arXiv:2107.13034*, 2021.

Roman Novak, Lechao Xiao, Jaehoon Lee, Yasaman Bahri, Greg Yang, Jiri Hron, Daniel A. Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Bayesian deep convolutional networks with many channels are gaussian processes. In *International Conference on Learning Representations*, 2019.

Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A. Alemi, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. Neural tangents: Fast and easy infinite neural networks in python. In *International Conference on Learning Representations*, 2020. URL https://github.com/google/neural-tangents.

Boris N. Oreshkin, Pau Rodríguez López, and Alexandre Lacoste. Tadam: Task dependent adaptive metric for improved few-shot learning. In *NeurIPS*, 2018.

Daniel S Park, Jaehoon Lee, Daiyi Peng, Yuan Cao, and Jascha Sohl-Dickstein. Towards nngp-guided neural architecture search. *arXiv preprint arXiv:2011.06006*, 2020.

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

Stefano Spigler, Mario Geiger, Stéphane d'Ascoli, Levent Sagun, Giulio Biroli, and Matthieu Wyart. A jamming transition from under-to over-parametrization affects generalization in deep learning. *Journal of Physics A: Mathematical and Theoretical*, 52(47):474001, 2019.

Andreas Steiner, Alexander Kolesnikov, , Xiaohua Zhai, Ross Wightman, Jakob Uszkoreit, and Lucas Beyer. How to train your vit? data, augmentation, and regularization in vision transformers. *arXiv preprint arXiv:2106.10270*, 2021.

Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *NeurIPS*, 2020.

Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. Mlp-mixer: An all-mlp architecture for vision, 2021.

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*, 2018.

Lechao Xiao, Jeffrey Pennington, and Samuel S Schoenholz. Disentangling trainability and generalization in deep learning. In *International Conference on Machine Learning*, 2020.

Greg Yang. Scaling limits of wide neural networks with weight sharing: Gaussian process behavior, gradient independence, and neural tangent kernel derivation. *arXiv preprint arXiv:1902.04760*, 2019.

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

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

Yufan Zhou, Zhenyi Wang, Jiayi Xian, Changyou Chen, and Jinhui Xu. Meta-learning with neural tangent kernels. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Ti87Pv5Oc8.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017. URL https://arxiv.org/abs/1611.01578.