4th Symposium on Advances in Approximate Bayesian Inference, 2022 1–39

Fast Finite Width Neural Tangent Kernel

Anonymous Authors
Anonymous Institution

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 provides a precise posterior distribution of the NN outputs after training, and can be used for uncertainty estimation and modelling deep ensembles. However, the infinite width NTK rarely admits a closed-form solution, and when it does, it is usually prohibitively expensive to compute exactly, and a finite width NTK can be used as a Monte Carlo approximation. The finite width NTK also models approximate inference in finite Bayesian NNs (BNNs), and has widespread applications in deep learning theory, meta-learning, neural architecture search, and many other areas.

Unfortunately, the finite width NTK is also 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, improving efficiency by orders of magnitude in a wide range of practical models on all major hardware platforms.

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 principled probabilistic modelling of deep neural networks. Much of this effort 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., 2020b; Yang, 2019) established that in the limit of infinite width, outputs of Bayesian Neural Networks at initialization converge to Gaussian Processes (called the NNGP – *Neural Network Gaussian Process*). Building on this development, Hron et al. (2020a) proved that after *Bayesian training* BNNs also converge to the respective NNGP posterior in the infinite width limit, while Jacot et al. (2018); Lee et al. (2019) showed that after *gradient descent training*, the posterior distribution over outputs is also described by a GP with the so called Neural Tangent Kernel.

The NTK has since been used in a variety of settings, including modelling deep ensembles (He et al., 2020), approximate inference in BNNs (Khan et al., 2019), uncertainty estimation and calibration (Adlam et al., 2020), insights into trainability and generalization (Xiao et al., 2020), improving NN initialization (Zhang et al., 2019; Dauphin and Schoenholz, 2019; Brock et al., 2021a,b), neural architecture search (Park et al., 2020; Chen et al., 2021b, NAS), deep learning phenomenology (Fort et al., 2020), and many others. The NTK has

© A. Authors.

additionally given insight into a wide range of phenomena such as: 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 meta-learning (Zhou et al., 2021).

Despite the significant promise of theory and applications based on the NTK, computing the NTK in practice is challenging. In the infinite width limit, the NTK can rarely be computed in closed-form, and when it can, it is usually prohibitively expensive, demanding thousands of GPU-hours on the simplest datasets like MNIST and CIFAR-10 (Arora et al., 2019b; Novak et al., 2020). Furthermore, in many applications finite width corrections are important to describe actual NNs used in practice, and using the infinite width limit is sub-optimal or not applicable. We provide a detailed discussion on the benefits and drawbacks of finite and infinite width NTK in §G, and focus on the finite width NTK in this work, which we denote as simply NTK.

The NTK matrix can be computed as the outer product of Jacobians using forward or reverse mode automatic differentiation (AD),

$$\underbrace{\Theta_\theta(x_1, x_2)}_{\mathbf{O}\times\mathbf{O}} := \underbrace{\left[\partial f(\theta, x_1)/\partial\theta\right]}_{\mathbf{O}\times\mathbf{P}} \underbrace{\left[\partial f(\theta, x_2)/\partial\theta\right]^T}_{\mathbf{P}\times\mathbf{O}}, \tag{1}$$

where f is the forward pass NN function producing outputs in $\mathbb{R}^{\mathbf{O}}$, $\theta \in \mathbb{R}^{\mathbf{P}}$ are all trainable parameters, and x_1 and x_2 are two inputs to the network. If inputs are batches of sizes $\mathbf{N}_1$ and $\mathbf{N}_2$, the NTK is an $\mathbf{N}_1\mathbf{O} \times \mathbf{N}_2\mathbf{O}$ matrix.

Unfortunately, evaluating Eq. (1) is often infeasible due to time and memory requirements.

In this note, we perform the first in-depth analysis of the compute and memory requirements for the NTK as in Eq. (1). Noting that forward and reverse mode AD are two extremes of a wide range of AD strategies (Naumann, 2004, 2008), we explore other methods for computing the NTK leveraging the structure of NNs used 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 (FCN) and convolutional (CNN) 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 orders of magnitude improvements on all major hardware platforms. We open-source implementations of both methods as JAX function transformations.

2. Algorithms for fast NTK computation

Here we describe our algorithms for efficiently computing the NTK. In §2.1 we cover the preliminaries, such as introducing notation (§2.1.1) and recalling the computational complexities of basic AD building blocks like Jacobian-vector products (JVP) and vector-Jacobian products (VJP) (§2.1.2), as well as the cost of evaluating the Jacobian matrix (§2.1.3).

In §2.2 we use the above building blocks to describe the computational complexity of the baseline approach to computing the NTK that is used in most (likely all) prior works.

In §2.3 and §2.4 we present our two algorithms that each allow to speed-up the computation by orders of magnitude in different ways.

2.1. Preliminaries

2.1.1. Notation

Consider a NN $f(\theta, x) \in \mathbb{R}^{\mathbf{O}}$ with $\mathbf{O}$ outputs (logits) per input x and a total number $\mathbf{P}$ of trainable parameters $\theta = \text{vec}\left[\theta^0, \ldots, \theta^{\mathbf{L}}\right]$, with each θ^l of size $\mathbf{P}^l$, $\mathbf{P} = \sum_{l=0}^{\mathbf{L}} \mathbf{P}^l$. Also assume the network has $\mathbf{K}$ intermediate pre-activations y^k of size $\mathbf{Y}^k$ each, $\mathbf{Y} = \sum_{k=1}^{\mathbf{K}} \mathbf{Y}^k$ (see Fig. 5 and Fig. 6). The NTK is

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

We denote $\mathbf{FP}$ to be the (time or memory, depending on the context) cost of a single forward pass $f(\theta, x)$. For memory, we exclude the cost of storing all $\mathbf{P}$ weights in memory, but rather define it to be the cost of evaluating f one JAX (Bradbury et al., 2018) primitive y^k at a time, amounting to no more than $\mathcal{O}\left(\max_l \mathbf{P}^l + \max_k \mathbf{Y}^k\right)$, which we denote as simply $\mathbf{P}^l + \mathbf{Y}^k$ for brevity.[1] Finally, we will consider x_1 and x_2 to be batches of $\mathbf{N}$ inputs each, in which case the NTK will be an $\mathbf{NO} \times \mathbf{NO}$ matrix.

2.1.2. Jacobian-vector products (JVP) and vector-Jacobian products (VJP)

Following Maclaurin et al. we define

$$\text{JVP}_{(f,\theta,x)} : \theta_t \in \mathbb{R}^{\mathbf{P}} \mapsto \frac{\partial f(\theta, x)}{\partial \theta}\theta_t \in \mathbb{R}^{\mathbf{O}}; \quad \text{VJP}_{(f,\theta,x)} : f_c \in \mathbb{R}^{\mathbf{O}} \mapsto \frac{\partial f(\theta, x)}{\partial \theta}^T f_c \in \mathbb{R}^{\mathbf{P}}. \tag{3}$$

The time cost of both is comparable to $\mathbf{FP}$ (see §H and Griewank and Walther (2008)). The memory cost of a JVP is $\mathbf{FP}$ as well, while the memory cost of a VJP is generally $\mathbf{Y} + \mathbf{P}$, since it requires storing all $\mathbf{K}$ intermediate pre-activations for efficient backprop and all $\mathbf{L}$ output cotangents. However, for the purpose of computing the NTK, we never need to store the whole Jacobian $\partial f / \partial \theta$, but only individual cotangents like $\partial f / \partial \theta^l$ to compute the sum in Eq. (2). Hence we consider VJP to cost $\mathbf{Y} + \mathbf{P}^l$ memory. Finally, for a batch of $\mathbf{N}$ inputs x,

JVP and VJP cost $\mathbf{N}\left[\mathbf{FP}\right]$ time; $\mathbf{N}\left[\mathbf{FP}\right] + \mathbf{P}$ and $\mathbf{N}\left[\mathbf{Y} + \mathbf{P}^l\right] + \mathbf{P}$ memory respectively.

2.1.3. Jacobian

For NNs, the Jacobian $\partial f / \partial \theta$ is most often computed via $\mathbf{O}$ VJP calls on rows of the identity matrix $I_{\mathbf{O}}$, i.e. costs $\mathbf{O}\left[\text{VJP}\right]$ time and memory less parameters and pre-activations that can be reused across VJPs. Jacobian costs $\mathbf{NO}\left[\mathbf{FP}\right]$ time; $\mathbf{NO}\left[\mathbf{Y}^k + \mathbf{P}^l\right] + \mathbf{NY} + \mathbf{P}$ memory.

2.2. Jacobian contraction – the baseline

This baseline method of evaluating the NTK consists in computing the Jacobians $\partial f / \partial \theta$ and contracting them as in Eq. (2). The contraction costs $\mathbf{N}^2\mathbf{O}^2\mathbf{P}$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOP}^l$ memory. Adding up the cost of computing the Jacobian $\partial f / \partial \theta$ (§2.1.3) we arrive at

Jacobian contraction: $\mathbf{NO}\left[\mathbf{FP}\right] + \mathbf{N}^2\mathbf{O}^2\mathbf{P}$ time; $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\left[\mathbf{Y}^k + \mathbf{P}^l\right] + \mathbf{NY} + \mathbf{P}$ mem.

1. To declutter notation throughout this work, in time and memory complexity expressions, we (1) omit the $\mathcal{O}$ symbol, and (2) imply taking the maximum over any free index.

2.3. NTK-vector products – our first contribution

Consider the NTK-vector product function: $\Theta\text{VP} : v \in \mathbb{R}^{\mathbf{O}} \mapsto \Theta_\theta v \in \mathbb{R}^{\mathbf{O}}$. Applying it to $\mathbf{O}$ columns of the identity matrix $I_{\mathbf{O}}$ allows to compute the NTK, i.e. $\Theta_\theta I_{\mathbf{O}} = \Theta_\theta$. Expand $\Theta\text{VP}(v) = \Theta_\theta v$ as

$$\frac{\partial f(\theta, x_1)}{\partial \theta} \frac{\partial f(\theta, x_2)}{\partial \theta}^T v = \frac{\partial f(\theta, x_1)}{\partial \theta} \text{VJP}_{(f,\theta,x_2)}(v) = \text{JVP}_{(f,\theta,x_1)}\left[\text{VJP}_{(f,\theta,x_2)}(v)\right], \tag{4}$$

where we have observed that the NTK-vector product can be expressed as a composition of a JVP and a VJP. The cost of computing Θ_θ is then equivalent to the cost of Jacobian, since it consists of $\mathbf{O}$ VJPs followed by $\mathbf{O}$ (cheaper) JVPs, therefore $\mathbf{O}\,[\mathbf{FP}]$ time and $\mathbf{O}\left[\mathbf{Y}^k + \mathbf{P}^l\right] + \mathbf{Y} + \mathbf{P}$ memory. In the batched setting Eq. (4) is repeated for each pair of inputs, and therefore time increases by a factor of $\mathbf{N}^2$ to become $\mathbf{N}^2\mathbf{O}\,[\mathbf{FP}]$. However, the memory cost grows only linearly in $\mathbf{N}$ (except for the cost of storing the NTK of size $\mathbf{N}^2\mathbf{O}^2$), since intermediate pre-activations and tangents/cotangents necessary to compute the JVP and VJP can be computed for each batch x_1 and x_2 separately, and then reused for every pairwise combination. Therefore memory cost is equivalent to Jacobian, and we arrive at

NTK-vector products cost $\mathbf{N}^2\mathbf{O}\,[\mathbf{FP}]$ time; $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\left[\mathbf{Y}^k + \mathbf{P}^l\right] + \mathbf{NY} + \mathbf{P}$ memory.

2.4. Structured derivatives – our second contribution

Rewrite Θ_θ from Eq. (2) using the chain rule and pre-activation y notation:

$$\Theta_\theta = \sum_{l,k_1,k_2} \left(\frac{\partial f_1}{\partial y_1^{k_1}} \frac{\partial y_1^{k_1}}{\partial \theta^l}\right) \left(\frac{\partial f_2}{\partial y_2^{k_2}} \frac{\partial y_2^{k_2}}{\partial \theta^l}\right)^T = \sum_{l,k_1,k_2} \frac{\partial f_1}{\partial y_1^{k_1}} \frac{\partial y_1^{k_1}}{\partial \theta^l} \frac{\partial y_2^{k_2}}{\partial \theta^l}^T \frac{\partial f_2}{\partial y_2^{k_2}}^T, \tag{5}$$

where $i \in \{1, 2\}$, $f_i = f(\theta, x_i)$, and we only consider $\partial y_i^{k_i}/\partial \theta^l$ if θ^l is a direct input to $y_i^{k_i}$.

Both Jacobian contraction and NTK-vector products perform this sum of contractions, albeit implicitly via VJPs and JVPs, without explicit instantiation of primitive Jacobians $\partial y/\partial \theta$. However, while VJPs and JVPs themselves are guaranteed to be computationally optimal, higher order computations like their composition (NTK-vector products) or contraction (Jacobian contraction) are not. The idea of Structured derivatives is to design rules for efficient computation of such contractions, similarly to AD rules for JVPs and VJPs.

Specifically, our rules identify a few simple types of structure (e.g. block diagonal, constant-block diagonal, tiling) in $\partial y^{k_i}/\partial \theta^l$, that allow us to simplify the contraction in Eq. (5). 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 (much) smaller subarrays and modifying the contraction. In §I we provide specific descriptions of our rules and their impact on the computational complexity of Eq. (5). We denote the generic contraction cost as $\mathbf{NOG}$, and show in §3 that it is asymptotically faster than Jacobian contraction for matrix multiplications and convolutions. For a simple example on FCNs, see §L.4, applying a more general Constant block-diagonal rule (§I.3).

The remaining cost to compute the factors $\partial f_i/\partial y_i^{k_i}$, and $\partial y_i^{k_i}/\partial \theta^l$ also depends on the specific pair of primitives $y_1^{k_1}$ and $y_2^{k_2}$, but is generally similar to the cost of Jacobian except for (1) we don't need to compute and store $\mathbf{NO}$ final weight space cotangents $\partial f_i/\partial \theta^l$, but (2) we do have to process $\mathbf{N}$ small subarrays of primitive Jacobians $\partial y_i^{k_i}/\partial \theta^l$, which we consider

Method	Time	Memory	Use when
Jacobian contraction	$\mathbf{N}\ \mathbf{O}\,[\mathbf{FP}] + \mathbf{N}^2\mathbf{O}^2\mathbf{P}$	$\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\,[\mathbf{Y}^k + \mathbf{P}^l] + \mathbf{NY} + \mathbf{P}$	$\mathbf{P} < \mathbf{Y}$, small $\mathbf{O}$
NTK-vector products	$\mathbf{N}^2\mathbf{O}\,[\mathbf{FP}]$	$\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\,[\mathbf{Y}^k + \mathbf{P}^l] + \mathbf{NY} + \mathbf{P}$	$\mathbf{FP} < \mathbf{OP}$, large $\mathbf{O}$, small $\mathbf{N}$
Structured derivatives	$\mathbf{N}\ \mathbf{O}\,[\mathbf{FP}] + \mathbf{N}\ \mathbf{O}\ \mathbf{G} + \mathbf{N}\,[\mathbf{J} - \mathbf{OP}]$	$\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOY}^k + \mathbf{NJ}_l^k + \mathbf{NY} + \mathbf{P}$	$\mathbf{FP} > \mathbf{OP}$, large $\mathbf{O}$, large $\mathbf{N}$

Table 1: **Generic NTK computation cost.** NTK-vector products trade-off contractions for more **FP**. Structured derivatives usually save both time and memory.

Method	Time	Memory	Use when
Jacobian contraction	$\mathbf{N}\ \mathbf{O}\,[\mathbf{LDFW}^2 + \mathbf{OW}] + \mathbf{N}^2\mathbf{O}^2\,[\mathbf{LFW}^2 + \mathbf{OW}]$	$\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\,[\mathbf{DW} + \mathbf{FW}^2 + \mathbf{OW}] + \mathbf{N}\,[\mathbf{LDW}] + [\mathbf{LFW}^2 + \mathbf{OW}^2]$	$\mathbf{D} > \mathbf{OW}$
NTK-vector products	$\mathbf{N}^2\mathbf{O}\,[\mathbf{LDFW}^2 + \mathbf{OW}]$	$\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\,[\mathbf{DW} + \mathbf{FW}^2 + \mathbf{OW}] + \mathbf{N}\,[\mathbf{LDW}] + [\mathbf{LFW}^2 + \mathbf{OW}^2]$	$\mathbf{N} = 1$
Structured derivatives	$\mathbf{N}\ \mathbf{O}\,[\mathbf{LDFW}^2 + \mathbf{OW}] + \mathbf{N}^2\mathbf{O}^2\left[\mathbf{L}\min\left(\mathbf{FW}^2, \mathbf{DW} + \frac{\mathbf{DFW}^2}{\mathbf{O}}, \mathbf{DW} + \frac{\mathbf{D}^2\mathbf{W}}{\mathbf{O}} + \frac{\mathbf{D}^2\mathbf{FW}}{\mathbf{O}^2}\right) + \mathbf{O}\right]$	$\mathbf{N}^2\mathbf{O}^2 + \mathbf{NO}\,[\mathbf{DW}] + \mathbf{NDFW} + \mathbf{N}\,[\mathbf{LDW}] + [\mathbf{LFW}^2 + \mathbf{OW}^2]$	$\mathbf{D} < \mathbf{OW}$

Table 2: **CNN NTK computation cost.** Structured derivatives reduce time complexity, and have lower memory cost if $\mathbf{D} < \mathbf{OW}$, which is a common setting.

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

Table 3: **FCN NTK computation cost.** NTK-vector products allow a reduction of the time complexity, while Structured derivatives reduce both time and memory complexity. For brevity $\mathbf{O} = \mathcal{O}(\mathbf{LW})$ is assumed in this table (see §M and Table 7 for no assumption).

to cost $\mathbf{J}_l^{k_i}$. We summarize generic cost estimates below and in Table 1, and show in §3 that they end up beneficial in most practical settings. In conclusion, Structured derivatives cost $\boxed{\mathbf{NO}\,[\mathbf{FP}] + \mathbf{NOG} + \mathbf{N}\,[\mathbf{J} - \mathbf{OP}] \text{ time; } \mathbf{N}^2\mathbf{O}^2 + \mathbf{NOY}^k + \mathbf{NJ}_l^k + \mathbf{NY} + \mathbf{P} \text{ memory.}}$

3. Applications and benchmarks

We consider $\mathbf{K} = \mathbf{L}$-layer CNNs with channel count $\mathbf{W}$, pixel count $\mathbf{D}$, filter size $\mathbf{F}$, and global average pooling before the top FC layer (see Fig. 5). Plugging $\mathbf{P}^l = \mathbf{FW}^2$ ($\mathbf{OW}$ for $l = \mathbf{L}$), $\mathbf{Y}^k = \mathbf{DW}$ ($\mathbf{O}$ for $k = \mathbf{K}$), $\mathbf{FP} = \mathbf{LDFW}^2 + \mathbf{OW}$, $\mathbf{J}_l^k = \mathbf{DFW}$, and $\mathbf{G}$ from Table 8 (convolutions and matrix multiplications have the Constant block-diagonal structure – see §I.3) we arrive at Table 2. For FCNs we simply put $\mathbf{D} = \mathbf{F} = 1$, and obtain Table 3 (and Fig. 1, Fig. 3). Notably, in both cases Structured derivatives are asymptotically better than Jacobian contraction in time and memory (the latter under a mild condition of $\mathbf{D} \leq \mathbf{OW}$).

Finally, we evaluate our methods in the wild, and confirm computational benefits on full ImageNet models in Fig. 2 (ResNets, He et al. (2016)) and Fig. 4 (WideResNets, Zagoruyko and Komodakis (2016); Vision Transformers and hybrids Dosovitskiy et al. (2021); Steiner et al. (2021); and MLP-Mixers Tolstikhin et al. (2021)). Computing the full $\mathbf{O} \times \mathbf{O} = 1000 \times 1000$ NTK in many settings is only possible with Structured derivatives.

4. API

We release all implementations as a single function decorator accepting an `implementation` argument, that can be also set to `AUTO`, i.e. a FLOPs cost analysis will be run once at compilation time, and the best implementation will be selected automatically (§C, §D).

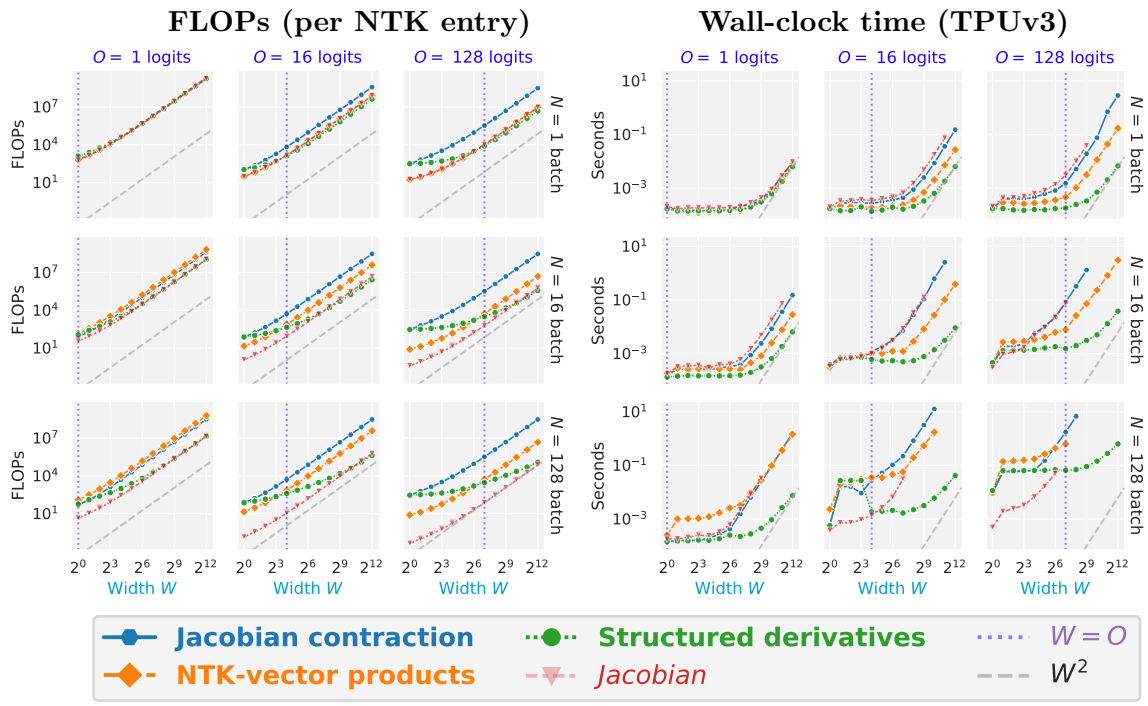

Figure 1: **FLOPs (left) and wall-clock time (right) of computing the NTK for a 10-layer ReLU FCN.** As predicted by Table 3, our methods almost always outperform Jacobian contraction, allowing orders of magnitude speed-ups and memory improvements (missing points are out-of-memory).

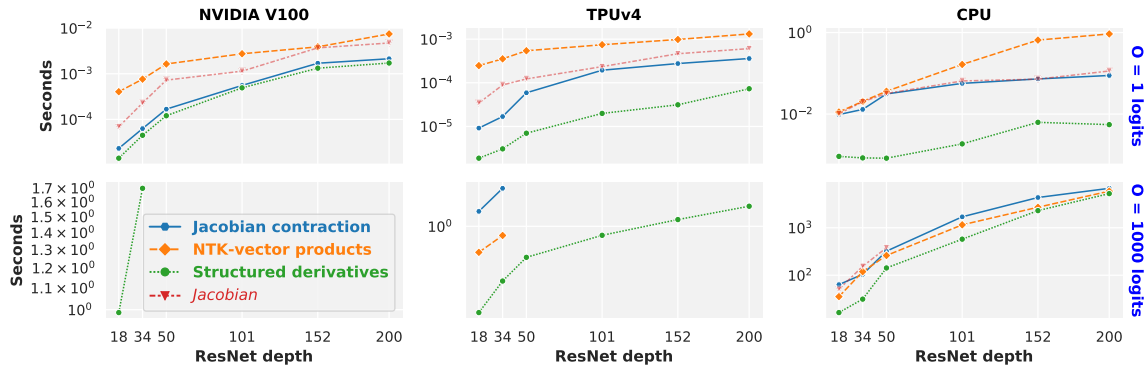

Figure 2: **Wall-clock time cost of computing an NTK for several ResNet sizes on a pair of ImageNet inputs**. Structured derivatives allow the NTK to be computed faster and for larger models (see bottom row – missing are out-of-memory). NTK-vector products, as predicted by Table 1, are advantageous for large **O** (bottom row), but also scale worse with **FP** than other methods, which is especially noticeable in CNNs. See Fig. 4 for more ImageNet models, and §O for experimental details, and §N for analysis of CNN NTK complexity analysis.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

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*, pages 8141–8150. Curran Associates, Inc., 2019a.

Sanjeev Arora, Simon S Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. *arXiv preprint arXiv:1901.08584*, 2019b.

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.

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*, pages 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.

Simon S Du, Kangcheng Hou, Russ R Salakhutdinov, Barnabas Poczos, Ruosong Wang, and Keyulu Xu. Graph neural tangent kernel: Fusing graph neural networks with graph kernels. In *Advances in Neural Information Processing Systems*. 2019.

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

Stanislav Fort, Gintare Karolina Dziugaite, Mansheej Paul, Sepideh Kharaghani, Daniel M Roy, and Surya Ganguli. Deep learning versus kernel learning: an empirical study of loss landscape geometry and the time evolution of the neural tangent kernel. *arXiv preprint arXiv:2010.15110*, 2020.

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.

Andreas Griewank and Andrea Walther. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, second edition, 2008. doi: 10.1137/1.9780898717761. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898717761.

Boris Hanin and Mihai Nica. Finite depth and width corrections to the neural tangent kernel. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SJgndT4KwB.

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, editors, *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*, pages 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.

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

Jiri Hron, Yasaman Bahri, Roman Novak, Jeffrey Pennington, and Jascha Sohl-Dickstein. Exact posterior distributions of wide bayesian neural networks, 2020a.

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*, 2020b.

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

Mohammad Emtiyaz E Khan, Alexander Immer, Ehsan Abedi, and Maciej Korzepa. Approximate inference turns deep networks into gaussian processes. In *Advances in neural information processing systems*, 2019.

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.

Jaehoon Lee, Samuel S Schoenholz, Jeffrey Pennington, Ben Adlam, Lechao Xiao, Roman Novak, and Jascha Sohl-Dickstein. Finite versus infinite neural networks: an empirical study. 2020.

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.

Herman Müntz. Solution directe de l'équation séculaire et de quelques problèmes analogues transcendants. *C. R. Acad. Sci. Paris*, 156:43–46, 1913.

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.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

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

Sho Yaida. Non-Gaussian processes and neural networks at finite widths. In *Mathematical and Scientific Machine Learning Conference*, 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.

Greg Yang. Tensor programs ii: Neural tangent kernel for any architecture. *arXiv preprint arXiv:2006.14548*, 2020.

Greg Yang, Jeffrey Pennington, Vinay Rao, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. A mean field theory of batch normalization. In *International Conference on Learning Representations*, 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.

Appendix

A. Additional figures

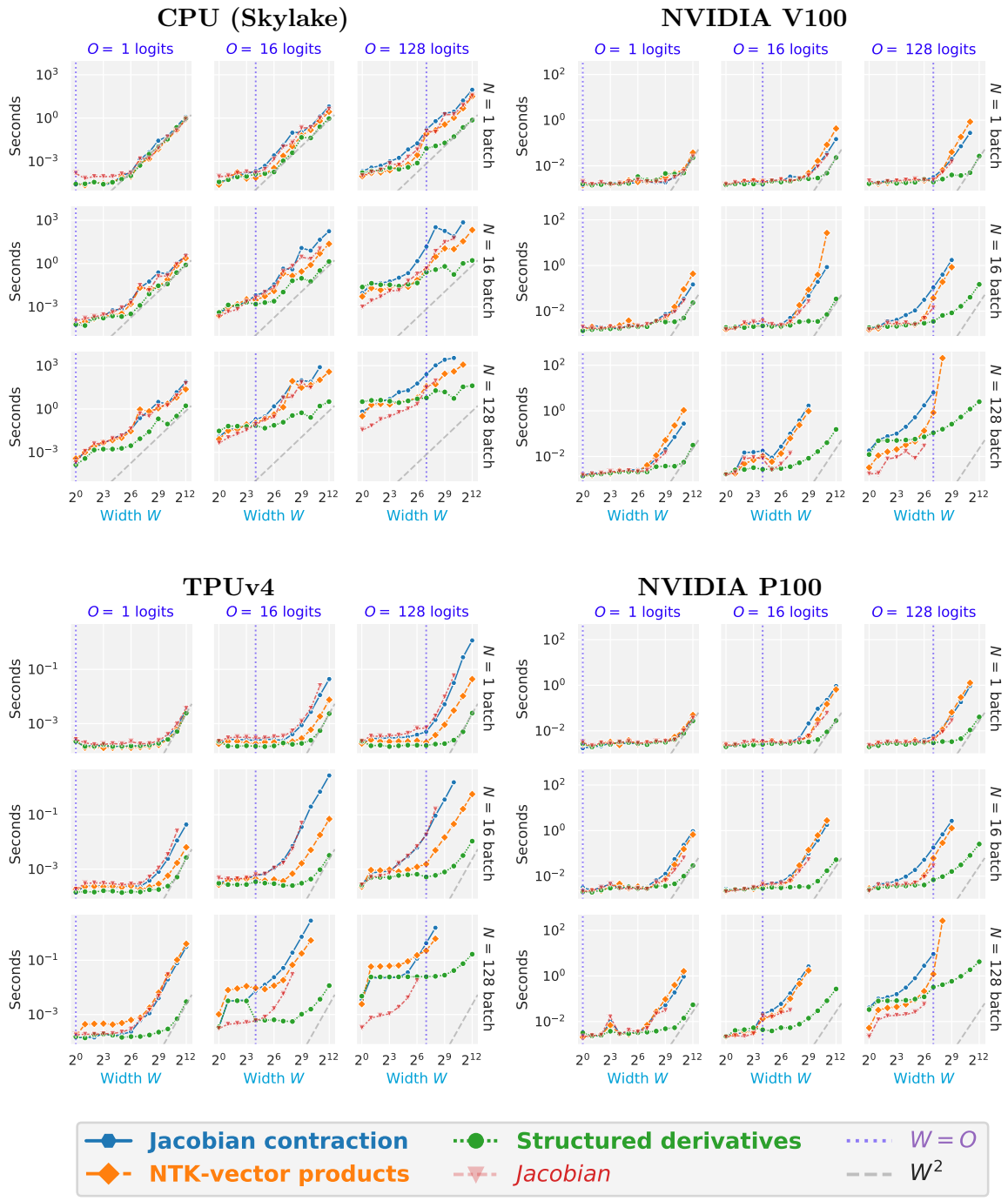

Figure 3: **Wall-clock time of computing the NTK of a 10-layer ReLU FCN on different platforms.** In all settings, Structured derivatives allow orders of magnitude improvement in wall-clock time and memory (missing points indicate out-of-memory error). See Fig. 1 for FLOPs, **TPUv3** platform, and more discussion. See §O for details.

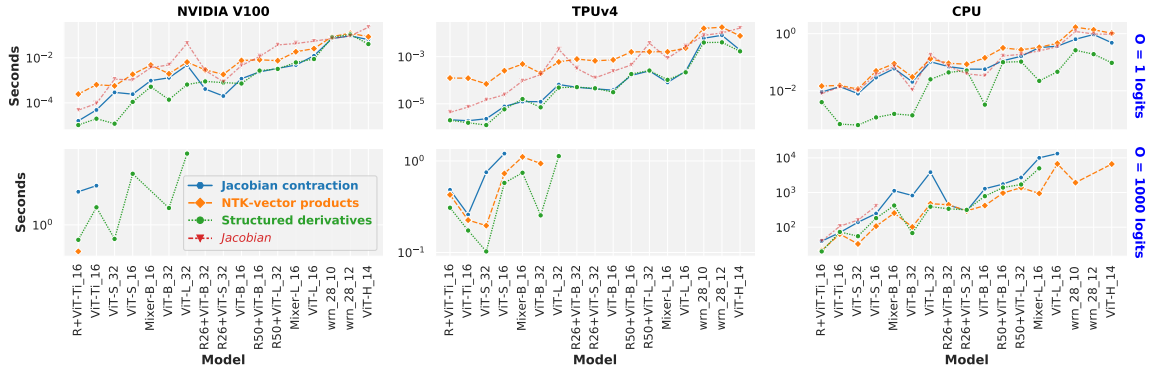

Figure 4: **Wall-clock time per input pair of computing NTK on various ImageNet models** like Vision Tansformers and hybrids (Dosovitskiy et al., 2021; Steiner et al., 2021), WideResNets (Zagoruyko and Komodakis, 2016) and MLP-Mixers (Tolstikhin et al., 2021). Structured derivatives generally allow fastest computation, but also are able to process more models due to lower memory requirements (lower left; missing points indicate out-of-memory error). For the case of single output logit $\mathbf{O} = 1$ (top row), NTK-vector products are generally detrimental due to a costly forward pass **FP** relative to the size of parameters $\mathbf{P}$ (i.e. a lot of weight sharing; see Table 1). However, since NTK-vector products scale better than other methods with output size, for $\mathbf{O} = 1000$ (bottom row), they perform comparably or better than other methods. Finally, we remark that Jacobian not only runs out of memory faster, but can also take more time to compute. We conjecture that due to a larger memory footprint, XLA can sometimes perform optimizations that trade off speed for memory, and therefore compute the Jacobian in a less optimal way than if it had more memory available. See Fig. 2 for ResNets, and §O for details.

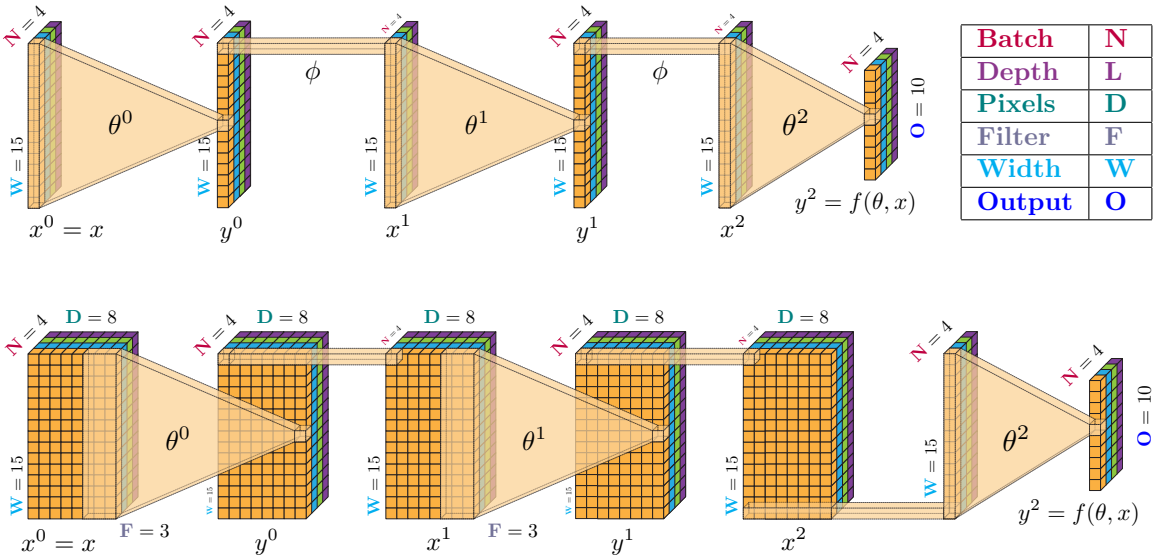

Figure 5: **Notation used in §2.1.1 (FCN, top) and §N (CNN, bottom).** For FCN, $\mathbf{D} = \mathbf{F} = 1$. For CNN, $\mathbf{D} = 8$, $\mathbf{F} = 3$, and the penultimate layer is global average pooling.

B. Glossary

- $\mathbf{N}$ - batch size of inputs x to the NN $f(\theta, x)$. In a more general setting (§I), the number of functions $f(\theta)$.

  n - batch indices ranging from 1 to $\mathbf{N}$.

- $\mathbf{O}$ - output size (e.g. number of logits) of $f(\theta, x)$ for a single ($\mathbf{N} = 1$) input x.

- The NTK matrix has shape $\mathbf{NO} \times \mathbf{NO}$.

- $\mathbf{W}$ - width of an FCN, or number of channels of a CNN. Individual inputs x are usually assumed to have the same size / number of channels.

- $\mathbf{L}$ - depth of the network, number of layers. In a more general setting, number of trainable parameter matrices, that are used in a possibly different number of subexpressions in the network.

  l - depth index ranging from 0 to $\mathbf{L}$.

- $\mathbf{K}$ - number of subexpressions (primitives, nodes in the computation graph) of the network $f(\theta, x)$. For NNs without weight sharing, $\mathbf{K} = \mathbf{L}$.

  k - subexpression index ranging from 1 to $\mathbf{K}$.

- $\mathbf{D}$ - total number of pixels (e.g. 1024 for a 32×32 image; 1 for an FCN) in an input and every intermediate layer of a CNN (`SAME` or `CIRCULAR` padding is assumed, to consider the spatial size unchanged from layer to layer).

- $\mathbf{F}$ - total filter size (e.g. 9 for a 3×3 filter; 1 for an FCN) in a convolutional filter of a CNN (no striding and dilation is assumed for simplicity).

- $\mathbf{Y}$ - total size of a pre-activation / primitive / subexpression y (e.g. $\mathbf{Y} = \mathbf{DW}$ for a layer with $\mathbf{D}$ pixels and $\mathbf{W}$ channels; $\mathbf{Y} = \mathbf{W}$ for FCN). Depending on the context, can represent size of a single or particular pre-activation in the network, or the size of all pre-activations together.

- $\mathbf{C}$ - in §I, the size of the axis along which a subexpression derivative $\partial y / \partial \theta$ admits certain structure ($\mathbf{C}$ can often be equal to $\mathbf{Y}$ or a significant fraction of it, e.g. $\mathbf{W}$).

  c - index along the structured axis, ranging from 1 to $\mathbf{C}$.

- $\mathbf{P}$ - total size of trainable parameters. Depending on the context, can represent the size of a particular weight matrix θ^l in some layer l (e.g. $\mathbf{W}^2$ for width-$\mathbf{W}$ FCN), or the size of all parameters in the network.

- **FP** - forward pass, cost (time or memory, depending on the context) of evaluating $f(\theta, x)$ on a single ($\mathbf{N} = 1$) input x.

- If a variable is present in complexity analysis with an index such as k or l, it is considered to be the maximum over that index, e.g. $\mathbf{Y}^k = \max_k \mathbf{Y}^k$. This is used in Table 1, Table 2, and Table 3.

- $\mathbf{J}_l^k$ is the cost of evaluating a single primitive Jacobian $\partial y^k / \partial \theta^l$, given the structure present in y^k according to §I.

C. 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 function accepts 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 θ. Inputs x, parameters θ, 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 JAX reverse-mode AD interpreter. On a high level, the algorithm performs the sum in Eq. (5). Each summand is a contraction of 4 factors: $\partial f_1/\partial y_1, \partial y_1/\partial \theta, \partial y_2/\partial \theta, \partial f_2/\partial y_2$.

First, we linearize f to obtain a computational graph constructed out of a limited set (54,[2] see Table 6) of linear primitives $y^1, \ldots, y^{\mathbf{K}}$. Then, we can obtain two factors $\partial f_1/\partial y_1, \partial f_2/\partial y_2$ as part of a backward pass almost identical to `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 (§I.8); for a given pair of primitives, these rules allow us to analytically simplify the contraction and avoid explicitly instantiating the derivatives.

Finally, 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.

D. Leveraging JAX design for efficient NTK computation

At the time of writing, Tensorflow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019) are more widely used than JAX (Bradbury et al., 2018). However, certain JAX features and design choices made it much more suitable, if not indispensable, for our project:

1. Structured derivatives require manual implementation of structure rules for different primitives in the computational graph of a function $f(\theta, x)$. JAX has a small primitive set of about 131 primitives, while PyTorch and Tensorflow have more than 400 (Frostig et al., 2021, Section 2). Further, by leveraging `jax.linearize`, we reduce our task to implementing structure rules for only *linear* primitives, of which JAX has only 54.[3] To our knowledge neither PyTorch nor Tensorflow have an equivalent transformation, which makes JAX a natural choice due to the very concise set of primitives that we need to handle (Table 6).

2. 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).
3. This follows from the fact that the NTK of a function is equal to the NTK of the linearized function at the same primal parameters θ. See also (Frostig et al., 2021, Section 1) for how JAX uses the same insight to not implement all 131 VJP rules, but only implement 54 transpose rules for reverse mode AD.

2. NTK-vector products critically rely on forward mode AD (JVP), and Structured derivatives also use it (albeit it's not crucial; see §J). At the time of writing, PyTorch does not implement an efficient forward mode AD.

3. Structured derivatives rely crucially on the ability to traverse the computation graph to rewrite contractions using our substitution rules. JAX provides a highly-convenient graph representation in the form of a Jaxpr, as well as tooling and documentation for writing custom Jaxpr interpreters.

4. All implementations (even Jacobian contraction) rely heavily on `jax.vmap` (and to our knowledge, in many cases, it is indispensable). While PyTorch has released a prototype of `vmap` in May 2021, it was not available when we started this project.

For researchers interested in interfacing with our library, we recommend looking into tools facilitating data exchange between different ML frameworks, such as DLPack and Jax2TF. See §C for more implementation details.

E. Comparison with specific related works

The finite width NTK 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. (2019a, 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 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 and 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 and 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. Remarking its prohibitive cost, 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,[4] which impacts the quality of the respective trainability/generalization proxy. In contrast, in this work we can compute the full 1000×1000 NTK on the same models (1000 classes), 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 due to better infinite width/linearization approximation or higher-quality trainability/generalization proxy respectively, which makes our work especially relevant to modern research.

F. Applications with a limited compute budget

While our methods allow to dramatically speed-up the computation of NTK, all of them still scale as $\mathbf{N}^2\mathbf{O}^2$ for both time and memory, which can be intractable for large datasets and/or large outputs.

Here we present several settings in which our proposed methods still provide substantial time and memory savings, even when instantiating the entire $\mathbf{NO} \times \mathbf{NO}$ NTK is not feasible or not necessary.

- **NTK-vector products**. In many applications one only requires computing the NTK-vector product linear map

$$\Theta_\theta : v \in \mathbb{R}^{\mathbf{NO}} \mapsto \Theta_\theta v \in \mathbf{NO}, \tag{6}$$

without computing the entire NTK matrix Θ_θ. One common setting is using the power iteration method (Müntz, 1913) to compute NTK condition number and hence trainability of the respective NN (Lee et al., 2019; Chen et al., 2021a,b). Another setting is using conjugate gradients to compute $\Theta_\theta^{-1}\mathcal{Y}$ when doing kernel ridge regression with the NTK (Jacot et al., 2018; Lee et al., 2019; Zhou et al., 2021).

Eq. (6) is the same map as the one we considered in §2.3, and naturally, NTK-vector products can provide a substantial speed-up over Jacobian contraction in this setting. Precisely, a straightforward application of Jacobian contraction yields

$$\underbrace{\Theta_\theta v}_{\mathbf{NO}\times\mathbf{1}} = \underbrace{\frac{\partial f(\theta, x_1)}{\partial \theta}}_{\mathbf{NO}\times\mathbf{P}} \underbrace{\frac{\partial f(\theta, x_2)}{\partial \theta}^T}_{\mathbf{P}\times\mathbf{NO}} \underbrace{v}_{\mathbf{NO}\times 1}. \tag{7}$$

Combined with the cost of computing the weight space cotangents $\partial f / \partial \theta$, such evaluation costs $\mathbf{NO}\,[\mathbf{FP}]$ time, i.e. the cost of instantiating the entire Jacobian. Alternatively, one could store the entire Jacobians of sizes $\mathbf{NOP}$ in memory, and compute a single NTK-vector product in $\mathbf{NOP}$ time.

4. 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).

In contrast, NTK-vector products allow to compute an NTK-vector product at a cost asymptotically equivalent to a single VJP call (§2.3), i.e. $\mathbf{N}\,[\mathbf{FP}]$, $\mathbf{O}$ times faster than Jacobian contraction without caching. With caching, fastest method will vary based on the cost of $[\mathbf{FP}]$ relative to $\mathbf{OP}$, as discussed in §**??**, but NTK-vector products will remain substantially more memory-efficient due to not caching the entire $\mathbf{NOP}$ Jacobians.

- **Batching.** In many applications it suffices to compute the NTK over small batches of the data. For example Dauphin and Schoenholz (2019); Chen et al. (2021a,b) estimate the conditioning by computing an approximation to the NTK on $\mathbf{N}$ equal to 128, 32, and 48 examples respectively. Similarly, Zhou et al. (2021) use a small batch size of $\mathbf{N} = 25$ to meta-learn the network parameters by replacing the inner SGD training loop with NTK regression.

- **Pseudo-NTK.** Many applications (§E) compute a pseudo-NTK of size $\mathbf{N} \times \mathbf{N}$, which is commonly equal to one of its $\mathbf{O}$ diagonal blocks, or to the mean of all $\mathbf{O}$ blocks. The reason for considering such approximation is that in the infinite width limit, off-diagonal entries often converge to zero, and for wide-enough networks this approximation can be justified. Compute-wise, these approximations are equivalent to having $\mathbf{O} = 1$. While an important contribution of our work is to enable computing the full $\mathbf{NO} \times \mathbf{NO}$ NTK quickly, if necessary, Structured derivatives can be combined with the $\mathbf{O} = 1$ approximations, and still provide an asymptotic speed-up and memory savings relative to prior works.

G. Finite and infinite width NTK

In this work we focus on computing the finite width NTK $\Theta_\theta\left(x_1, x_2\right)$, defined in Eq. (1), that we repeat below with an addition of a batch size $\mathbf{N}$:

$$\textbf{F-NTK (finite width):} \quad \underbrace{\Theta_\theta(x_1, x_2)}_{\mathbf{NO}\times\mathbf{NO}} := \underbrace{\left[\partial f(\theta, x_1)/\partial\theta\right]}_{\mathbf{NO}\times\mathbf{P}} \underbrace{\left[\partial f(\theta, x_2)/\partial\theta\right]^T}_{\mathbf{P}\times\mathbf{NO}}. \tag{8}$$

Another highly important object in deep learning theory is the *infinite width* NTK $\Theta\left(x_1, x_2\right)$, introduced in the seminal work of Jacot et al. (2018):

$$\textbf{I-NTK (infinite width):} \quad \underbrace{\Theta(x_1, x_2)}_{\mathbf{NO}\times\mathbf{NO}} := \lim_{\mathbf{W}\to\infty} \mathbb{E}_{\theta\sim\mathcal{N}(\mathbf{0}, I_{\mathbf{P}})} \left[\underbrace{\Theta_\theta\left(x_1, x_2\right)}_{\mathbf{NO}\times\mathbf{NO}}\right]. \tag{9}$$

A natural question to ask is what are the similarities and differences of F- and I-NTK, when is one more applicable than the other, and what are their implementation and compute costs.

Applications. At a high level, F-NTK describes the local/linearized behavior of the finite width NN $f\left(\theta, x\right)$ (Lee et al., 2019). In contrast, I-NTK is an approximation that is exact only in the infinite width $\mathbf{W}$ limit, and only at initialization ($\theta \sim \mathcal{N}\left(\mathbf{0}, I_{\mathbf{P}}\right)$). As such, the resulting I-NTK has no notion of width $\mathbf{W}$, parameters θ, and cannot be computed during draining, or in a transfer or meta-learning setting, where the parameters θ are

updated. As a consequence, any application to finite width networks (§E) is better served by the F-NTK, and often impossible with the I-NTK.

In contrast, I-NTK describes the behavior of an infinite ensemble of infinitely wide NNs. In certain settings this can be desirable, such as when studying the inductive bias of certain NN architectures (Xiao et al., 2020) or uncertainty (Adlam et al., 2020), marginalizing away the dependence on specific parameters θ. However, care should be taken when applying I-NTK findings to the finite width realm, since many works have demonstrated substantial finite width effects that cannot be captured by the I-NTK (Novak et al., 2019; Arora et al., 2019b; Lee et al., 2019; Yaida, 2020; Hanin and Nica, 2020; Lee et al., 2020).

Mathematical scope. Another significant difference between F- and I-NTK is the scope of their definitions in Eq. (8) and Eq. (9) and mathematical tractability.

The F-NTK is well-defined for any differentiable (w.r.t. θ) function f, and our methods are respectively applicable to any differentiable functions. In fact, our work supports any Tangent Kernels (not necessarily "Neural"), and is not specific to NNs at all.

In contrast, the I-NTK requires the function f to have the concept of width $\mathbf{W}$ (that can be meaningfully taken to infinity) to begin with, and further requires f and θ to satisfy many conditions in order for the I-NTK to be well-defined (Yang, 2019). In order for I-NTK to be well-defined *and computable in closed-form*, f needs to be built out of a relatively small, hand-selected number of primitives that admit certain Gaussian integrals to have closed-form solutions. Examples of ubiquitous primitives that *don't* allow a closed-form solution include attention with standard parameterization (Hron et al., 2020b); max-pooling; sigmoid, (log-)softmax, tanh, and many other nonlinearities; various kinds of normalization (Yang et al., 2019); non-trivial weight sharing (Yang, 2020); and many other settings. Going forward, it is unlikely that the I-NTK will scale to the enormous variety of architectures introduced by the research community each year.

Implementation tractability. Above we have demonstrated that the I-NTK is defined for a very small subset of functions admitting the F-NTK. A closed-form solution exists for an even smaller subset. However, even when the I-NTK admits a closed-form solution, it is important to consider the complexity of implementing it.

Our implementation for computing the F-NTK is applicable to any differentiable function f, and requires no extra effort when switching to a different function g. It is similar to JAX's highly-generic function transformations such as `jax.jit` or `jax.vmap`.

In contrast, there is no known way to compute the I-NTK for an arbitrary function f, even if the I-NTK exists in closed form. The best existing solution to date is provided by Novak et al. (2020), which allows to *construct* f out of the limited set of building blocks provided by the authors. However, one cannot compute the I-NTK for a function implemented in a different library such as Flax (Heek et al., 2020), or Haiku (Hennigan et al., 2020), or bare-bone JAX. One would have to re-implement it using the primitives provided by Novak et al. (2020). Further, for a generic architecture, the primitive set is unlikely to be sufficient, and the function will need to be adapted to admit a closed-form I-NTK.

Computational tractability. F-NTK and I-NTK have different time and memory complexities, and a fully general comparison is an interesting direction for future work. Here we provide discussion for deep FCNs and CNNs.

Networks having a fully-connected top ($\mathbf{L}$) readout layer have a constant-block diagonal I-NTK, hence its cost *does not* scale with $\mathbf{O}$. The cost of computing the I-NTK for a deep FCN scales as $\mathbf{N}^2\mathbf{L}$ for time and $\mathbf{N}^2$ for memory. A deep CNN without pooling costs $\mathbf{N}^2\mathbf{DL}$ time and $\mathbf{N}^2\mathbf{D}$ memory (where $\mathbf{D}$ is the total number of pixels in a single input/activation; $\mathbf{D} = 1$ for FCNs). Finally, a deep CNN with pooling, or any other generic architecture that leverages the spatial structure of inputs/activations, costs $\mathbf{N}^2\mathbf{D}^2\mathbf{L}$ time and $\mathbf{N}^2\mathbf{D}^2$ memory. This applies to all models in Fig. 2 and Fig. 4, Graph Neural Networks (Du et al., 2019), and the vast majority of other architectures used in practice.

The quadratic scaling of the I-NTK cost with $\mathbf{D}$ is especially burdensome, since, for example, for ImageNet $\mathbf{D}^2 = 224^4 = 2,517,630,976$. As a result, it would be impossible to evaluate the I-NTK on even a single ($\mathbf{N} = 1$) pair of inputs with a V100 GPU for any model for which we've successfully evaluated the F-NTK in Fig. 2 and Fig. 4.

The F-NTK time and memory only scale linearly with $\mathbf{D}$ (Table 2). However, the F-NTK cost scales with other parameters such as width $\mathbf{W}$ or number of outputs $\mathbf{O}$, and in general the relative F- and I-NTK performance will depend on these parameters. As a rough point of comparison, we consider the cost of evaluating the I-NTK of a 20-layer binary classification ReLU CNN with pooling on a V100 GPU used by Arora et al. (2019b) against the respective F-NTK with $\mathbf{W} = 128$ also used by Arora et al. (2019b, Section B). Arora et al. (2019b) and Novak et al. (2020) report from 0.002 to 0.003 seconds per I-NTK entry on a pair of CIFAR-10 inputs. Using Structured derivatives, we can compute the respective F-NTK entry on same hardware in at most 0.000014 seconds, i.e. at least 100 times faster than the I-NTK. In 0.002 – 0.003 seconds per NTK entry, we can compute the F-NTK on a pair of *ImageNet* inputs (about 50x larger than CIFAR-10) for a *200-layer ResNet* (about 10x deeper than the model above) in Fig. 2 (top left).

Finally, we remark that efficient NTK-vector products without instantiating the entire $\mathbf{NO} \times \mathbf{NO}$ NTK are only possible using the F-NTK (§F).

H. JVP and VJP costs

Here we provide intutition for why JVP and VJP are asymptotically equivalent in time to the forward pass **FP**, as we mentioned in §2.1.2. See (Griewank and Walther, 2008, Section 3) for a rigorous treatment, and the JAX Autodiff Cookbook for a hands-on introduction.

JVP can be computed by traversing a computational graph of the same topology as **FP**, except for primitive nodes in the graph need to be augmented to compute not only the forward pass of the node, but also the JVP of the node (see Fig. 6). Due to identical topology and order of evaluation, asymptotically time and memory costs remain unchanged. However, constructing the augmented nodes in the JVP graph, and their consequent evaluation results in extra time cost proportional to the size of the graph. Therefore in practice JVP costs about $3 \times$ **FP** time and $2 \times$ **FP** memory.

VJP, as a linear function of cotangents f_c, is precisely the transpose of the linear function JVP. As such, it can be computed by traversing the transpose of the JVP graph (Fig. 6, right), with each JVP node replaced by its transposition as well. This results in identical time and memory costs, as long as node transpositions are implemented efficiently. However, their evaluation requires primal outputs y^k (now inputs to the transpose nodes), which is why VJP necessitates an extra **FP** time cost to compute them (hence costlier than

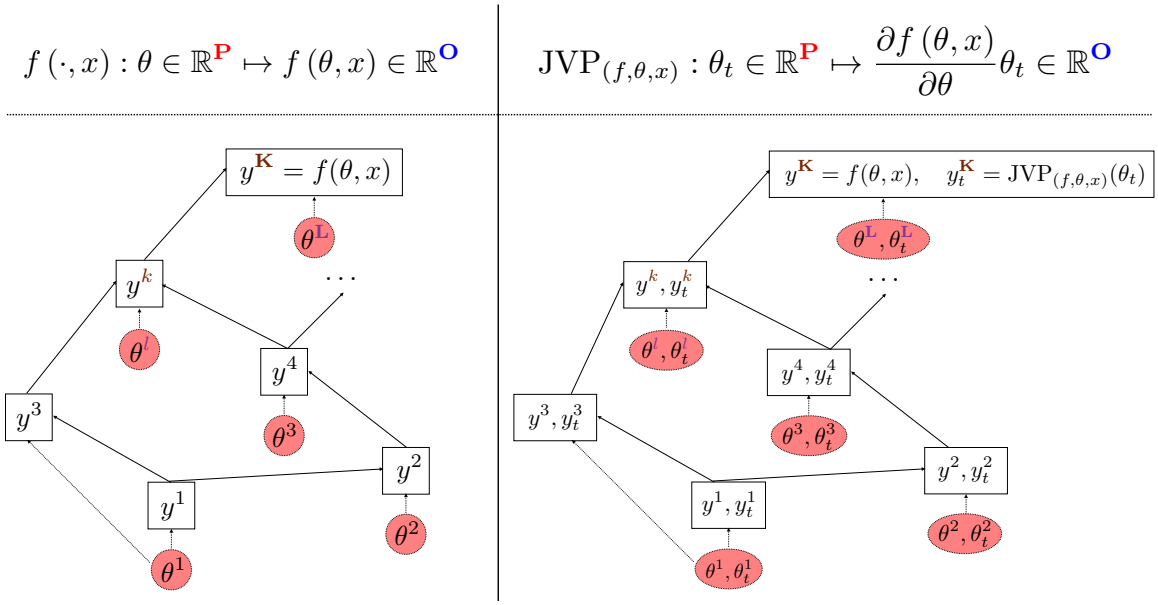

Figure 6: **Visual demonstration for why JVP time and memory costs are asymptotically comparable to the forward pass (FP). Left**: computational graph of the forward pass $f(\theta, x)$. **Right**: computational graph of joint evaluation of the forward pass $f(\theta, x)$ along with $\mathrm{JVP}_{(f,\theta,x)}(\theta_t)$. Each node of the JVP graph accepts both primal and tangent inputs, and returns primal and tangent outputs, but the topology of the graph and order of execution remains identical to **FP**. As long as individual nodes of the JVP graph do not differ significantly in time and memory from the **FP** nodes, time and memory of a JVP ends up asymptotically equivalent to **FP** due to identical graph structure. However, in order to create JVP nodes and evaluate them, the time cost does grow by a factor of about 3 compared to **FP**. See §L.1 for discussion.

JVP, but still inconsequential asymptotically) and extra memory to store them, which can generally increase asymptotic memory requirements.

I. Types of structured derivatives

Here we continue §2.4 and list the types of structures in primitive derivatives $\partial y / \partial \theta$ that allow linear algebra simplifications of the NTK expression. Analysis from the following subsections is summarized in Table 4.

I.1. No structure

We first consider the default cost of evaluating a single summand in Eq. (5), denoting individual matrix shapes underneath:

$$\Theta_\theta^{l,k_1,k_2}(f_1, f_2) := \frac{\partial f_1}{\partial y_1^{k_1}} \frac{\partial y_1^{k_1}}{\partial \theta^l} \frac{\partial y_2^{k_2}}{\partial \theta^l}^T \frac{\partial f_2}{\partial y_2^{k_2}}^T =: \overbrace{\underbrace{\frac{\partial f_1}{\partial y_1}}_{\mathbf{O}\times\mathbf{Y}} \underbrace{\frac{\partial y_1}{\partial \theta}}_{\mathbf{Y}\times\mathbf{P}} \underbrace{\frac{\partial y_2}{\partial \theta}^T}_{\mathbf{P}\times\mathbf{Y}} \underbrace{\frac{\partial f_2}{\partial y_2}^T}_{\mathbf{Y}\times\mathbf{O}}}^{\mathbf{O}\times\mathbf{O}} \tag{10}$$

Structure of $\partial y/\partial\theta \downarrow$	**Outside-in**	**Left-to-right**	**Inside-out**
None w/ VJPs & JVPs:	$\mathbf{NO}\,[\mathbf{FP}] + \mathbf{N}^2\mathbf{O}^2\mathbf{P}$	$\mathbf{N}^2\mathbf{O}\,[\mathbf{FP}]$	Not possible
None w/ explicit matrices	$\mathbf{NOYP} + \mathbf{N}^2\mathbf{O}^2\mathbf{P}$	$\mathbf{N}^2\mathbf{OYP} + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}$	$\mathbf{N}^2\mathbf{Y}^2\mathbf{P} + \mathbf{N}^2\mathbf{OY}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}$
Block-diagonal	$\mathbf{NOYP}/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{P}$	$\mathbf{N}^2\mathbf{OYP}/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}$	$\mathbf{N}^2\mathbf{Y}^2\mathbf{P}/\mathbf{C}^2 + \mathbf{N}^2\mathbf{OY}^2/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}$
Constant block-diagonal	$\mathbf{NOYP}/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{P}$	$\mathbf{N}^2\mathbf{OYP}/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}$	$\mathbf{N}^2\mathbf{Y}^2\mathbf{P}/\mathbf{C}^3 + \mathbf{N}^2\mathbf{OY}^2/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}$
Input block-tiled	$\mathbf{NOYP}/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{P}$	$\mathbf{N}^2\mathbf{OYP}/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}$	$\mathbf{N}^2\mathbf{Y}^2\mathbf{P}/\mathbf{C} + \mathbf{N}^2\mathbf{OY}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}$
Output block-tiled	$\mathbf{NOYP}/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{P} + \mathbf{NOY}$	$\mathbf{N}^2\mathbf{OYP}/\mathbf{C} + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}/\mathbf{C} + \mathbf{NOY}$	$\mathbf{N}^2\mathbf{Y}^2\mathbf{P}/\mathbf{C}^2 + \mathbf{N}^2\mathbf{OY}^2/\mathbf{C}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}/\mathbf{C} + \mathbf{NOY}$
Block-tiled	$\mathbf{NOYP}/\mathbf{C}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{P}/\mathbf{C} + \mathbf{NOY}$	$\mathbf{N}^2\mathbf{OYP}/\mathbf{C}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}/\mathbf{C}^2 + \mathbf{NOY}$	$\mathbf{N}^2\mathbf{Y}^2\mathbf{P}/\mathbf{C}^3 + \mathbf{N}^2\mathbf{OY}^2/\mathbf{C}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{Y}/\mathbf{C} + \mathbf{NOY}$

Table 4: **Asymptotic time complexities of computing the contractions for NTK summands** $\Theta\left(f_1^{n_1}, f_2^{n_2}\right)(\theta^0, \ldots, \theta^{\mathbf{L}})_l^{k_1,k_2} \in \mathbb{R}^{\mathbf{O}\times\mathbf{O}}$ in Eq. (10), for all n_1 and n_2 from 1 to $\mathbf{N}$ (resulting in a $\mathbf{NO} \times \mathbf{NO}$ NTK matrix). Time complexity of Structured derivatives is the minimum (due to using `np.einsum` with optimal contraction order) of the row corresponding to the structure present in in a pair of primitives $y_1^{k_1}$ and $y_2^{k_2}$. How it compares to Jacobian contraction and NTK-vector products (top row) depends on many variables, including the cost of evaluating the primitive **FP**. See Table 2 and Table 3 for exact comparison in the case of convolution and matrix multiplication. See §B for legend.

We have dropped indices l, k_1 and k_2 on the right-hand side of Eq. (10) to avoid clutter, and consider $\theta := \theta^l, y_1 := y_1^{k_1}, y_2 := y_2^{k_2}$ until the end of this section. There are 3 ways of contracting Eq. (10) that cost

(a) **Outside-in:** $\mathbf{OYP} + \mathbf{O}^2\mathbf{P}$

(b) **Left-to-right and right-to-left:** $\mathbf{OYP} + \mathbf{O}^2\mathbf{Y}$.

(c) **Inside-out-left and inside-out-right:** $\mathbf{Y}^2\mathbf{P} + \mathbf{OY}^2 + \mathbf{O}^2\mathbf{Y}$.

In the next sections, we look at how these costs are reduced given certain structure in $\partial y/\partial\theta$.

I.2. Block diagonal

Assume $\partial y/\partial\theta = \oplus_{c=1}^{\mathbf{C}} \partial y^c/\partial\theta_c$, where $\oplus$ stands for direct sum of matrices, i.e. $\partial y/\partial\theta$ is a block diagonal matrix made of blocks $\{\partial y^c/\partial\theta_c\}_{c=1}^{\mathbf{C}}$, where $\partial y^c/\partial\theta_c$ have shapes $(\mathbf{Y}/\mathbf{C}) \times (\mathbf{P}/\mathbf{C})$. Here $\{y^c\}_{c=1}^{\mathbf{C}}$ and $\{\theta_c\}_{c=1}^{\mathbf{C}}$ are partitions of y and θ respectively. In NNs this structure is present in binary bilinear operations (on θ and another argument) such as multiplication, division, batched matrix multiplication, or depthwise convolution. Then Eq. (10) can be re-written as

$$\Theta_\theta^{l,k_1,k_2}(f_1, f_2) = \frac{\partial f_1}{\partial y_1}\frac{\partial y_1}{\partial\theta}\frac{\partial y_2}{\partial\theta}^T\frac{\partial f_2}{\partial y_2}^T \tag{11}$$

$$= \frac{\partial f_1}{\partial y_1}\left(\oplus_{c=1}^{\mathbf{C}}\frac{\partial y_1^c}{\partial\theta_c}\right)\left(\oplus_{c=1}^{\mathbf{C}}\frac{\partial y_2^c}{\partial\theta_c}\right)^T\frac{\partial f_2}{\partial y_2}^T \tag{12}$$

$$= \frac{\partial f_1}{\partial y_1}\left(\oplus_{c=1}^{\mathbf{C}}\left[\frac{\partial y_1^c}{\partial\theta_c}\frac{\partial y_2^c}{\partial\theta_c}^T\right]\right)\frac{\partial f_2}{\partial y_2}^T \tag{13}$$

$$= \sum_{c=1}^{\mathbf{C}}\frac{\partial f_1}{\partial y_1^c}\left[\frac{\partial y_1^c}{\partial\theta_c}\frac{\partial y_2^c}{\partial\theta_c}^T\right]\frac{\partial f_2}{\partial y_2^c}^T, \tag{14}$$

where we have applied the block matrix identity

$$[A^1, \dots, A^{\mathbf{C}}]^T \left(\oplus_{c=1}^{\mathbf{C}} B^c\right) [D^1, \dots, D^{\mathbf{C}}] = \sum_{c=1}^{\mathbf{C}} A^c B^c D^c.$$

We now perform a complexity analysis similar to Eq. (10):

$$\Theta_\theta^{l,k_1,k_2}(f_1, f_2) = \sum_{c=1}^{\mathbf{C}} \overbrace{\underbrace{\frac{\partial f_1}{\partial y_1^c}}_{\mathbf{O}\times(\mathbf{Y}/\mathbf{C})} \underbrace{\frac{\partial y_1^c}{\partial \theta_c}}_{(\mathbf{Y}/\mathbf{C})\times(\mathbf{P}/\mathbf{C})} \underbrace{\frac{\partial y_2^c}{\partial \theta_c}^T}_{(\mathbf{P}/\mathbf{C})\times(\mathbf{Y}/\mathbf{C})} \underbrace{\frac{\partial f_2}{\partial y_2^c}^T}_{(\mathbf{Y}/\mathbf{C})\times\mathbf{O}}}^{\mathbf{O}\times\mathbf{O}}$$

In this case complexities of the three methods become

1. **Outside-in:** $\mathbf{OYP}/\mathbf{C} + \mathbf{O}^2\mathbf{P}$.
2. **Left-to-right and right-to-left:** $\mathbf{OYP}/\mathbf{C} + \mathbf{O}^2\mathbf{Y}$.
3. **Inside-out-left and inside-out-right:** $\mathbf{Y}^2\mathbf{P}/\mathbf{C}^2 + \mathbf{OY}^2/\mathbf{C} + \mathbf{O}^2\mathbf{Y}$.

I.3. Constant-block diagonal

Assume $\frac{\partial y}{\partial \theta} = I_{\mathbf{C}} \otimes \frac{\partial y^1}{\partial \theta_1}$, and $\frac{\partial y^1}{\partial \theta_1}$ has shape $(\mathbf{Y}/\mathbf{C}) \times (\mathbf{P}/\mathbf{C})$. In NNs, this is present in fully-connected, convolutional, locally-connected, attention, and many other layers that contain a matrix multiplication along some axis. This is also present in all unary elementwise linear operations on θ like transposition, negation, reshaping and many others. This is a special case of §I.2 with $\frac{\partial y^c}{\partial \theta_c} = \frac{\partial y^1}{\partial \theta_1}$ for any c. Here a similar analysis applies, yielding

$$\Theta_\theta^{l,k_1,k_2}(f_1, f_2) = \sum_{c=1}^{\mathbf{C}} \overbrace{\underbrace{\frac{\partial f_1}{\partial y_1^c}}_{\mathbf{O}\times(\mathbf{Y}/\mathbf{C})} \underbrace{\frac{\partial y_1^1}{\partial \theta_1}}_{(\mathbf{Y}/\mathbf{C})\times(\mathbf{P}/\mathbf{C})} \underbrace{\frac{\partial y_2^1}{\partial \theta_1}^T}_{(\mathbf{P}/\mathbf{C})\times(\mathbf{Y}/\mathbf{C})} \underbrace{\frac{\partial f_2}{\partial y_2^c}^T}_{(\mathbf{Y}/\mathbf{C})\times\mathbf{O}}}^{\mathbf{O}\times\mathbf{O}}$$

and the same contraction complexities as in §I.2, except for the **Inside-out** order, where the inner contraction term costs only $\mathbf{Y}^2\mathbf{P}/\mathbf{C}^3$, since it is only contracted once instead of $\mathbf{C}$ times as in Block-diagonal.

I.4. Input block-tiled

Assume $\frac{\partial y}{\partial \theta} = \mathbb{1}_{(1,\mathbf{C})} \otimes \frac{\partial y}{\partial \theta_1}$, where $\mathbb{1}_{(1,\mathbf{C})}$ is an all-ones matrix of shape $1 \times \mathbf{C}$, and $\frac{\partial y}{\partial \theta_1}$ has shape $\mathbf{Y} \times (\mathbf{P}/\mathbf{C})$. In this case

$$\Theta_\theta^{l,k_1,k_2}(f_1,f_2) = \frac{\partial f_1}{\partial y_1}\frac{\partial y_1}{\partial \theta}\frac{\partial y_2}{\partial \theta}^T\frac{\partial f_2}{\partial y_2}^T \tag{15}$$

$$= \frac{\partial f_1}{\partial y_1}\left(\mathbb{1}_{(1,\mathbf{C})}\otimes\frac{\partial y_1}{\partial \theta_1}\right)\left(\mathbb{1}_{(1,\mathbf{C})}\otimes\frac{\partial y_2}{\partial \theta_1}\right)^T\frac{\partial f_2}{\partial y_2}^T \tag{16}$$

$$= \frac{\partial f_1}{\partial y_1}\left(\mathbf{C}\mathbb{1}_{(1,1)}\otimes\left[\frac{\partial y_1}{\partial \theta_1}\frac{\partial y_2}{\partial \theta_1}^T\right]\right)\frac{\partial f_2}{\partial y_2}^T \tag{17}$$

$$= \mathbf{C}\frac{\partial f_1}{\partial y_1}\left[\frac{\partial y_1}{\partial \theta_1}\frac{\partial y_2}{\partial \theta_1}^T\right]\frac{\partial f_2}{\partial y_2}^T. \tag{18}$$

The matrix shapes are

$$\Theta_\theta^{l,k_1,k_2}(f_1,f_2) = \mathbf{C}\ \overbrace{\underbrace{\frac{\partial f_1}{\partial y_1}}_{\mathbf{O}\times\mathbf{Y}}\ \underbrace{\frac{\partial y_1}{\partial \theta_1}}_{\mathbf{Y}\times(\mathbf{P}/\mathbf{C})}\ \underbrace{\frac{\partial y_2}{\partial \theta_1}^T}_{(\mathbf{P}/\mathbf{C})\times\mathbf{Y}}\ \underbrace{\frac{\partial f_2}{\partial y_2}^T}_{\mathbf{Y}\times\mathbf{O}}}^{\mathbf{O}\times\mathbf{O}}$$

Which leads to the following resulting complexities:

1. **Outside-in:** $\mathbf{OYP}/\mathbf{C} + \mathbf{O}^2\mathbf{P}$.
2. **Left-to-right and right-to-left:** $\mathbf{OYP}/\mathbf{C} + \mathbf{O}^2\mathbf{Y}$.
3. **Inside-out and inside-out-right:** $\mathbf{Y}^2\mathbf{P}/\mathbf{C} + \mathbf{OY}^2 + \mathbf{O}^2\mathbf{Y}$.

I.5. Output block-tiled

Assume $\frac{\partial y}{\partial \theta} = \mathbb{1}_{(\mathbf{C},1)}\otimes\frac{\partial y^1}{\partial \theta}$, where $\frac{\partial y^1}{\partial \theta}$ has shape $(\mathbf{Y}/\mathbf{C})\times\mathbf{P}$. This occurs during broadcasting or broadcasted arithmetic operations. In this case

$$\Theta_\theta^{l,k_1,k_2}(f_1,f_2) = \frac{\partial f_1}{\partial y_1}\frac{\partial y_1}{\partial \theta}\frac{\partial y_2}{\partial \theta}^T\frac{\partial f_2}{\partial y_2}^T \tag{19}$$

$$= \frac{\partial f_1}{\partial y_1}\left(\mathbb{1}_{(\mathbf{C},1)}\otimes\frac{\partial y_1^1}{\partial \theta}\right)\left(\mathbb{1}_{(\mathbf{C},1)}\otimes\frac{\partial y_2^1}{\partial \theta}\right)^T\frac{\partial f_2}{\partial y_2}^T \tag{20}$$

$$= \frac{\partial f_1}{\partial y_1}\left(\mathbb{1}_{(\mathbf{C},\mathbf{C})}\otimes\left[\frac{\partial y_1^1}{\partial \theta}\frac{\partial y_2^1}{\partial \theta}^T\right]\right)\frac{\partial f_2}{\partial y_2}^T \tag{21}$$

$$= \left(\sum_{c=1}^{\mathbf{C}}\frac{\partial f_1}{\partial y_1^c}\right)\left[\frac{\partial y_1^1}{\partial \theta_1}\frac{\partial y_2^1}{\partial \theta_1}^T\right]\left(\sum_{c=1}^{\mathbf{C}}\frac{\partial f_2}{\partial y_2^c}^T\right), \tag{22}$$

where we have used a block matrix identity

$$[A^1,\ldots,A^{\mathbf{C}}]^T\left(\mathbb{1}_{(\mathbf{C},\mathbf{C})}\otimes B\right)[D^1,\ldots,D^{\mathbf{C}}] = \left(\sum_{c=1}^{\mathbf{C}}A^c\right)B\left(\sum_{c=1}^{\mathbf{C}}D^c\right).$$

Finally, denoting the shapes,

$$\Theta_\theta^{l,k_1,k_2}(f_1,f_2) = \overbrace{\underbrace{\left(\sum_{c=1}^{\mathbf{C}} \frac{\partial f_1}{\partial y_1^c}\right)}_{\mathbf{O}\times(\mathbf{Y}/\mathbf{C})} \underbrace{\frac{\partial y_1^1}{\partial \theta}}_{(\mathbf{Y}/\mathbf{C})\times\mathbf{P}} \underbrace{\frac{\partial y_2^1}{\partial \theta}^T}_{\mathbf{P}\times(\mathbf{Y}/\mathbf{C})} \underbrace{\left(\sum_{c=1}^{\mathbf{C}} \frac{\partial f_2}{\partial y_2^c}^T\right)}_{(\mathbf{Y}/\mathbf{C})\times\mathbf{O}}}^{\mathbf{O}\times\mathbf{O}},$$

complexities of the three methods become (notice we add an $\mathbf{OY}$ term to perform the sums)

1. **Outside-in:** $\mathbf{OYP}/\mathbf{C} + \mathbf{O}^2\mathbf{P} + \mathbf{OY}$.
2. **Left-to-right:** $\mathbf{OYP}/\mathbf{C} + \mathbf{O}^2\mathbf{Y}/\mathbf{C} + \mathbf{OY}$.
3. **Inside-out:** $\mathbf{Y}^2\mathbf{P}/\mathbf{C}^2 + \mathbf{OY}^2/\mathbf{C}^2 + \mathbf{O}^2\mathbf{Y}/\mathbf{C} + \mathbf{OY}$.

I.6. Block-tiled

Assume $\frac{\partial y}{\partial \theta} = \mathbb{1}_{(\mathbf{C},\mathbf{C})} \otimes \frac{\partial y^1}{\partial \theta_1}$, where $\frac{\partial y^1}{\partial \theta_1}$ has shape $(\mathbf{Y}/\mathbf{C}) \times (\mathbf{P}/\mathbf{C})$. This occures for instance when y is a constant. In this case

$$\Theta_\theta^{l,k_1,k_2}(f_1,f_2) = \frac{\partial f_1}{\partial y_1}\frac{\partial y_1}{\partial \theta}\frac{\partial y_2}{\partial \theta}^T\frac{\partial f_2}{\partial y_2}^T \tag{23}$$

$$= \frac{\partial f_1}{\partial y_1}\left(\mathbb{1}_{(\mathbf{C},\mathbf{C})} \otimes \frac{\partial y_1^1}{\partial \theta_1}\right)\left(\mathbb{1}_{(\mathbf{C},\mathbf{C})} \otimes \frac{\partial y_2^1}{\partial \theta_1}\right)^T\frac{\partial f_2}{\partial y_2}^T \tag{24}$$

$$= \frac{\partial f_1}{\partial y_1}\left(\mathbf{C}\mathbb{1}_{(\mathbf{C},\mathbf{C})} \otimes \left[\frac{\partial y_1^1}{\partial \theta_1}\frac{\partial y_2^1}{\partial \theta_1}^T\right]\right)\frac{\partial f_2}{\partial y_2}^T \tag{25}$$

$$= \mathbf{C}\left(\sum_{c=1}^{\mathbf{C}} \frac{\partial f_1}{\partial y_1^c}\right)\left[\frac{\partial y_1^1}{\partial \theta_1}\frac{\partial y_2^1}{\partial \theta_1}^T\right]\left(\sum_{c=1}^{\mathbf{C}} \frac{\partial f_2}{\partial y_2^c}^T\right), \tag{26}$$

This results in the following contraction:

$$\Theta_\theta^{l,k_1,k_2}(f_1,f_2) = \mathbf{C}\ \overbrace{\underbrace{\left(\sum_{c=1}^{\mathbf{C}} \frac{\partial f_1}{\partial y_1^c}\right)}_{\mathbf{O}\times(\mathbf{Y}/\mathbf{C})} \underbrace{\frac{\partial y_1^1}{\partial \theta_1}}_{(\mathbf{Y}/\mathbf{C})\times(\mathbf{P}/\mathbf{C})} \underbrace{\frac{\partial y_2^1}{\partial \theta_1}^T}_{(\mathbf{P}/\mathbf{C})\times(\mathbf{Y}/\mathbf{C})} \underbrace{\left(\sum_{c=1}^{\mathbf{C}} \frac{\partial f_2}{\partial y_2^c}^T\right)}_{(\mathbf{Y}/\mathbf{C})\times\mathbf{O}}}^{\mathbf{O}\times\mathbf{O}},$$

with final complexities of

1. **Outside-in:** $\mathbf{OYP}/\mathbf{C}^2 + \mathbf{O}^2\mathbf{P} + \mathbf{OY}$.
2. **Left-to-right:** $\mathbf{OYP}/\mathbf{C}^2 + \mathbf{O}^2\mathbf{Y}/\mathbf{C}^2 + \mathbf{OY}$.
3. **Inside-out:** $\mathbf{Y}^2\mathbf{P}/\mathbf{C}^3 + \mathbf{OY}^2/\mathbf{C}^2 + \mathbf{O}^2\mathbf{Y}/\mathbf{C} + \mathbf{OY}$.

I.7. Batched NTK cost analysis

For simplicity, we have considered evaluating the NTK $\Theta(f_1, f_2)$ on a single pair of functions f_1 and f_2. In practice one is almost always interested in computing the NTK for all pairs of functions $f_1^{n_1}$ and $f_2^{n_2}$ from two batches $\{f_1^{n_1}\}_{n_1=1}^{\mathbf{N}_1}$ and $\{f_2^{n_2}\}_{n_2=1}^{\mathbf{N}_2}$, resulting in a $\mathbf{N}_1\mathbf{O}_1 \times \mathbf{N}_2\mathbf{O}_2$ NTK matrix. In common NNs, this corresponds to having batches of $\mathbf{N}_1$ and $\mathbf{N}_2$ inputs x_1 and x_2 respectively, and having $f_1^{n_i}\left(\theta^0, \ldots, \theta^{\mathbf{L}}\right) := f\left(\theta^0, \ldots, \theta^{\mathbf{L}}, x_i^{n_i}\right)$. In this case the same argument as in previous section follows (given identical assumptions for all n_1 and n_2), but the cost of contractions involving terms from different batches grow by a multiplicative factor of $\mathbf{N}_1\mathbf{N}_2$, while all other costs grow by a factor of $\mathbf{N}_1$ or $\mathbf{N}_2$. To declutter notation we consider $\mathbf{N}_1 = \mathbf{N}_2 = \mathbf{N}$, and summarize resulting costs in Table 4.

I.8. Complex structure cost analysis

In previous sections we have considered $\partial y_1/\partial\theta$ and $\partial y_2/\partial\theta$ admitting the same, and at most one kind of structure. While this is a common case, in general these derivatives may admit multiple types of structures along multiple axes (for instance, addition is Constant block-diagonal along non-broadcasted axes, and Output block-tiled along the broadcasted axes), and $\partial y_1/\partial\theta$ and $\partial y_2/\partial\theta$ may have different types of structures and respective axes, if the same weight θ is used in multiple different subexpressions of different kind. In such cases, equivalent optimizations are possible (and are implemented in the code) along the largest common subsets of axes for each type of structure that $\partial y_1/\partial\theta$ and $\partial y_2/\partial\theta$ have.

For example, let θ be a matrix in $\mathbb{R}^{\mathbf{W}\times\mathbf{W}}$, y_1 be multiplication by a scalar $y_1(\theta) = 2\theta$, and y_2 be matrix-vector multiplication $y_2(\theta) = \theta x$, $x \in \mathbb{R}^{\mathbf{W}}$. In this case $\partial y_1/\partial\theta = 2I_{\mathbf{W}} \otimes I_{\mathbf{W}}$, i.e. it is Constant block-diagonal along axes both 1 and 2. $\partial y_2/\partial\theta = I_{\mathbf{W}} \otimes x^T$, i.e. it is also Constant block-diagonal, but only along axis 1. Hence, the NTK term containing $\partial y_1/\partial\theta$ and $\partial y_2/\partial\theta$ will be computed with Constant block-diagonal simplification along axis 1. There are probably more computationally optimal ways of processing different structure combinations, as well as more types of structures to be leveraged for NTK computation, and we intend to investigate it in future work.

I.9. Example

In §2.4 and previous sections we have demonstrated how structure in primitive derivatives $\partial y/\partial\theta$ can be leveraged to reduce the cost of computing NTK. In this section we will consider a simple example of applying the framework of structured derivatives to FCNs to reproduce Table 3. See §N for equivalent application for CNNs.

As in §L.4, we consider a deep FCN with width $\mathbf{W}$ and $\mathbf{O}$ outputs. We assume the network is deep and/or wide enough to ignore the size of inputs x, and we ignore biases. In this case the number of parameters is quadratic in width $\mathbf{P} \sim \mathbf{W}^2$, and intermediate primitive outputs have the same size as the width, $\mathbf{Y} = \mathbf{W}$. We recognize that individual primitives $y^{k,n}(\theta_k) = \theta_l x^{k,n}$, as matrix multiplications $\left(\theta_k \in \mathbb{R}^{\mathbf{W}\times\mathbf{W}}, x^{k,n} \in \mathbb{R}^{\mathbf{W}}\right)$ admit the Constant block-diagonal structure $\left(\partial y^{k,n}/\partial\theta_k = I_{\mathbf{W}} \otimes {x^{k,n}}^T\right)$ with $\mathbf{C} = \mathbf{Y} = \mathbf{W} = \mathbf{J}$. Finally, **FP** costs $\mathbf{W}^2$. Substituting all these equalities into Table 4 we get a simplified Table 5, that confirms the benefits of NTK-vector products and Structured derivatives for FCNs.

Structure of $\partial y/\partial\theta \downarrow$	**Outside-in**	**Left-to-right**	**Inside-out**
None w/ JVPs and VJPs	$\mathbf{N}^2\mathbf{O}^2\mathbf{W}^2$	$\mathbf{N}^2\mathbf{O}\mathbf{W}^2$	Not possible
None	$\mathbf{N}\mathbf{W}^3\mathbf{O} + \mathbf{N}^2\mathbf{O}^2\mathbf{W}^2$	$\mathbf{N}^2\mathbf{O}\mathbf{W}^3 + \mathbf{N}^2\mathbf{O}^2\mathbf{W}$	$\mathbf{N}^2\mathbf{W}^4 + \mathbf{N}\mathbf{O}\mathbf{W}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{W}$
Constant block-diagonal	$\mathbf{N}^2\mathbf{O}^2\mathbf{W}^2$	$\mathbf{N}^2\mathbf{O}\mathbf{W}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{W}$	$\mathbf{N}^2\mathbf{O}^2\mathbf{W}$

Table 5: **Asymptotic time complexities of computing a single fully-connected layer NTK contribution**. See §I.9 for discussion, Table 4 for a more general setting, Table 3 for the case of deep networks, and §B for detailed legend.

J. Jacobian rules for structured derivatives

Here we discuss computing primitive $\partial y/\partial\theta$ Jacobians as part of our implementation in §C. We provide 4 options to compute them through arguments `j_rules` and `fwd`:

1. **Forward mode**, `fwd = True`, is equivalent to `jax.jacfwd`, forward mode Jacobian computation, performed by applying the JVP to $\mathbf{P}$ columns of the $I_{\mathbf{P}}$ identity matrix. Best for $\mathbf{P} < \mathbf{Y}$.

2. **Reverse mode**, `fwd = False`, is equivalent to `jax.jacrev`, reverse mode Jacobian computation, performed by applying the VJP to $\mathbf{Y}$ columns of the $I_{\mathbf{Y}}$ identity matrix. Best for $\mathbf{P} > \mathbf{Y}$.

3. **Automatic mode**, `fwd = None`, selects forward or reverse mode for each primitive based on parameters and output shapes.

4. **Rule mode**, `j_rules = True`, queries a dictionary of Jacobian rules (similar to the dictionary of structure rules) with our custom implementations of primitive Jacobians, instead of computing them through VJPs or JVPs. The reason for introducing custom rules follows our discussion in §2.4: while JAX has computationally optimal VJP and JVP rules, the respective Jacobian computations are not guaranteed to be most efficient. In practice, we find our rules to be most often faster, however this effect is not perfectly consistent (can occasionally be slower) and often negligible, requiring further investigation.

The default setting is `j_rules = True`, `fwd = None`, i.e. a custom Jacobian implementation is preferred, and, if absent, Jacobian is computed in forward or reverse mode based on parameters and output sizes. Note that in all settings, structure of $\partial y/\partial\theta$ is used to compute only the smallest Jacobian subarray necessary, and therefore most often inputs to VJP/JVP will be smaller identity matrices $I_{\mathbf{P}/\mathbf{C}}$ or $I_{\mathbf{Y}/\mathbf{C}}$ respectively, and all methods will return a smaller Jacobian matrix of size $(\mathbf{Y}/\mathbf{C}) \times (\mathbf{P}/\mathbf{C})$. If for any reason (for example debugging) you want the whole $\partial y/\partial\theta$ Jacobians computed, you can set the `a_rules=False`, i.e. disable structure rules.

K. Known issues

We will continue improving our function transformations in various ways after release, and welcome bug reports and feature requests. Below are the missing features / issues at the time of submission:

Transposable primitive in `jax.ad.primitive_transposes`	Constant block-diagonal	Block-diagonal	Output block-tiled
add	✓		✓
add_any	✓		✓
all_gather			
all_to_all			
broadcast_in_dim	✓		✓
call			
complex	✓		
concatenate			
conj	✓		
conv_general_dilated			
convert_element_type	✓		
cumsum			
custom_lin			
custom_linear_solve			
device_put	✓		
div	✓	✓	
dot_general	✓	✓	
dynamic_slice			
dynamic_update_slice			
fft			
gather			
imag	✓		
linear_call			
mul	✓	✓	
named_call			
neg	✓		
pad	✓		
pdot			
ppermute			
psum			
real	✓		
reduce_sum	✓		
reduce_window_sum	✓		
remat_call			
reshape	✓		
rev	✓		
scatter			
scatter-add			
scatter-mul			
select			
select_and_gather_add			
select_and_scatter_add			
sharding_constraint			
sharding_constraint			
slice			
squeeze	✓		
sub	✓		✓
transpose	✓		
triangular_solve			
while			
xla_call			
xla_pmap			
xmap			
zeros_like	✓		

Table 6: **List of all linear primitives and currently implemented Structured derivatives rules.** In the future, more primitives and more rules can be supported, yet at the time of writing even the small set currently covered enables dramatic speed-up and memory savings in contemporary ImageNet models as in Fig. 2 and Fig. 4.

1. No support for complex differentiation.
2. Not tested on functions with advanced JAX primitives like parallel collectives (`psum`, `pmean`, etc.), gradient checkpointing (`remat`), compiled loops (`scan`; Python loops are supported).
3. Our current implementation of NTK-vector products relies on XLA's common subexpression elimination (CSE) in order to reuse computation across different pairs of inputs x_1 and x_2, and, as shown in Fig. 1 and Fig. 3, can have somewhat unpredictable wall-clock time performance and memory requirements. We believe this could correspond to CSE not always working perfectly, and are looking into a more explicitly efficient implementation.

L. Complexity analysis for fully-connected networks

This section presents our contributions in a simplified setting of fully-connected (FCN) networks. It can be read independently from §2, where a more general discussion is presented.

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 θ^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 Fig. 5 for a visual schematic of these quantities. For simplicity, we assume that inputs x also have width $\mathbf{W}$, and $\mathbf{O} = \mathcal{O}(\mathbf{L}\mathbf{W})$, 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

$$\Theta_\theta := \frac{\partial f(\theta, x_1)}{\partial \theta} \frac{\partial f(\theta, x_2)}{\partial \theta}^T = \sum_{l=0}^{\mathbf{L}} \frac{\partial f(\theta, x_1)}{\partial \theta^l} \frac{\partial f(\theta, x_2)}{\partial \theta^l}^T =: \sum_{l=0}^{\mathbf{L}} \Theta_\theta^l \in \mathbb{R}^{\mathbf{O} \times \mathbf{O}}, \tag{27}$$

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

In §L.1 and §L.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 §L.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 §L.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 §L.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.

L.1. Jacobian-vector products and vector-Jacobian products

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

$$\mathrm{JVP}_{(f,\theta,x)} : \theta_t \in \mathbb{R}^{\mathbf{LW}^2+\mathbf{OW}} \mapsto \frac{\partial f(\theta, x)}{\partial \theta}\theta_t \in \mathbb{R}^{\mathbf{O}}, \tag{28}$$

$$\mathrm{VJP}_{(f,\theta,x)} : f_c \in \mathbb{R}^{\mathbf{O}} \mapsto \frac{\partial f(\theta, x)}{\partial \theta}^T f_c \in \mathbb{R}^{\mathbf{LW}^2+\mathbf{OW}}. \tag{29}$$

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[5] of both operations is comparable to the forward pass (**FP**; see §H), i.e. $[\mathbf{FP}] = [\text{cost of all intermediate layers}] + [\text{cost of the top layer}] = \left[\mathbf{LW}^2\right] + [\mathbf{OW}] \sim \mathbf{LW}^2$.

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

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

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,

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

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.

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

L.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.

$$\left[\partial f\left(\theta, x\right)/\partial\theta\right]^T = \left[\partial f\left(\theta, x\right)/\partial\theta\right]^T I_{\mathbf{O}} \in \mathbb{R}^{(\mathbf{LW}^2+\mathbf{OW})\times\mathbf{O}}. \tag{30}$$

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

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

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.

L.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 Eq. (2):

$$\underbrace{\Theta_\theta^l}_{\mathbf{O}\times\mathbf{O}} = \underbrace{\frac{\partial f\left(\theta, x_1\right)}{\partial\theta^l}}_{\mathbf{O}\times(\mathbf{W}\times\mathbf{W})} \underbrace{\frac{\partial f\left(\theta, x_2\right)}{\partial\theta^l}^T}_{(\mathbf{W}\times\mathbf{W})\times\mathbf{O}}. \tag{31}$$

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 θ^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 [NTK matrix size] + [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 §L.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.

L.4. Leveraging structured derivatives for computing the NTK

We can rewrite ${\Theta^l}_\theta$ in Eq. (31) using the chain rule and our pre- and post-activation notation as:

$$\Theta^l_\theta = \left[\frac{\partial f(\theta, x_1)}{\partial y^l_{x_1}} \frac{\partial y^l_{x_1}}{\partial \theta^l}\right] \left[\frac{\partial f(\theta, x_2)}{\partial y^l_{x_2}} \frac{\partial y^l_{x_2}}{\partial \theta^l}\right]^T = \underbrace{\frac{\partial f(\theta, x_1)}{\partial y^l_{x_1}}}_{\mathbf{O}\times\mathbf{W}} \underbrace{\frac{\partial y^l_{x_1}}{\partial \theta^l}}_{\mathbf{W}\times(\mathbf{W}\times\mathbf{W})} \underbrace{\frac{\partial y^l_{x_2}}{\partial \theta^l}^T}_{(\mathbf{W}\times\mathbf{W})\times\mathbf{W}} \underbrace{\frac{\partial f(\theta, x_2)}{\partial y^l_{x_2}}^T}_{\mathbf{W}\times\mathbf{O}}. \tag{32}$$

At face value, rewriting Eq. (31) 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

$$\frac{\partial y^l_{x_1}}{\partial \theta^l} = I_{\mathbf{W}} \otimes {x^l_1}^T, \quad \frac{\partial y^l_{x_2}}{\partial \theta^l} = I_{\mathbf{W}} \otimes {x^l_2}^T, \tag{33}$$

where $\otimes$ is the Kronecker product. Plugging Eq. (33) into Eq. (32) we get

$$\Theta^l_\theta(x_1, x_2) = \frac{\partial f(\theta, x_1)}{\partial y^l_{x_1}} \left(I_{\mathbf{W}} \otimes {x^l_1}^T\right) \left(I_{\mathbf{W}} \otimes {x^l_2}^T\right)^T \frac{\partial f(\theta, x_2)}{\partial y^l_{x_2}}^T = \tag{34}$$

$$= \frac{\partial f(\theta, x_1)}{\partial y^l_{x_1}} \left(I_{\mathbf{W}} \otimes \left[{x^l_1}^T x^l_2\right]\right) \frac{\partial f(\theta, x_2)}{\partial y^l_{x_2}}^T = \left({x^l_1}^T x^l_2\right) \left[\frac{\partial f(\theta, x_1)}{\partial y^l_{x_1}} \frac{\partial f(\theta, x_2)}{\partial y^l_{x_2}}^T\right], \tag{35}$$

where we were able to pull out $\left({x^l_1}^T x^l_2\right)$ since it is a scalar. Therefore we obtain

$$\Theta^l_\theta = \begin{pmatrix} \underbrace{{x^l_1}^T}_{1\times\mathbf{W}} & \underbrace{x^l_2}_{\mathbf{W}\times 1} \end{pmatrix} \left[\underbrace{\frac{\partial f(\theta, x_1)}{\partial y^l_{x_1}}}_{\mathbf{O}\times\mathbf{W}} \quad \underbrace{\frac{\partial f(\theta, x_2)}{\partial y^l_{x_2}}^T}_{\mathbf{W}\times\mathbf{O}}\right], \tag{36}$$

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, $\Theta^l_\theta(x_1, x_2)$, and factors, $\partial f(\theta, x) / \partial y^l_x$, 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,

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

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 §2.4 and detail in the §I, 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 Eq. (33) 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 discuss how we impose this structure to compute the NTK for general neural networks in §I.

L.5. NTK via NTK-vector products

Computing the Jacobian contraction using Jacobian 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: $\Theta\mathrm{VP} : v \in \mathbb{R}^{\mathbf{O}} \mapsto \Theta_\theta v \in \mathbb{R}^{\mathbf{O}}$. 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 $\Theta\mathrm{VP}(v)$. We now expand $\Theta\mathrm{VP}(v) = \Theta_\theta v$ as

$$\frac{\partial f(\theta, x_1)}{\partial \theta} \frac{\partial f(\theta, x_2)}{\partial \theta}^T v = \frac{\partial f(\theta, x_1)}{\partial \theta} \mathrm{VJP}_{(f,\theta,x_2)}(v) = \mathrm{JVP}_{(f,\theta,x_1)}\left[\mathrm{VJP}_{(f,\theta,x_2)}(v)\right], \tag{37}$$

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 Eq. (37) 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.

M. Complexity analysis without the $\mathbf{O} = \mathcal{O}(\mathbf{LW})$ assumption

Here we repeat the same analysis as in §L without the assumption of $\mathbf{O} = \mathcal{O}(\mathbf{LW})$. This results in Table 7, where Jacobian contraction and Structured derivatives gain an extra $\mathbf{N}^2\mathbf{O}^3\mathbf{W}$ and $\mathbf{N}^2\mathbf{O}^3$ time terms respectively. This does not affect our main text conclusions.

M.1. JVP and VJP

As in §L.1, the time cost of both operations is comparable to the forward pass (**FP**), i.e. $[\mathbf{FP}] = [\text{cost of all intermediate layers}] + [\text{cost of the top layer}] = \mathbf{LW}^2 + \mathbf{OW}$.

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

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

As in §L.1, 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 $\mathbf{NLW}^2 + \mathbf{NOW}$. The memory cost is more nuanced. Since weights can be shared across inputs, the memory cost of the JVP and VJP are respectively,

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

Recall from §L.1 we only need to store one cotangent weight matrix, $\partial f / \partial \theta^l$, at a time. Therefore

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

M.2. Jacobian

The time and memory costs to compute the Jacobian are identical to §2.1.3,

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

Therefore, asymptotically, the costs are identical to §2.1.3:

> 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.

M.3. Jacobian contraction

The time cost of the contraction in Eq. (31) is $\mathbf{O}^2\mathbf{W}^2$ for $l < \mathbf{L}$, but is $\mathbf{O}^3\mathbf{W}$ for the top layer $l = \mathbf{L}$. The memory necessary to instantiate each factor and the result is $\mathbf{OW}^2 + \mathbf{O}^2$ for $l < \mathbf{L}$ and $\mathbf{O}^2\mathbf{W}$ for the top layer $l = \mathbf{L}$.

Accounting for all layers together, we arrive at $\mathbf{LO}^2\mathbf{W}^2 + \mathbf{O}^3\mathbf{W}$ time cost and $\mathbf{OW}^2 + \mathbf{O}^2\mathbf{W}$ 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 $\mathbf{N}^2\left(\mathbf{LO}^2\mathbf{W}^2 + \mathbf{O}^3\mathbf{W}\right)$, while memory grows to [NTK matrix size] + [factors size] $= \mathbf{N}^2\mathbf{O}^2 + \mathbf{N}\left(\mathbf{OW}^2 + \mathbf{O}^2\mathbf{W}\right)$.

Adding the cost of the Jacobian described in §M.2, we obtain

> Jacobian contraction costs $\mathbf{N}^2\mathbf{LO}^2\mathbf{W}^2 + \mathbf{N}^2\mathbf{O}^3\mathbf{W}$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOW}^2 + \mathbf{NO}^2\mathbf{W} + \mathbf{LW}^2 + \mathbf{NLW}$ memory.

M.4. Structured derivatives

The contraction in Eq. (36) takes $\mathbf{O}^2\mathbf{W}$ time and $\mathbf{OW} + \mathbf{O}^2$ memory for $l < \mathbf{L}$, and $\mathbf{O}^3 + \mathbf{W}$ time and $\mathbf{O}^2 + \mathbf{W}$ memory for $l = \mathbf{L}$.

Accounting for all layers, time cost becomes $\mathbf{LO}^2\mathbf{W} + \mathbf{O}^3$, and memory remains $\mathbf{OW} + \mathbf{O}^2$.

Batched inputs. In the batched setting, the time cost grows quadratically with the size of the NTK to $\mathbf{N}^2\mathbf{LO}^2\mathbf{W} + \mathbf{N}^2\mathbf{O}^3$, while the memory cost increases to $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOW}$.

Extra memory cost for computing the derivatives is

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

The extra time cost is asymptotically the cost of $\mathbf{O}$ forward passes, $\mathbf{NLOW}^2 + \mathbf{NO}^2\mathbf{W}$ which is the same as 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 $\mathbf{N}^2\mathbf{LO}^2\mathbf{W} + \mathbf{N}^2\mathbf{O}^3 + \mathbf{NLOW}^2$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOW} + \mathbf{LW}^2 + \mathbf{NLW}$ memory.

M.5. NTK-vector products

The cost analysis of NTK-vector products in §2.3 is not impacted by the $\mathbf{O} = \mathcal{O}(\mathbf{LW})$ assumption, hence it remains the same as in §2.3:

Method	Time	Memory	Use when
Jacobian contraction	$\mathbf{N}^2\mathbf{L}\mathbf{O}^2\mathbf{W}^2 + \mathbf{N}^2\mathbf{O}^3\mathbf{W}$	$\mathbf{N}\mathbf{O}\mathbf{W}^2 + \mathbf{N}^2\mathbf{O}^2 + \mathbf{N}\mathbf{L}\mathbf{W} + \mathbf{L}\mathbf{W}^2$	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	$\mathbf{N}^2\mathbf{L}\mathbf{O}^2\mathbf{W} + \mathbf{N}\,\mathbf{L}\mathbf{O}\mathbf{W}^2 + \mathbf{N}^2\mathbf{O}^3$	$\mathbf{N}\mathbf{O}\mathbf{W} + \mathbf{N}^2\mathbf{O}^2 + \mathbf{N}\mathbf{L}\mathbf{W} + \mathbf{L}\mathbf{W}^2$	$\mathbf{O} < \mathbf{W}$ or $\mathbf{L} = 1$

Table 7: **Asymptotic time and memory cost of computing the NTK for an FCN without assuming that $\mathbf{O} = \mathcal{O}(\mathbf{L}\mathbf{W})$.** 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 $\mathbf{N}\mathbf{O} \times \mathbf{N}\mathbf{O}$. NTK-vector products allow a reduction of the time complexity, while Structured derivatives reduce both time and memory complexity. **Note:** presented are asymptotic cost estimates; in practice, all methods incur large constant multipliers (e.g. at least 3x for time; see §L.1). However, this generally does not impact the relative performance of different methods. See Table 3 for a simplified cost summary under the assumption of $\mathbf{O} = \mathcal{O}(\mathbf{L}\mathbf{W})$ (differing only by lacking the $\mathbf{N}^2\mathbf{O}^3\mathbf{W}$ and $\mathbf{N}^2\mathbf{O}^3$ terms in Jacobian contraction and Structured derivatives time costs respectively), Table 2 for CNN, and Table 1 for more generic cost analysis.

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

N. Complexity analysis for convolutional networks

Here we go through the same analysis as in §L for the case of convolution, where before the top layer $\mathbf{L}$ global average pooling is applied. In this case the weights of the network θ are expanded by the total filter size $\mathbf{F}$, and inputs x, pre-activations y^l and post-activations x^l become matrices of shape $\mathbf{D} \times \mathbf{W}$, where $\mathbf{D}$ is the total number of pixels. See Fig. 5 for visual depiction. We will again assume that $\mathbf{O} = \mathcal{O}(\mathbf{L}\mathbf{W})$.

N.1. JVP and VJP

Forward pass, JVP, and VJP costs [cost of all intermediate layers]+[cost of the top layer] $= \left[\mathbf{L}\mathbf{D}\mathbf{F}\mathbf{W}^2\right] + [\mathbf{O}\mathbf{W}] \sim \mathbf{L}\mathbf{D}\mathbf{F}\mathbf{W}^2$ time. Forward pass and JVP require [all weights] + [activations at a single layer] $= \left[\mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{O}\mathbf{W}\right] + [\mathbf{D}\mathbf{W} + \mathbf{O}] \sim \mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{D}\mathbf{W}$ memory. VJP requires [all weights] + [activations in all layers] + [a single weight matrix] $= \left[\mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{O}\mathbf{W}\right] + [\mathbf{L}\mathbf{D}\mathbf{W} + \mathbf{O}] + \left[\mathbf{F}\mathbf{W}^2 + \mathbf{O}\mathbf{W}\right] \sim \mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{L}\mathbf{D}\mathbf{W}$ memory.

Batched inputs. Time cost of JVP and VJP increase linearly in $\mathbf{N}$ up to $\mathbf{N}\mathbf{L}\mathbf{D}\mathbf{F}\mathbf{W}^2$. JVP memory cost becomes [all weights] + $\mathbf{N}$ [activations at a single layer] $= \left[\mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{O}\mathbf{W}\right] + \mathbf{N}\,[\mathbf{D}\mathbf{W} + \mathbf{O}] \sim \mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{N}\mathbf{D}\mathbf{W} + \mathbf{N}\mathbf{O}$. VJP memory cost becomes [all weights] + $\mathbf{N}$ [activations in all layers] + $\mathbf{N}$ [a single weight matrix] $= \left[\mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{O}\mathbf{W}\right] + \mathbf{N}\,[\mathbf{L}\mathbf{D}\mathbf{W} + \mathbf{O}] + \mathbf{N}\left[\mathbf{F}\mathbf{W}^2 + \mathbf{O}\mathbf{W}\right] \sim \mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{N}\mathbf{L}\mathbf{D}\mathbf{W} + \mathbf{N}\mathbf{F}\mathbf{W}^2 + \mathbf{N}\mathbf{O}\mathbf{W}$.

> - JVP costs $\mathbf{N}\mathbf{L}\mathbf{D}\mathbf{F}\mathbf{W}^2$ time and $\mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{N}\mathbf{D}\mathbf{W} + \mathbf{N}\mathbf{O}$ memory.
> - VJP costs $\mathbf{N}\mathbf{L}\mathbf{D}\mathbf{F}\mathbf{W}^2$ time and $\mathbf{L}\mathbf{F}\mathbf{W}^2 + \mathbf{N}\mathbf{L}\mathbf{D}\mathbf{W} + \mathbf{N}\mathbf{F}\mathbf{W}^2 + \mathbf{N}\mathbf{O}\mathbf{W}$ memory.

N.2. Jacobian

Computing the Jacobian costs $\mathbf{O}$ times the cost of VJP, hence time is $\mathbf{ON}$ ([cost of all intermediate layers] + [cost of the top layer]) $= \mathbf{ON}\left(\left[\mathbf{LDFW}^2\right] + [\mathbf{OW}]\right) \sim \mathbf{NLODFW}^2 + \mathbf{NO}^2\mathbf{W}$. Memory is [all weights] + $\mathbf{N}$ [activations in all layers] + $\mathbf{ON}$ [a single weight matrix] + $\mathbf{ON}$ [activations in a single layer] $= \left[\mathbf{LFW}^2 + \mathbf{OW}\right] + \mathbf{N}\left[\mathbf{LDW} + \mathbf{O}\right] + \mathbf{ON}\left[\mathbf{FW}^2 + \mathbf{OW}\right] + \mathbf{ON}\left[\mathbf{DW}\right] \sim \mathbf{LW}^2 + \mathbf{NLDW} + \mathbf{NODW} + \mathbf{NOFW}^2 + \mathbf{NO}^2\mathbf{W}$

> Jacobian costs $\mathbf{NLODFW}^2 + \mathbf{NO}^2\mathbf{W}$ time and $\mathbf{LW}^2 + \mathbf{NOFW}^2 + \mathbf{NO}^2\mathbf{W} + \mathbf{NLDW} + \mathbf{NODW}$ memory.

N.3. Jacobian contraction

Since weight matrices are increased by $\mathbf{F}$, the contraction cost goes up to $\mathbf{N}^2\mathbf{LO}^2\mathbf{FW}^2$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOFW}^2$ memory. The cost of computing the Jacobian is also modified (§N.2), which results in $\mathbf{N}^2\mathbf{LO}^2\mathbf{FW}^2 + \mathbf{NLODFW}^2 + \mathbf{NO}^2\mathbf{W} \sim \mathbf{N}^2\mathbf{LO}^2\mathbf{FW}^2 + \mathbf{NLODFW}^2$ time and $\left(\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOFW}^2\right) + \left(\mathbf{LFW}^2 + \mathbf{NOFW}^2 + \mathbf{NO}^2\mathbf{W} + \mathbf{NLDW} + \mathbf{NODW}\right) \sim \mathbf{N}^2\mathbf{O}^2 + \mathbf{NOFW}^2 + \mathbf{NO}^2\mathbf{W} + \mathbf{NLDW} + \mathbf{NODW} + \mathbf{LFW}^2$ memory.

> Jacobian contraction costs $\mathbf{N}^2\mathbf{LO}^2\mathbf{FW}^2 + \mathbf{NLODFW}^2$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOFW}^2 + \mathbf{NO}^2\mathbf{W} + \mathbf{NLDW} + \mathbf{NODW} + \mathbf{LFW}^2$ memory.

N.4. Structured derivatives

Convolution is Constant block-diagonal along the output channel axis with $\mathbf{C} = \mathbf{W}$, $\mathbf{P} = \mathbf{FW}^2$, $\mathbf{Y} = \mathbf{DW}$. Substituting this in Table 4, the cost of contraction is the minimum of the costs from Table 8. If we exclude the **Inside-out** contraction path from `np.einsum` (in practice it will always select the best out of three) for simplicity, we can and conclude that for $\mathbf{L}$ layers, the time cost of the contraction is at most $\mathbf{N}^2\mathbf{LO}^2 \min\left(\mathbf{FW}^2, \mathbf{DW}\right) + \mathbf{DFNLOW}^2$, as the minimum cost between the **Outside-in** and **Left-to-right**. Note that this dominates the time cost of the Jacobian from §N.2, so we don't need to modify it further. Memory due to Jacobian computation is [all weights] + $\mathbf{N}$ [activations in all layers] + $\mathbf{NO}$ [activations in a single layer] + [size of primitive derivatives] $= \left[\mathbf{LFW}^2 + \mathbf{OW}\right] + \mathbf{N}\left[\mathbf{LDW} + \mathbf{O}\right] + \mathbf{NO}\left[\mathbf{DW}\right] + \mathbf{N}\left[\mathbf{DW}\right] \sim \mathbf{LFW}^2 + \mathbf{NLDW} + \mathbf{NODW}$. Again, as in §L.4, and unlike other methods, we do not need to compute or store $\partial f / \partial \theta^l$ derivatives, allowing to avoid the $\mathbf{NOFW}^2 + \mathbf{NO}^2\mathbf{W}$ extra memory overhead. However, we need to add the cost of storing the (subarray of) primitive Jacobians $\partial y / \partial \theta$, while have the size of $\mathbf{J} = \mathbf{YP}/\mathbf{C}^2 = \mathbf{DFW}$, hence the extra cost is $\mathbf{NDFW}$.

> Structured derivatives cost $\mathbf{N}^2\mathbf{LO}^2 \min(\mathbf{FW}^2, \mathbf{DW}) + \mathbf{NLODFW}^2$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NDFW} + \mathbf{NLDW} + \mathbf{NODW} + \mathbf{LFW}^2$ memory.

Structure of $\partial y / \partial \theta \downarrow$	**Outside-in**	**Left-to-right**	**Inside-out**
Constant block-diagonal	$\mathbf{NODFW}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{FW}^2$	$\mathbf{NODFW}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{DW}$	$\mathbf{N}^2\mathbf{D}^2\mathbf{FW}^2 + \mathbf{N}^2\mathbf{OD}^2\mathbf{W} + \mathbf{N}^2\mathbf{O}^2\mathbf{DW}$

Table 8: **Time complexity of contracting $\Theta_\theta^{l,k_1,k_2}(f_1, f_2)$ corresponding to a CNN primitive** obtained by substituting $\mathbf{Y} = \mathbf{DW}$, $\mathbf{C} = \mathbf{W}$, and $\mathbf{P} = \mathbf{FW}^2$ into Table 4. The time cost of Structured derivatives are the minimum of the three entries due to using optimal contraction path by `np.einsum`.

N.5. NTK-vector products

The cost of this approach is asymptotically equivalent to the cost of Jacobian (§N.2), since it consists of $\mathbf{O}$ VJPs followed by $\mathbf{O}$ (cheaper) JVPs. Therefore it costs $\mathbf{LODFW}^2 + \mathbf{O}^2\mathbf{W}$ time and $\mathbf{LFW}^2 + \mathbf{OFW}^2 + \mathbf{O}^2\mathbf{W} + \mathbf{LDW} + \mathbf{ODW}$ memory.

Batched inputs. In a batched setting Eq. (37) is repeated for each pair of inputs, and therefore time increases by a factor of $\mathbf{N}^2$ to become $\mathbf{N}^2\mathbf{LODFW}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{W}$. Memory only grows linearly in $\mathbf{N}$ (except for storing the result of size $\mathbf{N}^2\mathbf{O}^2$), by similar argument to §L.5, i.e. becomes $\mathbf{N}^2\mathbf{O}^2 + \left(\mathbf{LFW}^2 + \mathbf{NOFW}^2 + \mathbf{NO}^2\mathbf{W} + \mathbf{NLDW} + \mathbf{NODW}\right)$ total memory.

> NTK computation as a sequence of NTK-vector products costs $\mathbf{N}^2\mathbf{LODFW}^2 + \mathbf{N}^2\mathbf{O}^2\mathbf{W}$ time and $\mathbf{N}^2\mathbf{O}^2 + \mathbf{NOFW}^2 + \mathbf{LFW}^2 + \mathbf{NLDW} + \mathbf{NODW}$ memory.

O. Experimental details

All experiments were performed in JAX (Bradbury et al., 2018) using 32-bit precision.

Throughout this work we assume the cost of multiplying two matrices of shapes (M, K) and (K, P) to be MKP. While there are faster algorithms for very large matrices, the XLA compiler (used by JAX, among other libraries) does not implement them, so our assumption is accurate in practice.

Hardware. **CPU** experiments were run on Dual 28-core Intel Skylake CPUs with at least 240 GiB of RAM. **NVIDIA V100** and **NVIDIA P100** used a respective GPU with 16 GiB GPU RAM. **TPUv3** and **TPUv4** have 8 and 32 GiB of RAM respectively, and use the default 16/32-bit mixed precision.

Fig. 1 and Fig. 3: a 10-layer, ReLU FCN was constructed with the Neural Tangents (Novak et al., 2020) `nt.stax` API. Defeault settings (weight variance 1, no bias) were used. Individual inputs x had size 3. Jacobian contraction was evaluated using `nt.empirical_ntk_fn` with `trace_axes=(), diagonal_axes=(), vmap_axes=0`. Jacobian was evaluated using `jax.jacobian` with a `vmap` over inputs x. For time measurements, all functions were `jax.jit`ted, and timing was measured as the average of 100 random samples (compilation time was not included). For FLOPs, the function was not JITted, and FLOPs were measured on CPU using the `utils.get_flops` function that is released together with our code.[6]

6. The XLA team has let us know that if JITted, the FLOPs are currently correctly computed only on TPU, but are incorrect on other platforms. Therefore we compute FLOPs of non-JITted functions.

Fig. 2 and Fig. 4: for ResNets, implementations from Flax (Heek et al., 2020) were used, specifically `flax.examples.imagenet.models`. For WideResNets, the code sample from Novak et al. (2020) was used.[7] For all other models, we used implementations from https://github.com/google-research/vision_transformer. Inputs were random arrays of shapes $224 \times 224 \times 3$. All models were JITted. All reported values are averages over 10 random samples. For each setting, we ran a grid search over the batch size $\mathbf{N}$ in $\left\{2^k\right\}_{k=0}^{9}$, and reported the best time divided by $\mathbf{N}^2$, i.e. best possible throughput in each setting.

7. We replaced `stax.AvgPool((8, 8)), stax.Flatten()` with `stax.GlobalAvgPool()`.