Tensor Programs IIb: Architectural Universality of Neural Tangent Kernel Training Dynamics

Greg Yang [1 *] Etai Littwin [2 *]

Abstract

Yang (2020a) recently showed that the Neural Tangent Kernel (NTK) at initialization has an infinite-width limit for a large class of architectures including modern staples such as ResNet and Transformers. However, their analysis does not apply to training. Here, we show the same neural networks (in the so-called *NTK parametrization*) during training follow a kernel gradient descent dynamics in function space, where the kernel is the infinite-width NTK. This completes the proof of the *architectural universality* of NTK behavior. To achieve this result, we apply the *Tensor Programs* technique: Write the entire SGD dynamics inside a Tensor Program and analyze it via the *Master Theorem*. To facilitate this proof, we develop a graphical notation for Tensor Programs. See the full version of our paper at `arxiv.org/abs/2105.03703`.

1. Introduction

(Jacot et al., 2018)'s pioneering work showed that a multilayer perceptron (MLP) trained by gradient descent (GD) evolves like a linear model. This spurred a flurry of research papers using this insight to tackle the core questions in deep learning theory, from optimization to generalization in both finite and infinite width regimes. (Jacot et al., 2018)'s argument consists of two observations:

NTKINIT For the output of a network $f(\xi; w)$ with parameters w given example ξ, (Jacot et al., 2018) identified the kernel $\mathcal{K}(\xi, \bar{\xi}) = \langle \nabla f(\xi; w), \nabla f(\bar{\xi}, w) \rangle$, known as the *Neural Tangent Kernel (NTK)*. They showed that if f is parametrized and initialized appropriately, then $\mathcal{K}$ converges to a deterministic kernel $\mathring{\mathcal{K}}$ as the width of the network tends to infinity.

NTKTRAIN As the infinitely wide network is trained by gradient descent, the NTK remains frozen in its initial state, and the network evolves by kernel gradient descent with kernel $\mathring{\mathcal{K}}$.

*Equal contribution [1]Microsoft Research [2]Apple Research. Correspondence to: Greg Yang <gregyang@microsoft.com>, Etai Littwin <elittwin@apple.com>.

Proceedings of the 38th International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

In (Yang, 2020a), the NTKINIT property was proven to hold for *standard architectures*, meaning any composition of MLPs, recurrent neural networks (RNN), LSTMs (Hochreiter & Schmidhuber, 1997), gated recurrent unit (GRU) (Cho et al., 2014), convolutions (Fukushima, 1980; 1975; Lecun et al., 1998; 2000; Rumelhart et al., 1986), residual connections (He et al., 2016; Huang et al., 2017), batch normalization (Ioffe & Szegedy, 2015), graph neural networks (Bruna et al., 2014; Defferrard et al., 2016; Duvenaud et al., 2015; Henaff et al., 2015; Kipf & Welling, 2017) and attention (Bahdanau et al., 2015; Vaswani et al., 2017), along with arbitrary weight sharing between components. More generally, it holds for any architecture expressible in a so-called *Tensor Program* (Yang, 2019b;a; 2020a;b), of which the standard architectures are a subset. However, their reasoning is limited to initialization only.

A statement is *architecturally universal* if it holds for any *reasonable* neural architecture. This is an informal property, but here we will formalize it by taking *reasonable* to be "expressable in Tensor Programs." By the expressiveness of such programs (Yang, 2019a; 2020a), architectural universality is a fairly robust notion that covers present (and, we expect, future) architectures comprehensively. In this terminology, (Yang, 2020a) showed that NTKINIT is architecturally universal.

Our Contribution We show the architectural universality of the entire NTK theory by proving NTKTRAIN for the same architectures discussed above, including all standard architectures. In the process, we introduce a new graphical form of Tensor Programs that is both required in our proofs and useful for the pedagogy of Tensor Programs.

The Tensor Program Series This paper follows (Yang, 2019b;a; 2020a;b; Yang & Hu, 2020) in the series. While we number this paper "IIb" right after (Yang, 2020a), we actually need the complete theoretical foundation developed in III (Yang, 2020b). See Footnote 21 for more details.

2. Background

Let $f(\xi; w) \in \mathbb{R}$ denote the (scalar) output of a neural network parameterized by w, given example ξ. To understand how the output changes with a slight change in the network parameters $w_0 - \delta w$, we may naively expand the network function using the first order Taylor expansion around a base point w_0:

$$f(\xi, w_0 - \delta w) - f(\xi; w_0) \approx \langle \nabla_w f(\xi, w_0), \delta w \rangle. \quad (1)$$

Under the SGD algorithm, the weight update δw is given by the gradient $\delta w = -\eta \chi(\hat{\xi}) \nabla_w f(\hat{\xi}; w_0)$ where $\chi(\hat{\xi})$ is the loss derivative, $\hat{\xi}$ is a sample from the training set, and η is the learning rate. Plugging into Eq. (1), we get:

$$f(\xi, w_0 - \delta w) - f(\xi; w_0) \approx -\eta \chi(\hat{\xi}) \mathcal{K}(\xi, \hat{\xi}). \quad (2)$$

where $\mathcal{K}(\xi, \hat{\xi}) = \langle \nabla_w f(\xi; w_0), \nabla_w f(\hat{\xi}; w_0) \rangle$ is the NTK. The NTK theory of infinitely wide neural networks as first proposed by (Jacot et al., 2018) boils down to the the following observations: *When the width of f tend to infinity, the NTK $\mathcal{K}$ converges to a fixed kernel $\mathring{\mathcal{K}}$ at random initialization, independent of the specific instantiation of the weights, and remains frozen during the optimization process.* Eq. (2) then gives an accurate description of the output evolution with if we substitue $\mathcal{K}$ with $\mathring{\mathcal{K}}$. The seemingly complex optimization trajectory of SGD therefore reduce to the convex trajectory of kernel gradient descent with a time-independent kernel $\mathring{\mathcal{K}}$.
Consider the output of the network $f \in \mathbb{R}^D$ on the full training dataset. As shown in (Jacot et al., 2018), when the $L2$ loss is used the evolution of the output f_t at time t under continuous time GD (i.e. gradient flow) takes a simple form:

$$f_t - f^\star = e^{-\eta \mathring{\mathcal{K}} t}(f_0 - f^\star).$$

where $\mathring{\mathcal{K}} \in \mathbb{R}^{D \times D}$ is the full NTK matrix evaluated on the training data, $f^\star$ is the label function, and f_0 is the output at initialization. Hence, provided $\mathring{\mathcal{K}}$ is full rank, as $t \to \infty$ we have that $f_t \to f^\star$, and the network can fit the training data perfectly.

Previous Approaches vs Ours A common theme in showing NTKTRAIN for MLP is to derive high-probability bounds on the deviation of the NTK $\mathcal{K}$ from its initial value after training (e.g. Allen-Zhu et al. (2018); Du et al. (2018); Zou et al. (2018)).[1] Obtaining these bounds usually requires developing ad hoc methods on a per-architecture basis, hindering the scalability of the method to other settings. In the present work we take a more holistic approach, leveraging the recently developed Tensor Programs framework (Yang, 2019b;a; 2020a;b). It consists of two layers of arguments: 1) The bottom layer analyzes how the *distribution of (pre-)activations* change throughout the course of training; this crucially leverages the mathematical machinery of the Tensor Programs Master Theorem.[2] 2) The top layer packages these insights systematically via the notion of *paths* so as to apply to any architecture expressible by a Tensor Program. We will illustrate 1) through examples in Section 3 and 2) through figures in Section 5.1.

[1]In the original NTK paper (Jacot et al., 2018), the limit is taken as each layer width goes to infinity sequentially, which already doesn't make sense for weight-tied architectures like RNNs.

[2]In particular, we need to use the Master Theorem in (Yang, 2020b), so (Yang, 2020a) could not have obtained NTKTRAIN at the same time as NTKINIT.

Setup and Notations In this paper, we will consider the architecture (including depth), data, and training time to be fixed as width $n \to \infty$.[3] We describe common notations used in the remainder of the paper. For simplicity, we will consider SGD with batch size 1 and learning rate η (often set to 1 WLOG).[4] We use ξ_t to denote the input and $\mathcal{L}_t$ to denote the loss function (absorbing the label) at step t. More generally, subscript t on any symbol means *time* t. However, for brevity, we abuse notation and shorthand f_t for $f_t(\xi_t)$, and, for any (pre-)activation x, x_t for $x_t(\xi_t)$.[5] We will also write χ_t for the loss derivative $\mathcal{L}'_t(f_t)$. For any vector $x(\xi)$ we define $\delta x_{t+1}(\xi) \overset{\text{def}}{=} \sqrt{n}\big(x_{t+1}(\xi) - x_t(\xi)\big)$ and $dx(\xi) \overset{\text{def}}{=} \sqrt{n}\frac{\partial f(\xi)}{\partial x(\xi)}$. We will track the evolution of f on an arbitrary input $\tilde{\xi}$.[6] Similar to above, we shorthand $\tilde{x}_t$, $\tilde{f}_t$ for $x_t(\tilde{\xi})$, $f_t(\tilde{x})$.

3. Motivating Examples

The purpose of this section is to illustrate our key ideas via simple, intuitive examples without diving into the specifics of Tensor Programs. In the process, we will gain insight into how randomness from initialization propagates over the course of training. As these examples intend to provide the reader with the proper intuition, we use informal arguments alone and relegate all formal statements to the appendix. For brevity, we will gloss over minor details or routine calculations, but interested readers can see Appendix A for these omissions.

Key Idea It turns out that the random initialization and the overparametrization of weights cause each (pre-)activation vector $x_t(\xi) \in \mathbb{R}^n$, its gradient $dx_t(\xi) \in \mathbb{R}^n$, and its (scaled) change $\delta x_t(\xi) \in \mathbb{R}^n$ every time step t to have roughly iid coordinates, not just initially but *throughout training*.[7] Then, as we shall demonstrate through the examples below, to track the evolution of the neural network function, it suffices to track the evolution of the coordinate distributions of $x(\xi)$, $dx(\xi)$, $\delta x(\xi)$. We write $Z^{x(\xi)}$, $Z^{dx(\xi)}$, $Z^{\delta x(\xi)} \in \mathbb{R}$ for the random variables corresponding to such coordinate distributions.[8]

[3]They will affect the rate of convergence to the infinite-width limit, but since we are only concerned with whether convergence occurs, they do not appear in our theorem statements here.

[4]This generalizes readily to any batch size and learning rate.

[5]We will not refer to the *function* $x_t : \xi \to x_t(\xi)$ (likewise for f_t, χ_t), so this abuse of notation should cause no confusion.

[6]It might help to think of $\tilde{\xi}$ as some *test sample*, but it can also fall in the training set.

[7]This is a consequence of the Tensor Program Master Theorem.

[8]As we will explain below, different Zs may correlate, reflecting correlations between corresponding vectors.

Our goal is to derive, from these insights,

Claim 3.1. *In the large width limit, $\tilde{f}_t = f_t(\tilde{\xi})$ changes by*

$$\lim_{n\to\infty} \tilde{f}_{t+1} - \tilde{f}_t = -\mathring{\chi}_t \mathring{\mathcal{K}}(\tilde{\xi}, \xi_t) \tag{3}$$

at step t, where $\mathring{\mathcal{K}}$ is the limiting NTK of the architecture and $\mathring{\chi}_t = \mathcal{L}'_t(\lim_n f_t)$ is the loss derivative.

We start with an example derivation for 1-hidden-layer MLP, before moving on to 2-hidden-layers, where the mathematics quickly become much more involved.

3.1. 1 Hidden Layer

Consider a 1-hidden-layer network with nonlinearity ϕ:

$$f = V^\top x, \quad x = \phi(h), \quad h = U\xi$$

where $\xi \in \mathbb{R}^d$, $U = \frac{u}{\sqrt{d}} \in \mathbb{R}^{n\times d}$, $V = \frac{v}{\sqrt{n}} \in \mathbb{R}^{n\times 1}$, for trainable parameter tensor u, initialized iid from $\mathcal{N}(0,1)$. In the interest of clarity we assume the output layer is not trained, and $d = 1$.

For a vector $v \in \mathbb{R}^n$, let $v = \Theta(n^a)$ mean that "v has coordinates of order n^a when n is large"[9]; likewise for $o(n^a)$, etc. Recall the notations $x_t = x_t(\xi_t), \tilde{x}_t = x_t(\tilde{\xi}), \delta x_t = \sqrt{n}(x_t - x_{t-1})$ and likewise for $h_t, \tilde{h}_t, \delta h_t$. The key insights are as follows:

Preliminary Calculations It turns out $\tilde{x}_{t+1} - \tilde{x}_t = \Theta(1/\sqrt{n}) = o(1)$ so $\delta\tilde{x}_{t+1} = \sqrt{n}(\tilde{x}_{t+1} - \tilde{x}_t))$ has $\Theta(1)$ coordinates. Likewise for $\delta\tilde{h}_{t+1}$. Consequently, for any t and input ξ, by telescoping,

$$h_t(\xi) = h_0(\xi) + o(1). \tag{4}$$

Using $\nabla_u f = \frac{1}{\sqrt{n}} dh_t \xi_t^\top$ and $dh_t = \phi'(h_t) \odot v$, we have:

$$\delta\tilde{h}_{t+1} = -\chi_t \xi_t^\top \tilde{\xi} \phi'(h_t) \odot v. \tag{5}$$

Also, $\delta\tilde{x}_{t+1} = \sqrt{n}\big(\phi(\tilde{h}_t + \frac{\delta\tilde{h}_{t+1}}{\sqrt{n}}) - \phi(\tilde{h}_t)\big)$. Since $\delta\tilde{h}_{t+1} = \Theta(1)$, by intuitive Taylor expansion, we have

$$\delta\tilde{x}_{t+1} \approx \phi'(\tilde{h}_t) \odot \delta\tilde{h}_{t+1}. \tag{6}$$

The change in the output on example $\tilde{\xi}$ from step t to step $t+1$ is given by:

$$\tilde{f}_{t+1} - \tilde{f}_t = V^\top(\tilde{x}_{t+1} - \tilde{x}_t) = v^\top \delta\tilde{x}_{t+1}/n \tag{7}$$

IID Coordinates By definition v has iid coordinates. It turns out $h_t(\xi), \delta h_t(\xi)$ (likewise for x) all have approx. iid coordinates of size $\Theta(1)$ as well.[10] Let $Z^{\delta\tilde{x}_t}, Z^v$ denote the random variables encoding the corresponding coordinate distributions; likewise for the other vectors. Note that $Z^{\delta\tilde{x}_t}, Z^v$ will in general be correlated, reflecting the coordinatewise correlation between v and $\delta\tilde{x}_t$.

[9] More rigorously, we mean that $\|v\|^2/n = \Theta(n^{2a})$. Note this is *different* from the common interpretation that $\|v\| = \Theta(n^a)$.

Law of Large Numbers and Eqs. (5) and (7) imply,

$$\lim_{n\to\infty} \tilde{f}_{t+1} - \tilde{f}_t = \mathbb{E}\, Z^v Z^{\delta\tilde{x}_{t+1}} \tag{8}$$

$$= -\mathring{\chi}_t \xi_t^\top \tilde{\xi}\, \mathbb{E}\, \phi'(Z^{\tilde{h}_t})\phi'(Z^{h_t})(Z^v)^2. \tag{9}$$

where $\mathring{\chi}_t = \mathcal{L}'_t(\lim_n f_t)$ as in Claim 3.1.

Kernel Expression as Gaussian Expectation By Eq. (4), in the $n \to \infty$ limit, $Z^{\tilde{h}_t} = Z^{h_0(\tilde{\xi})}$ and $Z^{h_t} = Z^{h_0(\xi_t)}$. They are independent from Z^v and jointly Gaussian with variances $\|\tilde{\xi}\|^2, \|\xi_t\|^2$ and covariance $\tilde{\xi}^\top \xi_t$. So (using the initialization of v to simplify $\mathbb{E}(Z^v)^2 = 1$),

$$\lim_{n\to\infty} \tilde{f}_{t+1} - \tilde{f}_t = -\mathring{\chi}_t \xi_t^\top \tilde{\xi}\, \mathbb{E}\, \phi'(Z^{h_t})\phi'(Z^{\tilde{h}}) \tag{10}$$

This can easily be seen to be Eq. (3) (recall we assumed for simplicity the output layer V is not trained).

Summary Our strategy so far has been computing the form of $Z^{\delta\tilde{x}_t}$ and plugging it into Eq. (8) to compute the dynamics of the output in the limit. Note that our approach differs from previous work which mainly focus on proving a bound on the change of the NTK post training. As the architecture gets more complex, bounding the NTK movement becomes quite complex, but our approach easily scales due to the automation provided by the *Tensor Programs* framework (see Section 4).

In the previous example, the coordinate distribution $Z^{\tilde{x}_t}$ took a fairly simple form, which allowed us to intuitively compute the expectation $\mathbb{E}\, Z^{\tilde{x}_t} Z^v$. Before introducing a method for computing coordinate distributions in a general architecture, we move on to a slightly more involved architecture, with the intention of highlighting the intuition behind the general case.

3.2. 2 Hidden Layers

In this example we consider a model of the form:

$$f = V^\top x, \quad x = \phi(h), \quad h = Wz$$
$$z = \phi(g), \quad g = U\xi$$

where $U = \frac{u}{\sqrt{d}} \in \mathbb{R}^{n\times d}$, $W = \frac{w}{\sqrt{n}} \in \mathbb{R}^{n\times n}$, $V = \frac{v}{\sqrt{n}} \in \mathbb{R}^{n\times 1}$, for trainable parameters u, w, initialized iid from a

[10] Technically, they have iid coordinates only after conditioning on the initial function (GP) f_0. Likewise, when we take expectation in this example (e.g. Eqs. (9) and (17)), it's a conditional expectation of this kind. See Appendix D.1.1 for a rigorous treatment. However, to convey the main intuition, we gloss over this technicality here.

normal distribution. As before we assume the last layer is not trained, and $d = 1$.

Again, we want to establish Claim 3.1 for this model. As in the 1-hidden-layer example, the dynamics of the output in the limit is still given by Eq. (8). This time around, the second hidden layer adds nontrivial complexity when evaluating the expectation $\mathbb{E}\, Z^v Z^{\delta \tilde{x}_{t+1}}$. As we shall see, this complexity arises from the dependency of $\tilde{x}$ on the $n \times n$ matrices w and $w^\top$, which will make it *wrong* to naively apply LLN arguments. Resolving this complexity will pave the way to a general strategy which we will then be able to apply in any arbitrary architecture. We now apply the same insights as in the 1-hidden-layer MLP. Namely:

- Eq. (4) continues to hold with h replaced by any of $\{x, h, z, g\}$.
- After some brief calculations, with dh_t denoting the scaled gradient $\sqrt{n}\nabla_{h_t} f$,

$$\delta\tilde{g}_{t+1} \approx -\chi_t \xi_t^\top \tilde{\xi} \phi'(g_t) \odot \left(W^\top dh_t\right) \tag{11}$$

$$\delta\tilde{h}_{t+1} \approx W\delta\tilde{z}_{t+1} - \chi_t \frac{z_t^\top \tilde{z}_t}{n} \phi'(h_t) \odot v. \tag{12}$$

- As in the 1-hidden-layer case, for all $x \in \{g, z, h, x\}$, $x_t(\xi), \delta x_t(\xi)$ have iid coordinates of size $\Theta(1)$, as does v by definition.[11] Let $Z^{\delta\tilde{x}_t}, Z^v$ denote the (generally correlated) random variables encoding the corresponding coordinate distributions.
- As in Eq. (7), by naive Taylor expansion we have:

$$\delta\tilde{z}_{t+1} \approx \phi'(\tilde{g}_t) \odot \delta\tilde{g}_{t+1}. \tag{13}$$

- Eqs. (6) and (7) in the 1-hidden-layer case continue to hold here. Then by Eq. (12) and Law of Large Numbers,

$$\lim_{n\to\infty} \tilde{f}_{t+1} - \tilde{f}_t = \mathbb{E}\, Z^v Z^{\delta\tilde{x}_{t+1}} \tag{14}$$

$$= -\mathring{\chi}_t \xi_t^\top \tilde{\xi}\, \mathbb{E}[Z^{z_t} Z^{\tilde{z}}]\, \mathbb{E}[\phi'(Z^{h_t})\phi'(Z^{\tilde{h}})] \tag{15}$$

$$- \mathring{\chi}_t \xi_t^\top \tilde{\xi}\, \mathbb{E}[\phi'(Z^{\tilde{h}_t}) Z^{W\delta\tilde{z}_{t+1}} Z^v]. \tag{16}$$

where $\mathring{\chi}_t = \mathcal{L}'_t(\lim_n f_t)$ as in Claim 3.1.

In this expression, the first term (Eq. (15)) can easily be seen to correspond to the contribution from w to the NTK. It remains to show that the second (Eq. (16)) corresponds to the contribution from u.

Challenge of Analyzing Eq. (16) To do this, we must reason about the coordinate distribution of $W\delta\tilde{z}_{t+1}$ (encoded by random variable $Z^{W\delta\tilde{z}_{t+1}}$) and compute the expectation in Eq. (16). To understand why this represents a greater challenge than it might first appear, note that from $\delta\tilde{z}_{t+1} \approx \phi'(\tilde{g}_t) \odot \delta\tilde{g}_{t+1}$ (Eq. (13)), the term $W\delta\tilde{z}_{t+1}$ hides within itself a dependency on $W^\top dh_t$ through $\delta\tilde{g}$ (Eq. (11)). While at $t = 0$, we may assume $W^\top$ be independent of W and obtain the correct results (Gradient Independent Assumption (Yang & Schoenholz, 2017; Yang, 2020a)), this is no longer the case for $t > 0$: $Z^{W\delta\tilde{z}_{t+1}}$ will be nontrivially correlated with $\phi'(Z^{\tilde{h}_t})$ and Z^v (which would be false if $W^\top$ can be assumed independent of W). We will give some intuition why later in Eq. (20). Now, what is this dependence exactly?

Claim 3.2. *Based on the above discussion and some easy calculations, $\delta\tilde{z}_{t+1}$ can be written as $\Phi(W^\top dh_t)$ for some $\Phi : \mathbb{R} \to \mathbb{R}$ applied coordinatewise (which will depend on other vectors not of the form $W^\top \bullet$). Then it turns out*[11]

$$Z^{W\delta\tilde{z}_{t+1}} = G + Z^{dh_t}\, \mathbb{E}\, \frac{\partial Z^{\delta\tilde{z}_{t+1}}}{\partial Z^{W^\top dh_t}}, \tag{17}$$

where G is some Gaussian variable independent from Z^v, and $\frac{\partial Z^{\delta\tilde{z}_{t+1}}}{\partial Z^{W^\top dh_t}} \overset{def}{=} \Phi'(Z^{W^\top dh_t})$.

Getting Claim 3.1 Thus, from Eqs. (11) and (13), it follows that:

$$Z^{\delta\tilde{z}_{t+1}} = -\mathring{\chi}_t \xi_t^\top \tilde{\xi} \phi'(Z^{\tilde{g}_t}) \phi'(Z^{g_t}) Z^{W^\top dh_t} \tag{18}$$

$$\mathbb{E}\, \frac{\partial Z^{\delta\tilde{z}_{t+1}}}{\partial Z^{W^\top dh_t}} = -\mathring{\chi}_t \xi_t^\top \tilde{\xi}\, \mathbb{E}[\phi'(Z^{\tilde{g}_t})\phi'(Z^{g_t})]. \tag{19}$$

Plugging into Eqs. (14) and (17), followed by some straightforward calculation, then yields Claim 3.1.

Intuition behind Claim 3.2 Eq. (17) may appear cryptic at first, so let's give some intuition using an example. Suppose Φ in Claim 3.2 is actually identity. For brevity, we set $\mathbf{x} = dh_t, \mathbf{y} = \delta\tilde{z}_{t+1} \in \mathbb{R}^n$. Then, following straightforward calculation, $W\mathbf{y} = WW^\top\mathbf{x}$ has coordinates

$$(W\mathbf{y})_\alpha = \sum_{\gamma\neq\alpha} \mathbf{x}_\gamma \sum_{\beta=1}^n W_{\alpha\beta} W_{\gamma\beta} + \sum_{\beta=1}^n (W_{\alpha\beta})^2 \mathbf{x}_\alpha \tag{20}$$

Now, the second sum converges via LLN to $\mathbf{x}_\alpha$ as $n \to \infty$. On the other hand, the first sum will converge via CLT to $\mathcal{N}(0, \lim \|\mathbf{x}\|^2/n)$. Thus, in terms of Zs, we have

$$Z^{W\mathbf{y}} = G + Z^{\mathbf{x}} = G + Z^{\mathbf{x}}\, \mathbb{E}\, \Phi' \tag{21}$$

for some Gaussian G; this corresponds directly to Eq. (17).[12] For general Φ, a similar intuition applies after Taylor expansion of Φ.

[11]Again, this is technically true only after conditioning on f_0; see Footnote 10.

[12]This example was worked out in (Yang, 2020a;b) as well, though in different contexts. Readers needing more explanation may see those works.

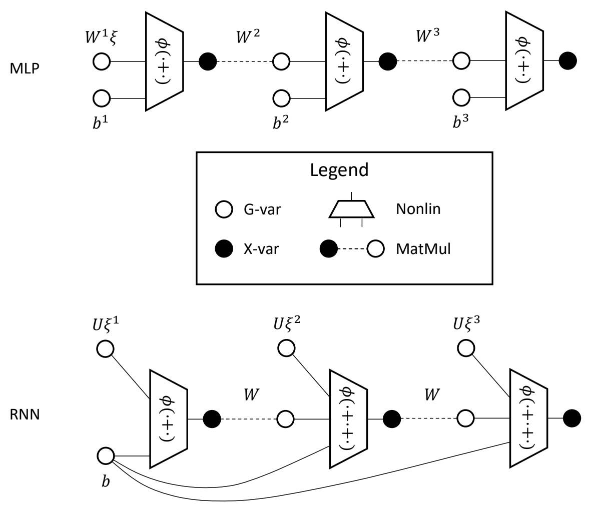

Figure 1: **Graphical form of NETSOR⊤ programs for MLP and RNN.** An empty node corresponds to a G-var (an initial vector or a vector created by MATMUL), while a filled node corresponds to an X-var (a vector created by NONLIN). Each dashed edge corresponds to matrix multiplication by the labeled matrix. Each gate (connected by solid edges) represents a NONLIN application. The computation follows the direction of the gates (left-to-right here). Note we have $W^1\xi$ instead of ξ as an initial vector because we only allow $\mathbb{R}^n$ vectors; likewise for $U\xi^i$. For the same reason, we don't express the network output in this graph.

Summary This 2-hidden-layer example proceeded much the same as the 1-hidden-layer case, with the main exception of analyzing the interaction of the $n \times n$ Gaussian matrix W and $W^\top$ (Eq. (16)) that occurs after taking at least 1 step of SGD. This was absent in the 1-hidden-layer case because each weight matrix has at most one side tending to infinity. Such analysis is crucial to obtaining the right results, as assuming $W^\top$ be independent from W would imply f does not move from initialization.[13]

It turns out these two examples have essentially covered all of the core ideas needed to extend the analysis into arbitrary architectures. To formalize and scale up our calculations, we now turn to the *Tensor Programs* framework.

4. Tensor Programs

So far, our results have been obtained by unrolling the SGD updates on toy models with specific architectures, and using informal arguments. Obviously, these computations quickly become unmanageable when the architecture becomes more complex. The sheer amount of architectural innovations that have sprung up in recent years requires us to adopt a much more general formulation of our results. To that end, we adopt the Tensor Programs (TP) framework developed in (Yang, 2019a; 2020a;b). In a nutshell, it provides a language for describing typical computations done in the context of neural networks, such as forward and backward propagation. It is simultaneously simple and expressive, covering all standard architectures (Yang, 2019a; 2020a). Here we review two basic forms of Tensor Programs, NETSOR⊤ and NETSOR⊤$^+$.

Definition 4.1. A NETSOR⊤ program is just a sequence of $\mathbb{R}^n$ vectors inductively generated via one of the following instructions from an initial set $\mathcal{V}$ of random $\mathbb{R}^n$ vectors and a set $\mathcal{W}$ of random $n \times n$ matrices

NONLIN For $x^1, \ldots, x^k \in \mathbb{R}^n$ in the program and any $\psi : \mathbb{R}^k \to \mathbb{R}$, we can generate $\psi(x^1, \ldots, x^k) \in \mathbb{R}^n$

MATMUL Given $W \in \mathbb{R}^{n\times n}$ and $x \in \mathbb{R}^n$, we can generate $Wx \in \mathbb{R}^n$ or $W^\top x \in \mathbb{R}^n$

Graphical Form We propose to represent a NETSOR⊤ program as a computational graph, where each node in the graph represents vectors (initial or generated), each (dashed) edge represents a MATMUL, and each gate represents a NONLIN. For example, Fig. 1 shows the computation graphs expressing (the forward passes of) an MLP and an RNN. We can also express the backpropagation as well (see Fig. 6). Graphically, the initial vectors are the empty nodes with only one edge coming out, toward the direction of computation. The matrices correspond to (the labels of) the dashed edges. We can also define the *output vectors* to correspond to the nodes that have only one edge coming out, *against* the direction of computation.

Neural Network Representation Each NETSOR⊤ program can be thought of as computing a function $(\mathbb{R}^n)^{\mathcal{V}} \times (\mathbb{R}^{n\times n})^{\mathcal{W}} \to \mathbb{R}^{\mathcal{Y}}$ taking an instantiation of the initial vectors $\mathcal{V}$ and matrices $\mathcal{W}$ and computing the values of all output vectors $\mathcal{Y}$. We can say a program *represents a network* if it computes the *body* of it (without the input and output layers), as exemplified by Fig. 1. This is formalized below.

Definition 4.2. Consider a neural network $f : (\mathbb{R}^d)^k \to \mathbb{R}$ with input embedding matrices $U^1, \ldots, U^k \in \mathbb{R}^{n\times d}$ (not necessarily distinct) and readout matrix $V \in \mathbb{R}^n$, so that $f(\xi^1, \ldots, \xi^k) = V^\top \Phi(U^1\xi^1, \ldots, U^k\xi^k; \Theta)$ for some function $\Phi(x^1, \ldots, x^k; \Theta)$ with parameters Θ. We say a NETSOR⊤ program *represents* f if it computes Φ (under some correspondence of $\mathcal{V} \cup \mathcal{W}$ to $\{x^1, \ldots, x^k\} \cup \Theta$).

For example, the programs in Fig. 1 resp. represent a 3-hidden-layer MLP and an RNN running for 3 steps. Note that the initial vectors correspond to a combination of input embeddings (e.g. $W^1\xi$) and vector parameters (e.g. biases) and the matrices correspond to matrix parameters (e.g. weights).

Intuition for a Program in the Large n Limit Typically, the vectors (resp. matrices) in a program will be sampled

[13]One can see this easily by modifying our calculations above.

iid like $\mathcal{N}(0,1)$ (resp. $\mathcal{N}(0,1/n)$), corresponding to the "standard" initialization of neural networks.[14] In such cases, when $n\to\infty$, a program behaves as follows, in a gist:

IID Coordinates Any vector $x\in\mathbb{R}^n$ in the program has roughly iid coordinates. We write Z^x for the random variable encoding this coordinate distribution. This Z^x may be correlated with Z^y for other vector y in the program, such that, for example, $\lim_{n\to\infty} x^\top y/n = \mathbb{E}\, Z^x Z^y$.

Nonlin $Z^{\psi(x^1,\dots,x^k)} = \psi(Z^{x^1},\dots,Z^{x^k})$.

MatMul, without $(W, W^\top)$-Interaction Consider a matrix $W\in\mathbb{R}^{n\times n}$ in the program and any set of $\mathbb{R}^n$ vectors $\mathbf{X}$ not dependent on vectors of the form $W^\top\bullet$. Then the set of random variables $\{Z^{Wx} : x\in\mathbf{X}\}$ are jointly Gaussian with mean zero and covariance $\mathrm{Cov}(Z^{Wx}, Z^{W\tilde{x}}) = \mathbb{E}\, Z^x Z^{\tilde{x}}$ for any $x,\tilde{x}\in\mathbf{X}$. If $\bar{W}\neq W$ is another matrix in the program and $\mathbf{Y}$ is a set of such vectors w.r.t. $\bar{W}$, then the set $\{Z^{Wx} : x\in\mathbf{X}\}$ is independent from $\{Z^{\tilde{W}y} : y\in\mathbf{Y}\}$.

MatMul, with $(W, W^\top)$-Interaction For general x, Z^{Wx} decomposes into a sum of a Gaussian part, identical to Z^{Wx} in the above case, and a correction term. This decomposition is a generalization of Eq. (17).

See Theorem B.4 for formal details.

NETSOR$\top^+$ Programs (Yang, 2019a; 2020a) showed that NETSOR$\top$ suffices to express the forward and backward passes of most architectures such as ResNet (with Batchnorm), but Transformer and other standard architectures require adding to NETSOR$\top$ a new "averaging" instruction[15] that returns the "empirical average" $\frac{1}{n}\sum_{\alpha=1}^n x_\alpha$ of a vector $x\in\mathbb{R}^n$. In the $n\to\infty$ limit, this scalar converges to $\mathbb{E}\, Z^x$ as would be expected from the intuitions above. This extension of NETSOR$\top$ (called NETSOR$\top^+$) also allows us to express the network output and loss derivative (e.g. in contrast to Fig. 1), which will be a technical requirement for unrolling the entirety of SGD training inside a single program, a key step in the proof of our main result. See discussion of proof formalization in Section 5. We can say a NETSOR$\top^+$ program *represents a network f* if it computes the body of f.

5. Universality of Kernel Dynamics

(Yang, 2019a; 2020a) showed that any neural network of standard architecture is represented by a NETSOR$\top^+$ program. Moreover,

[14]In the original definition of (Yang, 2019a; 2020a;b), the vectors can have correlations between them, but we can always rewrite these vectors as linear image of another set of uncorrelated vectors.

[15]For readers familiar with Tensor Programs, this is equivalent in expressivity to MOMENT, which is a composition of a NONLIN and this averaging instruction.

Theorem 5.1 (Yang (2020a)). *For a neural network as in Setup 5.2 below, its Neural Tangent Kernel at initialization has a well-defined infinite-width limit $\mathring{\mathcal{K}}$.*

Setup 5.2 (Representable NN in NTK Parametrization). *Suppose a neural network f is represented by a* NETSOR$\top^+$ *program (in the sense of Definition 4.2) whose* NONLIN *all have polynomially bounded derivatives.*[16] *Adopt the NTK parametrization: for every matrix parameter $W\in\mathbb{R}^{n\times n}$ of f, we factor $W=\frac{1}{\sqrt{n}}w$ where w is the trainable parameter; likewise, for each input layer matrix $U^i\in\mathbb{R}^{n\times d}$, we factor $U^i=\frac{1}{\sqrt{d}}u^i$, and likewise the output matrix $V=\frac{1}{\sqrt{n}}v$, such that u^i, v are trainable. Finally, we randomly initialize all trainable parameters iid as $\mathcal{N}(0,1)$.*

Our main result is to show that the SGD training of such a neural network described in Setup 5.2 reduces to kernel gradient descent with kernel $\mathring{\mathcal{K}}$ in the infinite-width limit.

Theorem 5.3 (NTKTRAIN is Architecturally Universal). *Consider training a network f described in Setup 5.2 via SGD with batch-size 1 and (WLOG) learning rate 1. Let ξ_t be the input and $\mathcal{L}_t:\mathbb{R}\to\mathbb{R}$ be the loss function (absorbing the label) at time t. Suppose $\mathcal{L}_t$ is continuous for all t. Then, for any ξ and t, $f_t(\xi)$ converges almost surely to a random variable $\mathring{f}_t(\xi)$ as width $\to\infty$, such that*

$$\mathring{f}_{t+1}(\xi) - \mathring{f}_t(\xi) = -\mathring{\mathcal{K}}(\xi,\xi_t)\mathcal{L}'_t(\mathring{f}_t(\xi_t)) \tag{22}$$

where $\mathring{\mathcal{K}}$ is the infinite-width NTK (at initialization) of the neural network.

The full proof of Theorem 5.3 is given Appendix D.

Extension We briefly mention several ways our result can be easily extended. *0) Different batch sizes, learning rate schedules, and nonscalar outputs. 1) Variants of NTK parametrization.* We can deal with any parametrization that scales the same way as NTK parametrization, e.g. weights are sampled like $N(0,\sigma^2)$ for any σ, with the multipliers $\gamma/fanin$ for any γ. *2) Variable width.* In real networks, the width of different layers can often be different (e.g. in ResNet). Our result can be extended to the case where the widths tend to infinity at a fixed ratio, using the variable-width version of Tensor Programs (Yang, 2020b). *3) Unsupervised and other learning settings* can be covered because their training and testing computation can be written into Tensor Programs. *4) Weight decay, momentum, and other optimizer tricks* can be covered as well as they can be straightforwardly written into Tensor Programs, but in general the kernel will change from step to step in contrast to Theorem 5.3.

[16]More generally, we can allow any pseudo-Lipschitz function here, but for simplicity we go with the statement in the main text.

5.1. Proof Sketch of Special Case

To convey the main idea, we give a proof sketch of a simplified problem: we assume 1) the input, output layers and biases are not trained (the network has only $\mathbb{R}^{n\times n}$ matrices as trainable parameters); 2) the forward pass does not contain both a weight matrix W and its transpose $W^\top$ (but a single matrix W can still be used multiple times without being transposed); 3) input space is $\mathbb{R}^d$ (with $k=1$), and $f = V^\top x \in \mathbb{R}$; 4) the output vector x is a G-var; 5) the network is represented by a NETSOR⊤ (instead of NETSOR⊤$^+$) program; [17] In the appendix, we prove the general case with these simplifications lifted.

It turns out, every NETSOR⊤ can be simplified into a *standard form* of sorts, which greatly facilitates our proof.

Definition 5.4. In a NETSOR⊤ program, a *G-var*[18] is an initial vector or a vector created by MATMUL, while an *X-var* is a vector created by NONLIN.[19] We define a *reduced* NETSOR⊤ *program* as a program in which only G-vars are allowed as inputs to a NONLIN, while only an X-var is allowed as input to a MATMUL.

Observe that any NETSOR⊤ program may be trivially expressed as a reduced NETSOR⊤ program by: 1) collapsing chains of non-linearities which appear consecutively, and 2) insert a NONLIN operation with $\psi(x) = x$ in between consecutive G-vars. Hence, we may safely assume that f is representable by a reduced NETSOR⊤ program.

Key Idea: Paths The examples of Sections 3.1 and 3.2 exposed several insights, such as the iid-coordinates intuition, important for proving Theorem 5.3. Now we discuss the one remaining key idea for scaling up to general architectures:

Definition 5.5 (Paths)**.** In a NETSOR⊤ program, a path p starts with an X-var and ends with a G-var, alternating between X- and G-vars along the path. We write p^0 for the starting X-var, p^1 for the following G-var, and so on, as well as p^{-1} for the ending G-var (see Fig. 2 for a graphical illustration). For odd i, let W^{p^i} denote the defining matrix of G-var p^i. For two equal length paths p, q, we write $p \cong q$ (path p is isomorphic to path q) if for all odd i, W^{p^i} is the same matrix as W^{q^i}.[20] In other words, we say path p is isomorphic to path q if their sequences of MATMUL matrices are identical, (but the NONLIN don't have to be, see Fig. 3 for a graphical illustration). Let $|p|$ denote the number of vectors in p (this is always an even number).

The collection of paths p starting with an X-var $p^0 = x$ and

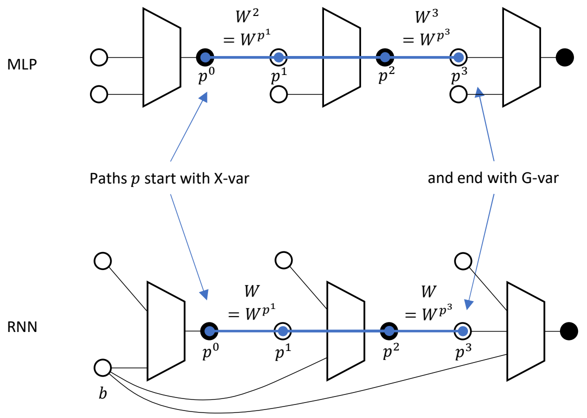

Figure 2: **Paths.** A graphical illustration of paths in an MLP and RNN. A path p always starts with an X-var and ends with a G-var. In these examples, path length $|p| = 4$.

ending with a G-var h describes all possible pathways of backpropagating an error signal dh at h to an error signal dx at x. Simultaneously, it also describes all possible pathways of forward propagating a change in x to a change in h.

Decomposing the Change in f to a Sum over Paths Because the gradient $\nabla_W f$ of a weight $W \in \mathbb{R}^{n\times n}$ is the sum of outer products $\sum_{h,x:h=Wx} dh \otimes x$, summing over all G-vars h and X-vars x in the program with $h = Wx$ (where dh denotes $\nabla_h f$), we also have

$$\nabla_W f = \sum_{h,x:h=Wx} \sum_p J^p \otimes x. \tag{23}$$

where

$$(J^p)^\top = \frac{\partial f}{\partial p^{-1}} \times \frac{\partial p^{-1}}{\partial p^{-2}} \times \frac{\partial p^{-2}}{\partial p^{-3}} \times \ldots \times \frac{\partial p^2}{\partial h} \tag{24}$$

i.e, J^p denotes the error signal at h from backpropagation through path p, and p ranges over all paths starting with $p^0 = x, p^1 = h$ and ending with the output node of the underlying program. Recall W factors as $\frac{1}{\sqrt{n}}w$ where w is the trainable parameter, not W. By the discussion above, updating w with $\nabla_w f = \frac{1}{\sqrt{n}}\nabla_W f$ causes f to change by

$$\begin{aligned}&\left\langle \frac{1}{\sqrt{n}}\nabla_W f, \frac{1}{\sqrt{n}}\nabla_W f\right\rangle \\ &= \frac{1}{n}\sum_{h,x:h=Wx}\sum_{\bar h,\bar x:\bar h=W\bar x}\sum_p\sum_{\bar p}\langle J^p, J^{\bar p}\rangle\langle x,\bar x\rangle.\end{aligned} \tag{25}$$

When every parameter w is randomly initialized iid as $\mathcal{N}(0,1)$, it turns out that $\langle J^p, J_{\bar p}\rangle$ will go to 0 as $n\to\infty$ unless $p \cong \bar p$ (Definition 5.5). If one think of J^p as a product of random Gaussian matrices (interleaved with other matrices), then this is akin to the fact that, for a mixed moment $M \overset{\text{def}}{=} \mathbb{E}\prod_{i=1}^r Z_{\gamma(i)}, \gamma : [r] \to [k]$ of standard iid

[17]This is not common in deep learning, but appears in some weight-tied autoencoders (Li & Nguyen, 2019).

[18]"G" because G-vars often are roughly Gaussian vectors

[19]*Var* is short for *variable*, as the vectors are considered *variables* in the program. In previous works, *H-var* refers to any vector in the program; we will not use this terminology.

[20]Here we are talking about equality of symbols rather than equality of values of those symbols.

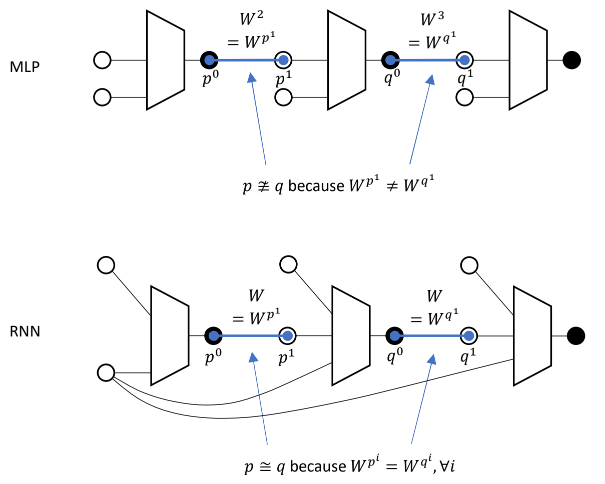

Figure 3: **Paths Isomorphism.** A graphical illustration of path isomorphism for paths p, q with $|p| = |q| = 2$ in an MLP and RNN. In the MLP example, p is not isomorphic to q because the weights W^1, W^2 defining the vectors p^1, q^1 are not identical. For the RNN, p is isomorphic to path q because p^1 and q^1 are defined with the same weight W.

Gaussians $Z_1, \ldots, Z_k$, M is nonzero iff every Z_i appears an even number of times in the product. This means we can replace the 4 nested sums in Eq. (25) with the single sum $\sum_{p \cong \bar{p}}$ and rewrite $x = p^0, \bar{x} = \bar{p}^0$.

We have suppressed dependence on input in Eq. (25). Being more explicit about it and performing updates on all weights, we have

$$f_{t+1}(\xi) - f_t(\xi)$$
$$\approx -\mathcal{L}'_t(f_t(\xi_t)) \sum_{p \cong \bar{p}} \langle J^p(\xi), J^{\bar{p}}(\xi_t) \rangle \langle p^0(\xi), \bar{p}^0(\xi_t) \rangle. \tag{26}$$

after taking a gradient step on input ξ_t at initialization $t = 0$. (Here $p^0(\xi)$ denotes the vector p^0 as a function of ξ at initialization). However, Eq. (25) holds for general t as well: The key insight is similar to Eq. (4) in the 1-hidden-layer example, that vectors $p^0(\xi)$, $J^p(\xi)$, etc change vanishingly from their initial values as $n \to \infty$, after any number of SGD steps. Because our arguments above only depend on inner products between vectors, this means that the error in Eq. (26) for $t > 0$ vanishes as $n \to \infty$. Finally, at least heuristically, it is straightforward to show the RHS of Eq. (26) is the NTK via a series of calculations exemplified by those in Sections 3.1 and 3.2, to get Eq. (22).

Putting It All Together and Proof Formalization While the core ideas discussed above are intuitive, making them rigorous at face value would be quite challenging. Instead we use the machinery offered by the Tensor Programs framework. The mechanics of the proof then goes as follows: 1) First we unroll SGD of f into a NETSOR$\top^+$ program.[21] This is similar to the equations in Sections 3.1 and 3.2; the key here is to express $\delta x_{t+1} = \sqrt{n}(x_{t+1} - x_t)$ as a vector in the program, for any (pre-)activation x. 2) We apply the NETSOR$\top^+$ *Master Theorem* (Yang, 2020b) to this program. This yields the coordinate distribution of each vector. The core insights here are demonstrated by the calculation with the Z random variables in Sections 3.1 and 3.2. 3) Finally, we need to show that (the rigorous version of) Eq. (26) indeed recovers the NTK and agrees with Eq. (22). This is done via an inductive (symbolic) computation, and the path concept in this section plays a key role here.

6. Related Works

The connection between kernel methods and neural networks has had a long history before its recent resurgence. The Gaussian Process (NNGP) view of wide neural networks, which characterizes the behaviour of training only the last layer of a wide neural network, has been studied in (Daniely et al., 2016; Hazan & Jaakkola, 2015; Roux & Bengio, 2007; Lee et al., 2018; Matthews et al., 2018; Hinton & Neal, 1995; Novak et al., 2019). Since the original NTK paper (Jacot et al., 2018), many works have informally derived the infinite-width NTK for various architectures such as CNNs (Arora et al., 2019), RNN (Alemohammad et al., 2020), attention (Hron et al., 2020), ensembles (Littwin et al., 2020b) and graph neural networks (Du et al., 2019), but none of them formally proved NTKINIT or NTKTRAIN for those architectures. Finite width corrections to the NTK were derived for fully connected networks in (Hanin & Nica, 2019; Littwin et al., 2020a). The validity of the NTK theory was empirically studied in (Lee et al., 2019) for a variety of architectures.

The Tensor Program framework (Yang, 2019a; 2020a;b) was introduced in an attempt to unify and generalize the NNGP/NTK theory to a broad range of architectures, eliminating the need to re-develop the theory for each new architecture. For example, (Yang, 2019a) proved the architectural universality of NNGP correspondence, while (Yang, 2020a) proved that of NTKINIT. On the other hand, (Yang, 2020b) developed the most general machinery for Tensor Programs and as a corollary constructed a comprehensive theory of nonlinear random matrix theory, that, for example, can calculate the singular value distribution of a wide neural network of any architecture. Our proofs depend on the machinery of (Yang, 2020b) crucially, as discussed in Section 5.

[21] We note that this formalization crucially relies on NETSOR$\top^+$ and its Master Theorem from (Yang, 2020b) because the SGD unrolling cannot be done in NETSOR$\top$. The reason is that we need to express the output and loss derivatives of the network, which are scalars (or at least finite dimensional), and that cannot be done in a NETSOR$\top$ program. Furthermore, the Master Theorem from (Yang, 2020a) only pertains to a specific type of programs that look like the *first* backpropagation after initialization. Thus, it cannot deal with the complete unrolling of SGD as we do here, which requires the more advanced Master Theorem from (Yang, 2020b).

7. Conclusion

New theories of deep learning almost always start with MLPs, rightly so as they are the simplest case and can often reveal the key insights more clearly. Of course, as deep learning itself is an applied field, one should always ask whether insights on MLPs extend to more general architectures, i.e. whether there is an *architecturally universal extension* of a proposed theory of MLPs. This is not always easy to answer.

In this paper, we showed that the NTK theory is architecturally universal, but more importantly, we showed that the Tensor Programs technique is a very powerful tool for answering the above question as a matter of routine. Looking forward, we hope to apply it to generate more novel and general insights.

References

Alemohammad, S., Wang, Z., Balestriero, R., and Baraniuk, R. The recurrent neural tangent kernel. *ArXiv*, abs/2006.10246, 2020.

Allen-Zhu, Z., Li, Y., and Song, Z. A Convergence Theory for Deep Learning via Over-Parameterization. *arXiv:1811.03962 [cs, math, stat]*, November 2018. URL http://arxiv.org/abs/1811.03962.

Arora, S., Du, S., Hu, W., Li, Z., Salakhutdinov, R., and Wang, R. On exact computation with an infinitely wide neural net. In *NeurIPS*, 2019.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203, 2014.

Cho, K., Merrienboer, B. V., Çaglar Gülçehre, Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *ArXiv*, abs/1406.1078, 2014.

Daniely, A., Frostig, R., and Singer, Y. Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity. In *NIPS*, 2016.

Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, 2016.

Du, S., Hou, K., Póczos, B., Salakhutdinov, R., Wang, R., and Xu, K. Graph neural tangent kernel: Fusing graph neural networks with graph kernels. In *NeurIPS*, 2019.

Du, S. S., Zhai, X., Poczos, B., and Singh, A. Gradient Descent Provably Optimizes Over-parameterized Neural Networks. *arXiv:1810.02054 [cs, math, stat]*, October 2018. URL http://arxiv.org/abs/1810.02054.

Duvenaud, D., Maclaurin, D., Aguilera-Iparraguirre, J., Gómez-Bombarelli, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, 2015.

Fukushima, K. Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20(3):121–136, 1975. doi: 10.1007/BF00342633. URL https://doi.org/10.1007/BF00342633.

Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980. doi: 10.1007/BF00344251. URL https://doi.org/10.1007/BF00344251.

Hanin, B. and Nica, M. Finite depth and width corrections to the neural tangent kernel. *ArXiv*, abs/1909.05989, 2019.

Hazan, T. and Jaakkola, T. Steps toward deep kernel methods from infinite neural networks. *ArXiv*, abs/1508.05133, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Henaff, M., Bruna, J., and LeCun, Y. Deep convolutional networks on graph-structured data. *ArXiv*, abs/1506.05163, 2015.

Hinton, G. E. and Neal, R. Bayesian learning for neural networks. 1995.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

Hron, J., Bahri, Y., Sohl-Dickstein, J., and Novak, R. Infinite attention: Nngp and ntk for deep attention networks. In *ICML*, 2020.

Huang, G., Liu, Z., and Weinberger, K. Q. Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, 2017.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ArXiv*, abs/1502.03167, 2015.

Jacot, A., Gabriel, F., and Hongler, C. Neural tangent kernel: Convergence and generalization in neural networks. In *NeurIPS*, 2018.

Kipf, T. and Welling, M. Semi-supervised classification with graph convolutional networks. *ArXiv*, abs/1609.02907, 2017.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278–2324, 12 1998. doi: 10.1109/5.726791.

Lecun, Y., Haffner, P., and Bengio, Y. Object recognition with gradient-based learning. 08 2000.

Lee, J., Bahri, Y., Novak, R., Schoenholz, S., Pennington, J., and Sohl-Dickstein, J. Deep neural networks as gaussian processes. *ArXiv*, abs/1711.00165, 2018.

Lee, J., Xiao, L., Schoenholz, S. S., Bahri, Y., Sohl-Dickstein, J., and Pennington, J. Wide neural networks of any depth evolve as linear models under gradient descent. *ArXiv*, abs/1902.06720, 2019.

Li, P. and Nguyen, P.-M. On random deep weight-tied autoencoders: Exact asymptotic analysis, phase transitions, and implications to training. In *ICLR*, 2019.

Littwin, E., Galanti, T., and Wolf, L. On random kernels of residual architectures. *arXiv: Learning*, 2020a.

Littwin, E., Myara, B., Sabah, S., Susskind, J., Zhai, S., and Golan, O. Collegial ensembles. *ArXiv*, abs/2006.07678, 2020b.

Matthews, A., Rowland, M., Hron, J., Turner, R., and Ghahramani, Z. Gaussian process behaviour in wide deep neural networks. *ArXiv*, abs/1804.11271, 2018.

Novak, R., Xiao, L., Bahri, Y., Lee, J., Yang, G., Hron, J., Abolafia, D., Pennington, J., and Sohl-Dickstein, J. Bayesian deep convolutional networks with many channels are gaussian processes. In *ICLR*, 2019.

Roux, N. L. and Bengio, Y. Continuous neural networks. In Meila, M. and Shen, X. (eds.), *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, volume 2 of *Proceedings of Machine Learning Research*, pp. 404–411, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR. URL http://proceedings.mlr.press/v2/leroux07a.html.

Rumelhart, D., Hinton, G. E., and Williams, R. J. Learning internal representations by error propagation. 1986.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *ArXiv*, abs/1706.03762, 2017.

Yang, G. Tensor programs i: Wide feedforward or recurrent neural networks of any architecture are gaussian processes. *ArXiv*, abs/1910.12478, 2019a.

Yang, G. Scaling Limits of Wide Neural Networks with Weight Sharing: Gaussian Process Behavior, Gradient Independence, and Neural Tangent Kernel Derivation. *arXiv:1902.04760 [cond-mat, physics:math-ph, stat]*, February 2019b.

Yang, G. Tensor programs ii: Neural tangent kernel for any architecture. *ArXiv*, abs/2006.14548, 2020a.

Yang, G. Tensor programs iii: Neural matrix laws. *ArXiv*, abs/2009.10685, 2020b.

Yang, G. and Hu, E. J. Feature learning in infinite-width neural networks. *ArXiv*, abs/2011.14522, 2020.

Yang, G. and Schoenholz, S. S. Mean Field Residual Network: On the Edge of Chaos. In *Advances in neural information processing systems*, 2017.

Zou, D., Cao, Y., Zhou, D., and Gu, Q. Stochastic Gradient Descent Optimizes Over-parameterized Deep ReLU Networks. *arXiv:1811.08888 [cs, math, stat]*, November 2018. URL http://arxiv.org/abs/1811.08888.