

# CONVOLUTIONAL NEURAL TANGENT KERNEL

—

WILLIAM CAPPELLETTI

## Abstract

In this paper, we study the relation between artificial neural networks and kernel regression. We start by defining some neural networks, such as the multilayer perceptron (MLP) and the convolutional neural network (CNN); then we discuss some theoretical results that support the equivalence with kernel methods. We use *tensor programs* to see that at initialization infinitely wide neural networks are equivalent to Gaussian processes. Then, we study the *Neural Tangent Kernel*, which gives the direction along which the network evolves during training. In the infinite width limit of MLPs, this kernel converges to a finite one and is constant during training. Furthermore, if the MLP is big enough, with high probability, the function given by the network after training is close to the function given by the kernel regressor using the limiting Neural Tangent Kernel.

At this point, we test the kernel regressor against the actual neural network. We do so with a multilayer perceptron and a convolutional neural network, which we train and test on the MNIST handwritten digits dataset, on a classification task.

Semester Project  
under the supervision of  
Professor C. Hongler and Dr. F. Gabriel  
SFL Chair

Autumn 2019

The logo of EPFL (École Polytechnique Fédérale de Lausanne) is displayed in a large, bold, red, sans-serif font. The letters are stylized, with the 'E' and 'F' having a distinctive blocky appearance.

Mathematics Section  
MSc in Applied Mathematics 2019–2020

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Artificial Neural Networks</b>	<b>1</b>
2.1	Multilayer perceptron . . . . .	2
2.2	Convolutional Neural Network . . . . .	2
<b>3</b>	<b>Neural Networks as Gaussian Processes</b>	<b>3</b>
3.1	Tensor Programs . . . . .	3
3.2	The Infinite Width Limit . . . . .	5
3.3	Neural Tangent Kernel . . . . .	7
3.4	Kernel Regression . . . . .	8
3.5	Multidimensional output . . . . .	9
<b>4</b>	<b>NTK implementation</b>	<b>9</b>
4.1	NTK for finite networks . . . . .	10
4.2	NTK at infinite width . . . . .	10
<b>5</b>	<b>Experimental results</b>	<b>11</b>
5.1	The dataset . . . . .	11
5.2	Multilayer Perceptron . . . . .	12
5.3	Convolutional Neural Network . . . . .	12

# 1 Introduction

In this paper we study the relation between artificial neural networks and kernel regression. We start by defining some neural networks, such as the multilayer perceptron and the convolutional neural network; then we discuss some theoretical results that support the relation with kernel methods.

It has been proved, in [7], [4], and [9], that at initialization, if the parameters are sampled independently at random with proper distribution, infinitely wide neural networks are equivalent to Gaussian processes. To understand what an infinitely wide network is and to justify this result, we report Yang definition of *tensor programs*, [9], in Section 3.1.

Then, in Section 3.2, we explain what it means for a network to be infinitely wide. Once this concept is clear, we give the conditions under which the infinite network behaves like a Gaussian process, which are quite general conditions satisfied by most standard deep learning architectures.

We then follow [4] and [1] and in Section 3.3 we study the Neural Tangent Kernel, which gives the direction along which the network evolves during training. In [4], Jacot et al. proved that in the infinite width limit this kernel converges to a finite one and is constant during training. Then, in [1], Arora et al. prove that if a network is big enough, with high probability, the function given by the network after training is close to the function given by the kernel regressor using the limiting Neural Tangent Kernel.

At this point, we focus on the experimental part. Firstly, in Section 4, we precise how to compute, practically, the Neural Tangent Kernel, with the technical limitations deriving from its definition. Then, in Section 5, we test the kernel regressor against the actual neural network. We do so with a multilayer perceptron and a convolutional neural network, which we train and test on the MNIST handwritten digits dataset, on a classification task. The task consisting in recognising the digit drawn on a  $28 \times 28$  grayscale image.

We see that the kernel methods accuracy is worse than the actual neural network, but it is consistently close to it.

## 2 Artificial Neural Networks

Artificial Neural Networks (ANNs) describe a wide family of parametrized functions, which has gained more and more popularity thanks to their outstanding performances in various regression and classification tasks, as well as in computer vision and natural language processing, [2]. Many different structures exist and have different properties, but the main idea is that we model a function through many layers, composed by different nodes. Each node is a function of other nodes appearing in previous layers; more precisely, we take a linear combination of said nodes and then apply a nonlinear function to the output.

The models are trained starting from a loss function, which gives a measure of the error with respect to the task the ANN should perform. We use this loss function to perform gradient descent, or variants of it, over the parameters of the network, using a technique known as backpropagation.

To get a better understanding of this family of functions, we give two examples, which are widely used in practice, namely the multilayer perceptron and the convolutional neural network. We will focus on these two architectures for the experimental part.

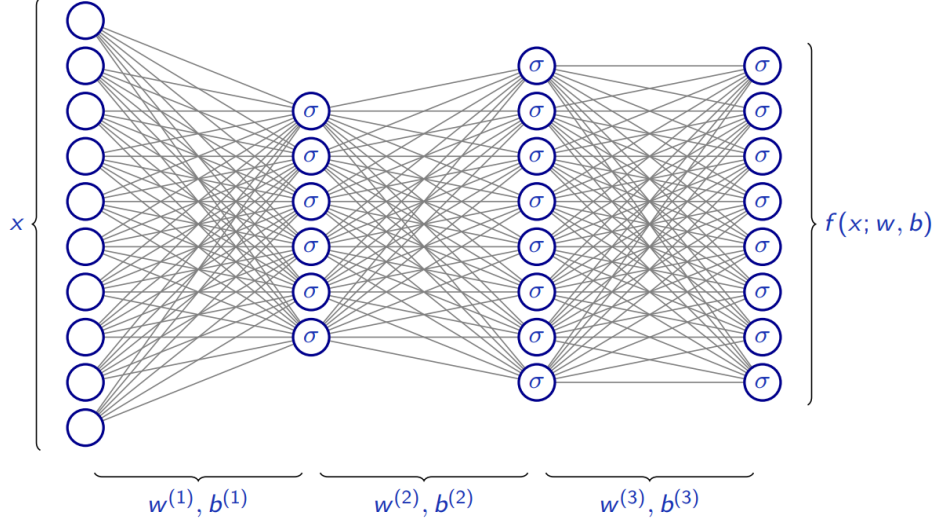


Figure 1: Illustration of a Multilayer perceptron with three hidden layers, from [2]. The last one, on the furthest right, is the readout layer  $f(x; w, b)$ , which is a function of both the input and the parameters.

## 2.1 Multilayer perceptron

The multilayer perceptron (MLP), also known as fully connected neural network, is the simplest example of ANN. It consists of an input layer and an arbitrary number, say  $L$ , of hidden layers, the last of which is also called the readout layer. The input layer is just the datapoint  $x = x^{(0)} \in \mathbb{R}^{n^0} = \mathbb{R}^{n_{\text{in}}}$  that we feed into the network, where each coordinate is considered as a node. Each hidden layer  $l$  consists of a weight matrix  $w^{(l)} \in \mathbb{R}^{n^{l-1} \times n^l}$ , a bias vector  $b^{(l)} \in \mathbb{R}^{n^l}$  and a nonlinearity  $\sigma$  which is applied component-wise. The nodes in the hidden layer  $l$  corresponds to the values of the vector  $x^{(l)}$  defined recursively as

$$x^{(l)} = \sigma(w^{(l)} \cdot x^{(l-1)} + b^{(l)}), \quad \forall l = 1, \dots, L. \quad (2.1)$$

The readout layer describes a function of both the input and the parameters, namely  $f(x; w^{(1)}, b^{(1)}, \dots, w^{(L)}, b^{(L)})$ . Figure 1 illustrates an MLP with three hidden layers.

## 2.2 Convolutional Neural Network

Convolutional neural networks (CNNs) are generally used with image inputs, i.e. two or three-dimensional tensor. They work by concatenating different convolutional layers, which are, basically, filters on images.

We can imagine a grayscale picture as a matrix encoding each pixel luminosity, which is a two-dimensional tensor, having a *width* and a *height*; on the other hand, colour images are encoded as three matrices, each one encoding the luminosity of a different primary colour (for instance Red, Green and Blue). In the latter case, we can stack the three matrices in one three dimensional tensor, the third dimension being referred to as *channel*.

What a convolutional layer does, in practice, is applying a filter, across multiple channels to parts of the images in a convolutional way, hence the name. To give a precise definition, we let

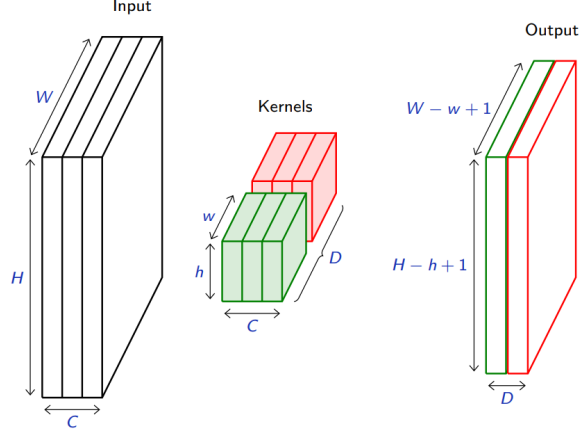


Figure 2: Illustration of a convolutional layer, from [2]. Kernel is used here as synonym for filter.  $W$ ,  $H$  and  $C$  are width, height and number of channels of the input image;  $w$ ,  $h$  are width and height of the filters,  $D$  is the number of channels of the output image.

$P$  be the width and  $Q$  be the height of the image, in practice  $P = Q$ . We use  $q \times q' \in \mathbb{N} \times \mathbb{N}$  to denote the filter size. For a convolutional filter  $w \in \mathbb{R}^{q \times q'}$  and an image  $x \in \mathbb{R}^{P \times Q}$ , the convolution operator is defined as

$$[w * x]_{ij} = \sum_{a=1}^q \sum_{b=1}^{q'} [w]_{a,b} [x]_{a+i, b+j}, \quad \text{for } i \in [P - q + 1], j \in [Q - q' + 1]. \quad (2.2)$$

The CNN is then defined recursively as follows

- Let  $x^{(0)} = x \in \mathbb{R}^{P \times Q \times C^{(0)}}$  be the input image, where  $C^{(0)}$  is the initial number of channels.
- For  $h = 1, \dots, L$ ,  $\beta = 1, \dots, C^{(h)}$ , the intermediate outputs are defined as

$$x_{(\beta)}^{(h)} = \sigma \left( \sum_{\alpha=1}^{C^{(h-1)}} W_{(\alpha),(\beta)}^{(h)} * x_{(\alpha)}^{(h-1)} \right), \quad (2.3)$$

where each  $W_{(\alpha),(\beta)}^{(h)}$  is a filter with appropriate size and  $\sigma$  is a nonlinear function applied component-wise.

As we can see from (2.3), the number of channels changes from one layer to the other (from  $C^{(h-1)}$  to  $C^{(h)}$ ) and from (2.2) the image size shrinks according to the filter size (as the filter cannot move outside the boundaries of previous layer). Figure 2 illustration of a convolutional layer.

### 3 Neural Networks as Gaussian Processes

#### 3.1 Tensor Programs

Our goal is to study how Artificial Neural Networks (ANN) behaves when their sizes become arbitrarily big and their parameters are initialized at random, following Glorot initialization [3].

To study such functions, Yang introduces the concept of *tensor program* in [9] and [10]. This framework is used to bind the sizes of parameters across different parts of the network, by explicitly binding to a line of the program introduction and usage of each parameter and non-linearity.

**Definition 3.1.** We consider programs of the following form, which we call *tensor programs*. Each line  $l$  contains an assignment and a dimension annotation and can have the following types.

- **VecIn** (G) a vector input  $x$

$$l : g^l := x \in \mathbb{R}^{n^l};$$

- **MatIn** (A) a matrix input  $A$

$$l : A^l := A \in \mathbb{R}^{n_1^l \times n_2^l};$$

- **T** (A) transpose of an A-var

$$l : A^l := (A^j)^t \in \mathbb{R}^{n_1^l \times n_2^l} = \mathbb{R}^{n_2^j \times n_1^j};$$

- **MatMul** (G) if  $A^k$  and  $g^j$  have  $n_2^k = n^j$ , then an assignment via a linear mapping

$$l : g^l := A^k g^j \in \mathbb{R}^{n^l} = \mathbb{R}^{n^k};$$

or similarly for H-vars

$$l : g^l := A^k h^j \in \mathbb{R}^{n^l} = \mathbb{R}^{n^k}$$

where  $j, k < l$ ;

- **LinComb** (G) if  $n^{j_1} = \dots = n^{j_k}$ , then an assignment via linear combination of G-vars that apperared in previous lines: with  $a_{j_i}^l \in \mathbb{R}$ ,

$$l : g^l := a_{j_1}^l g^{j_1} + \dots + a_{j_k}^l g^{j_k} \in \mathbb{R}^{n^{j_1}};$$

- **Nonlin** (H) if  $n^{j_1} = \dots = n^{j_k}$ , then an assignment via some general (possibly nonlinear) function  $\phi^l : \mathbb{R}^k \rightarrow \mathbb{R}$ , acting coordinatewise,

$$l : h^l := \phi^l(g^{j_1}, \dots, g^{j_k}) \in \mathbb{R}^{n^l} = \mathbb{R}^{n^{j_1}}.$$

The letters in parentheses indicate a naming convention for the variables introduced in each line and group them according to similar properties. We call *G-vars* the assignments of VecIn, MatMul and LinComb; *A-vars* those of MatIn and T; and *H-vars* the variables assigned by Nonlin.

In short, *tensor programs* allow the input of vectors (**VecIn**) and matrices (**MatIn**) and their usage in linear combination of vectors (**LinComb**), matrix multiplication (**MatMul**), matrix transpose (**T**) and application of general (possibly nonlinear) functions (**Nonlin**). Some examples of standard deep learning architectures, translated as tensor programs, can be found in [9, Section 3] and, in fact, this convention can express almost any neural network. In spite of being more cumbersome of the usual algebraic formulation, this notation makes it easier to study ANN as we let their sizes increase arbitrarily.

Intuitively, each program line  $l$  binds a variable assignment ( $g^l$ ,  $A^l$  or  $h^l$ ) and a dimension annotation ( $n^l$  or  $n_1^l \times n_2^l$ ), such that one can use each line result in other operations and keep track of the relations between dimensions. It follows that lines of type T, MatMul, LinComb, and Nonlin induce equality constraints on the dimensions of each line.

Given a program  $\pi$  and a possible set of additional dimensional constraints  $\Lambda$ , we can consider the smallest equivalence relation  $\sim$  on G-vars such that  $g \sim g'$  if their dimensions are constrained to be equal by  $\Lambda$  or by some line of the program. With this relation, we can split the G-vars into classes and we call each class a *common dimension class* (CDC); we write  $\mathfrak{C}$  the collection of all CDCs for a program  $\pi$  and additional constraints  $\Lambda$ . The CDCs are the main instrument to understand the ANN behaviour when we let the network's dimensions increase, as all its elements scale together.

### 3.2 The Infinite Width Limit

We study how ANNs behave when we let their dimensions go to infinity, in what is called the Infinite Width Limit. Before we go deeper into theoretical results, we want to clarify what it means, for different architectures, to become infinitely wide. The easiest case is the multilayer perceptron (MLP), which is a simple neural network in which all layers are fully connected. By definition, the only dimensions that are bound are the input size of a layer and the output size of the previous one. Therefore, each layer gives its own CDC and the dimensions that go to infinity are the number of nodes in each layer (i.e the columns in each weight matrix and bias vector). This justifies the approach used in [4], where the authors make layers sizes increase to infinity sequentially.

A different intuition arises when considering CNNs. In this case, by the definition of convolution, one cannot have the size of the filter to grow to infinity, as that would impose all layers to grow to infinity together and it would require an infinite size input, which is absurd. Therefore, in such a situation, the filter size stays the same, and the dimension that grows to infinity, in each layer, is the number of channels. To visualize what that means in terms of tensor programs, we can consider an image with different channels. We take a vector across channels for every single pixel and we can see each filter application as first multiplying each pixel-vector by a weight matrix, giving as result a pixel-vector with a different number of channels, and then obtaining the preactivations for each coordinate of the new tensor as linear combinations of the latter pixel-vectors. Finally, one can apply the nonlinearity pointwise. For a complete example, we refer the reader to [9, Appendix B.6].

In the same way, we can express through tensor programs more complex architectures, such as Residual Neural Networks and other kinds of transformer, as shown in [9, Appendix B]. After we understand what dimensions go to infinity, we can prove that some similar behaviour arises, in spite of the architectures being significantly different. The first consequence is that in the *Infinite Width Limit* all ANNs behave like *Gaussian Processes*.

Neal first proved, in [7], that a single-layer neural network with random parameters can converge in distribution to a Gaussian process as its width goes to infinity. In [9], Yang extends this result to any network that can be described by a tensor program.

We consider a sequence (in  $t \in \mathbb{N}$ ) of dimensions  $\{n^{lt}\}_{g^l \text{ or } h^l} \cup \{n_1^{lt}, n_2^{lt}\}_{A^l}$  respecting the equivalence relation  $\sim$  (defined in Section 3.1) in the program  $\pi$ , where  $g^l$ ,  $h^l$  and  $A^l$  are, respectively, G, H, and A-vars appearing at line  $l$  in  $\pi$ . At time  $t$ , we sample independently  $A_{ij}^{lt} \sim \mathcal{N}(0, (\sigma^{lt})^2/n_2^{lt})$ , for each  $i, j$ , for a set  $\{\sigma^{lt}\}_{A^l}$  (this is the well established Glorot initialization). For each common dimension class  $\mathfrak{c}$ , we also sample independently  $g^{c_{in}t} \sim \mathcal{N}(\mu^{c_{in}t}, K^{c_{in}t})$  for each  $i$ . Here  $\mathfrak{c}_{in}$  is the set

of input G-vars in  $\mathbf{c}$ ,  $g^{\mathbf{c}_{\text{in}} t} = (g_i^{lt})_{g^l \in \mathbf{c}_{\text{in}}}$ , and  $\mu^{\mathbf{c}_{\text{in}} t} : \mathbf{c}_{\text{in}} \rightarrow \mathbb{R}$ ,  $K^{\mathbf{c}_{\text{in}} t} : \mathbf{c}_{\text{in}} \times \mathbf{c}_{\text{in}} \rightarrow \mathbb{R}$  are specified mean and covariance at time  $t$ . Thus, given  $(\pi, \Lambda)$ , the data  $\{n^{\mathbf{c} t}\}_{\mathbf{c} \in \mathfrak{C}}$ ,  $\{\sigma^{lt}\}_{A^l}$ ,  $\{\mu^{\mathbf{c}_{\text{in}} t}\}_{\mathbf{c} \in \mathfrak{C}}$  and  $\{K^{\mathbf{c}_{\text{in}} t}\}_{\mathbf{c} \in \mathfrak{C}}$  realize a random program  $\pi(\{n^{\mathbf{c} t}\}_{\mathbf{c} \in \mathfrak{C}}, \{\sigma^{lt}\}_{A^l}, \{\mu^{\mathbf{c}_{\text{in}} t}\}_{\mathbf{c} \in \mathfrak{C}}, \{K^{\mathbf{c}_{\text{in}} t}\}_{\mathbf{c} \in \mathfrak{C}})$ .

Furthermore, we assume that as  $t \rightarrow \infty$ , for all  $\mathbf{c}, \mathbf{c}' \in \mathfrak{C}$ :

1.  $n^{\mathbf{c} t}$  is increasing with  $t$  and  $n^{\mathbf{c} t} \rightarrow \infty$ .
2.  $\lim_{t \rightarrow \infty} n^{\mathbf{c} t} / n^{\mathbf{c}' t} = \alpha_{\mathbf{c}, \mathbf{c}'} \in (0, \infty)$ , for some constant  $\alpha_{\mathbf{c}, \mathbf{c}'}$  depending only on  $\mathbf{c}' \in \mathfrak{C}$ .
3.  $\sigma^{lt} \rightarrow \sigma^{l\infty}$  for some finite  $\sigma^{l\infty} > 0$  for each input A-var  $A^l$ .
4.  $\mu^{\mathbf{c}_{\text{in}} t} \rightarrow \mu^{\mathbf{c}_{\text{in}} \infty}$  and  $K^{\mathbf{c}_{\text{in}} t} \rightarrow K^{\mathbf{c}_{\text{in}} \infty}$  for some finite  $\mu^{\mathbf{c}_{\text{in}} \infty}$  and  $K^{\mathbf{c}_{\text{in}} \infty}$ , and  $\text{rank } K^{\mathbf{c}_{\text{in}} t} = \text{rank } K^{\mathbf{c}_{\text{in}} \infty}$  for all large  $t$ .

Finally, for the main theorem proved by Yang in [9] to hold, we need to restrict to a general class of nonlinearities, which we can intuitively think as meaning that the function is at most exponential.

**Definition 3.2.** For  $\alpha > 0$ , a function  $\phi : \mathbb{R}^k \rightarrow \mathbb{R}$  is said to be *alpha-controlled* if for some  $C, c > 0$ , we have

$$|\phi(x)| \leq \exp \left( C \sum_{i=1}^k |x_i|^\alpha + c \right) \quad (3.1)$$

for all  $x \in \mathbb{R}^k$ .

**Theorem 3.3** (G. Yang 2019, [9] and [10]). *Consider dimension constraints  $\Lambda$  and a program  $\pi$  without  $T$  lines, i.e. no transpose allowed. Suppose the nonlinearities are  $\alpha$ -controlled for some  $\alpha < 2$ . Sample all input vars as explained beforehand (Glorot initialization). Then, for any  $\mathbf{c} \in \mathfrak{C}$  and any  $\alpha$ -controlled function  $\psi : \mathbb{R}^{\mathbf{c}} \rightarrow \mathbb{R}$ ,  $\alpha < 2$ ,*

$$\frac{1}{n^{\mathbf{c} t}} \sum_{i=1}^{n^{\mathbf{c} t}} \psi(g_i^{\mathbf{c} t}) \xrightarrow{a.s.} \mathbb{E} \psi(Z), \quad (3.2)$$

where  $g_i^{\mathbf{c} t} = (g_i^{lt})_{g^l \in \mathbf{c}}$  and  $\mathbb{R}^{\mathbf{c}} \ni Z = (Z^g)_{g \in \mathbf{c}} \sim \mathcal{N}(\mu^{\mathbf{c}}, K^{\mathbf{c}})$ , with  $\mu^{\mathbf{c}}$  and  $K^{\mathbf{c}}$  defined respectively in (3.3) and (3.4).

For any  $\mathbf{c} \in \mathfrak{C}$ , we recursively define

$$(\text{mean}) \quad \mu^{\mathbf{c}}(g^l) = \begin{cases} \mu^{\mathbf{c}_{\text{in}} \infty}(g^l) & \text{if } g^l \in \mathbf{c}_{\text{in}} \\ \sum_i a_{ji}^l \mu^{\mathbf{c}}(g_i^j) & \text{if } g^l := \sum_i a_{ji}^l g_i^j \\ 0 & \text{if } g^l := A^k g^j \text{ or } g^l := A^k h^j \end{cases} \quad (3.3)$$

and recursively define

$$(\text{covariance}) \quad K^{\mathbf{c}}(g^l, g^m) = \begin{cases} K^{\mathbf{c}_{\text{in}} \infty}(g^l, g^m) & \text{if } g^l, g^m \in \mathbf{c}_{\text{in}} \\ \sum_i a_{ji}^m K^{\mathbf{c}}(g^l, g_i^j) & \text{if } g^m := \sum_i a_{ji}^m g_i^j \\ \sum_i a_{ji}^l K^{\mathbf{c}}(g_i^j, g^m) & \text{if } g^l := \sum_i a_{ji}^l g_i^j \\ (\sigma^{k\infty})^2 \mathbb{E}_z[\phi^a(z) \phi^b(z)] & \text{if } g^l := A^k h^a, g^m := A^k h^b \\ 0 & \text{else} \end{cases} \quad (3.4)$$



where  $h^a := \phi^a(g_1^j, \dots, g_k^j)$ ,  $h^b := \phi^b(g_1^{j'}, \dots, g_k^{j'})$  and  $z \sim \mathcal{N}(\mu^c, K^c)$ . Note that (3.3) and (3.4) keep track of how the mean and covariance change while the variables pass through the network. The different branches characterize which kind of tensor program line assigned the variables  $g^l$  and  $g^m$ .

We now make a more technical comment about the recursion. While other branches are more clear, note that branch 4 in (3.4) also covers the case when  $g^l := A^k g^a$  or  $g^m := A^k g^b$  by “type-casting”  $g^a$  to an H-var and setting  $\phi^a = \text{id}$  (similarly for  $g^b$ ). Finally, remark that  $\phi^a$  will ignore irrelevant components of  $a$ , and the expectations only depend on the entries of  $\mu^c$  and  $K^c$  that correspond to already-defined values, so this describes a valid recursion.

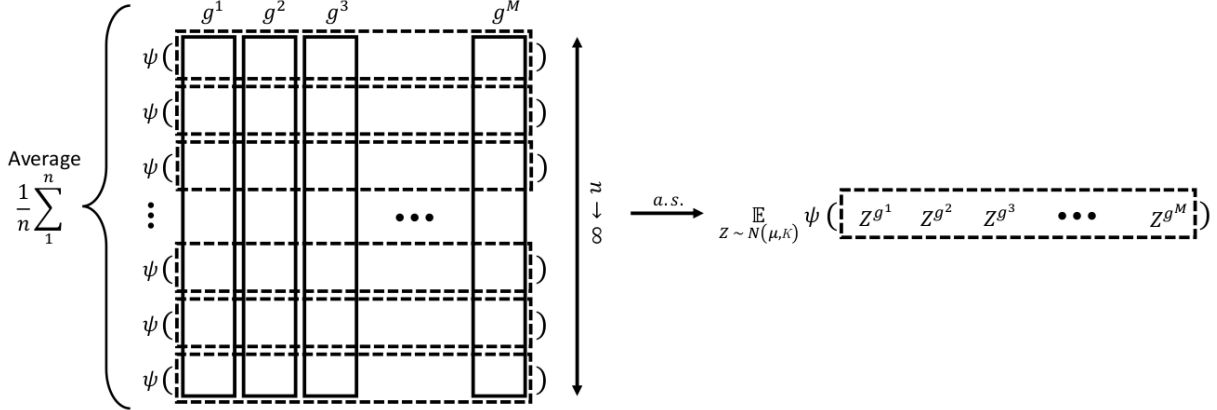


Figure 3: Illustration of Theorem 3.3, from [10]. If we suppose that  $g^1, \dots, g^M$  are all the G-vars in a CDC (i.e.  $g^c = (g^1, \dots, g^M)$ ), then, as the dimension  $n$  increase, the empirical mean along the coordinates of the G-vars of an  $\alpha$ -controlled function  $\psi$  converges to the expected value. Or, more intuitively, for each  $i$ ,  $(g_i^1, \dots, g_i^M) \approx \mathcal{N}(\mu, K)$ .

Intuitively, Theorem 3.3 states that  $g_i^{ct} \stackrel{d}{\approx} \mathcal{N}(\mu^c, K^c)$  for large  $t$ , iid for each  $i$ . This, jointly with the definitions of (3.3) and (3.4), means, roughly speaking, that the G-vars (preactivations, CFR Section 3.1) created from the same matrix  $A^k$  have nonzero correlations, but otherwise are asymptotically independent unless they appear together in a LinComb. The latter can be seen by checking which entries of  $K^c$  are nonzero by (3.4), in fact, the only nonzero values are those corresponding to the first four branches, which characterize G-vars that are either correlated by initialization (first branch), or if at some point they pass through the same matrix  $A^k$  (fourth branch), as the second and third, being recursive, will at some point lead to one of those cases.

### 3.3 Neural Tangent Kernel

As we have seen in Section 3.2, each neural network architecture is equivalent, in the infinite width limit to a Gaussian Process. This implies that we can study the regression we perform with the ANN as a kernel method, once we understand which kernel corresponds to its training. In this section,  $\{x^i\}_{i \in [N]} \subset \mathbb{R}^{\text{dim}_{\text{in}}}$  denote the network inputs,  $\theta \in \mathbb{R}^P$  contains all the parameters, distributed at initialization as in Section 3.2, and  $f(\theta, x)$  is the output of the network, which, for notation simplicity, we suppose unidimensional in this section.

As in [4] and [1], we consider a training dataset  $\{(x^i, y^i)\}_{i=1}^N \subset \mathbb{R}^{d_{\text{in}}} \times \mathbb{R}$  and we train the neural

network by minimizing a convex loss function

$$l(\theta) = \sum_{i=1}^N \mathcal{L}(f(\theta, x^i), y^i).$$

We update the parameters at step  $t$  using gradient descent with an infinitesimally small learning rate  $\frac{d\theta(t)}{dt} = -\nabla l(\theta(t))$ . With this setup, if we consider  $l$  to be the mean squared error, by a simple differentiation [1, Lemma 3.1] we see that the outputs of the network  $u(t) = (f(\theta(t), x^i))_{i \in [N]}$  evolve according to

$$\frac{du(t)}{dt} = -\text{NTK}(f_t) \cdot (u(t) - y), \quad (3.5)$$

where  $\text{NTK}(f_t)$  is the  $n \times n$  positive semidefinite matrix given by the *Neural Tangent Kernel*.

**Definition 3.4.** With the notation previously introduced, we define the *Neural Tangent Kernel*  $\text{NTK}(\cdot, \cdot)$  as the kernel whose values on points  $x, x'$  are given by

$$\text{NTK}(x, x') = \left\langle \frac{\partial f(\theta, x)}{\partial \theta}, \frac{\partial f(\theta, x')}{\partial \theta} \right\rangle. \quad (3.6)$$

For any kernel  $K$ , we abuse the notation and, when the context make it clear, use  $K$ , or specifically  $\text{NTK}$ , to define the matrix whose entries  $i, j$  are given by  $K(x^i, x^j)$ .

As we can see, the NTK depends on the parameters of the network and thus it is random at initialization and varies during training. Nonetheless, as NTK is an  $\alpha$ -controlled function on the inputs  $x, x'$ , if we let the network go to the infinite width limit, as explained in Section 3.2, we can apply Theorem 3.3 and see that the tangent kernel converges to a finite limit, as proved in [4] and [9].

**Definition 3.5.** We denote this limiting kernel  $\text{NTK}_\infty$ .

Going back to the dynamic expressed in (3.5), we see that if the network is infinitely wide, the expression is identical to that of *kernel regression* under gradient flow, as  $\text{NTK}(f_t) = \text{NTK}_\infty$  is constant during training, [4].

Although we started by considering only the mean squared error loss, it turns out that this idea applies to a wide variety of cases. In fact, in [4], Jacot et al. prove that the network function of an MLP evolves along the kernel gradient given by the Neural Tangent Kernel for any convex loss function. We conjecture that this result holds for any deep learning architecture.

### 3.4 Kernel Regression

Before moving to the experimental part, we remind the reader some concepts about *kernel regression*. Suppose we have a kernel function  $K(\cdot, \cdot)$  and a training dataset  $\{(x^i, y^i)\}_{i=1}^N \subset \mathbb{R}^{d_{\text{in}}} \times \mathbb{R}$ , kernel regression aims to approximate the underlying function with the linear combination

$$f_K(\cdot) = \sum_{i=1}^N \alpha^i K(\cdot, x^i). \quad (3.7)$$

The coefficients are estimated to minimize the loss function over prediction points

$$l(\alpha) = \mathcal{L}(f(x_*), y_*),$$

and, in practice, this minimization is carried over the training dataset

$$\hat{\alpha} = \arg \inf \left\{ \sum_{i=1}^N \mathcal{L} \left( \sum_{j=1}^N \alpha_j K(x^i, x^j), y^i \right) : \alpha_1, \dots, \alpha_N \in \mathbb{R} \right\}. \quad (3.8)$$

There is a nontrivial result linking a fully trained wide MLP  $f_{nn}$  to the kernel regression predictor  $f_{\text{NTK}}$ , using the  $\text{NTK}_{\infty}$ . If we consider a fresh observation  $x^*$  and let  $f_{nn} := \lim_{t \rightarrow \infty} f(\theta(t), x^*)$ , where  $f(\theta(t), \cdot)$  is the MLP output function as in Section 3.3, we can prove the following theorem.

**Theorem 3.6** (Arora et al. 2019, [1]). *Suppose an MLP with ReLU activation function and  $L$  hidden layers with  $m$  nodes, where  $m \geq \text{poly}(1/\kappa, L, 1/\lambda_0, N, \log(1/\delta))$ ,  $N$  being the size of the training dataset,  $1/\kappa = \text{poly}(1/\epsilon, \log(n/\delta))$  and  $\lambda_0$  being the smallest eigenvalue of the matrix  $\text{NTK}_{\infty}$  over the training data. Then, for any  $x^* \in \mathbb{R}^{d_{\text{in}}}$  with  $\|x^*\| = 1$ , with probability at least  $1 - \delta$  over the random initialization, we have*

$$|f_{nn}(x^*) - f_{\text{NTK}}(x^*)| \leq \epsilon.$$

The main consequence of this theorem is that, if a network is big enough, then we can expect its performance to be similar to that of the NTK regressor, with a non-asymptotic bound.

### 3.5 Multidimensional output

All we stated before about neural networks and kernels generalize immediately to the multidimensional case upon noting that a function  $f : \mathbb{R}^{d_{\text{in}}} \rightarrow \mathbb{R}^{d_{\text{out}}}$  is equivalent to a function  $\tilde{f} : \mathbb{R}^{d_{\text{in}}} \times [d_{\text{out}}] \rightarrow \mathbb{R}$  with  $\tilde{f}(x, k) = f_k(x)$ . This said we can generalize kernel regression, with each entry of the kernel  $K(x, x')$  now being a matrix instead of a scalar and  $\alpha$  being a vector, such that the vector-matrix product in (3.7) produces indeed a vector output. In this case, the kernel matrix  $K$  is a block matrix, where each block is given by the matrix  $K(x^i, x^j)$ .

For the Neural Tangent Kernel, this means that

$$[\text{NTK}(x^i, x^j)]_{k, k'} = \left\langle \frac{\partial f_k(\theta, x^i)}{\partial \theta}, \frac{\partial f_{k'}(\theta, x^j)}{\partial \theta} \right\rangle, \quad (3.9)$$

with  $k, k' \in [d_{\text{out}}]$ . An interesting point of the multidimensional approach is that, as we can see in (3.9), the kernel also encodes information between different output coordinates, i.e.  $k \neq k'$ .

## 4 NTK implementation

To practically test the theoretical results on different deep learning architectures, we choose to work using the efficient and popular library `Pytorch` [8], which is highly optimized for tensorial calculus. We gave in (3.6) an explicit formula to compute the NTK of a given neural network. This formulation is very suitable for the `Autograd` functionality of `Pytorch`. In fact, this module creates a computational graph when input is passed through the ANN and it is then possible to quickly compute derivatives with respect to the network parameters.

## 4.1 NTK for finite networks

To get the exact NTK corresponding to a finite network, we choose therefore to compute the full Jacobian  $\text{Jac}$  of the function  $f(\theta, (x^1, \dots, x^N))$ , where we pass all the input variables in a batch. In this way, we get a tensor of size  $N \times d_{\text{out}} \times P$ , where  $N$  is the number of data points,  $d_{\text{out}}$  is the length of the response vector and  $P$  is the number of parameters in the network, i.e. the length of  $\theta$ . At this point, we can compute the kernel by multiplying the Jacobian by its transpose (as it is a tensor we intend the transpose along the two first axes, corresponding to the input number and the coordinate of the output layer) and then by adding up the entries over the third axis, namely the one corresponding to parameters. This last step can be written in one line using Einstein summation convention, and computed efficiently in `Pytorch` with the function `einsum` and then reshaping the matrix:

```
einsum('abp, cdp -> abcd', Jac, Jac).reshape(N * dim_out, N * dim_out)
```

There are two main limitations to this approach. First of all, the Jacobian requires a lot of memory to be stored. For instance, if we consider an MLP with 3 hidden layers, 100 nodes in each layer and output size of 10, with biases at each layer; and we want to use it for a handwritten digit classification, for which the standard input images have  $28 \times 28$  grayscale pixels, our network would have  $99710 \approx 10^5$  parameters. This, jointly with a 1000 regression images, would require a tensor of size  $10^3 \times 10 \times 10^5$ , thus with a billion entries. Supposing we store each digit at 32-bit precision, the Jacobian would require approximately 3GB of memory. Then, another Jacobian would be necessary for prediction, thus doubling the memory required, if we suppose the test set to have equal size as the training one.

Secondly, as a consequence of `Pytorch` differentiation implementation, to retrieve the full Jacobian we need to iterate both on input images and on output dimensions. This is because differentiation always return a tensor of the same shape as the variable by which we differentiate and, thus, in case the derivative is a tensor of higher dimension, requires a direction over which aggregate the derivatives, which in this case would be a matrix with rows correspond to images and columns to output coordinates. This double loop seriously affects computation time, as the very fast `Pytorch` backend is accessed without any optimization through the `python` loops, which are very slow as it is an interpreted language.

## 4.2 NTK at infinite width

To estimate  $\text{NTK}_\infty$ , i.e. the kernel corresponding to the ANN in the infinite width limit, we have different options. The first one would be to find an explicit formula and exactly compute the kernel function, as Arora et al. did in [1]. Although explicit formulas are already known for MLPs and elementary CNNs, to compute them still requires to go through all layers, all input images and all output dimensions, which implies more and more computations the deeper the network is. Furthermore, every different architecture would explicitly require an explicit formula, which should be obtained beforehand.

We can avoid the latter problem leveraging on the fact that in the infinite width limit if our conjecture is right, each layer of the network has a kernel, which converges to the theoretical limiting one. We could thus use the already cited results, and iteratively compute an estimator of the kernel by passing through layers. By using the fact that, asymptotically, the kernel for each node is independent from the others, at least for MLPs [4], we can average of them to get an



Figure 4: Sample of 8 images from the MNIST dataset. The grayscale pictures represent, in the order, the digits 5, 0, 7, 1, 6, 9, 0, 7.

estimate of the actual value. Nonetheless, when considering more elaborate architectures, such as CNN or transformers, obtaining the NTK is not as straightforward.

For this reason, we choose to follow the same approach as for the finite case. More precisely, we initialize many big networks, compute the NTK for each of them and then estimate the limiting kernel as their average.

## 5 Experimental results

In this section we focus on testing experimentally our conjecture, namely that all the results presented in Section 3 generalize to general Deep Learning architectures as well. In particular, it has already been tested that for MLPs with a single output, regression based on NTK obtains results equivalent, or even better, than those of the original network. On the other hand, the same does not hold for CNNs, where experimental results show that the actual network greatly outperform the kernel methods [1]. Our objective is to understand what the network is actually learning, to confront it with what the  $NTK_\infty$  regression function, furthermore we study networks with multidimensional output.

More precisely, given an ANN with random initialization, we can compute its actual NTK, as explained in Section 4.1, and its theoretical-infinite-wide counterpart  $NTK_\infty$ . Then, we train the ANN to get  $f_{nn}$ , which produces another tangent kernel  $NTK_{tr}$ . By [4], we know that  $NTK_\infty$  does not change during training, so we can study the kernel regression functions  $f_{NTK_\infty}$  and  $f_{NTK_{tr}}$  given respectively by the infinite width Tangent Kernel and the NTK obtained by the finite and trained Network.

### 5.1 The dataset

We perform an handwritten-digit classification task, using the MNIST dataset [6]. It consists in grayscale images of handwritten digits, whose size is 28x28 pixels, and their labels, which correspond to the 10 digits from 0 to 9. We use 500 samples for train and we study the Kernel on a test of 500 samples. The two sets are disjoint and randomly drawn from the full dataset, using `Pytorch` functionalities.

We normalize the images beforehand, as the mean and standard deviation are well known, so that the observation space is supported in the unit sphere, which is one of the hypothesis used by Jacot et al. in [4].

To train the ANNs and the coefficients for kernel regression we use the Cross entropy loss, which is the standard when performing a classification task. To optimize the parameters we use `Adam` optimizer [5] over 5000 epochs.

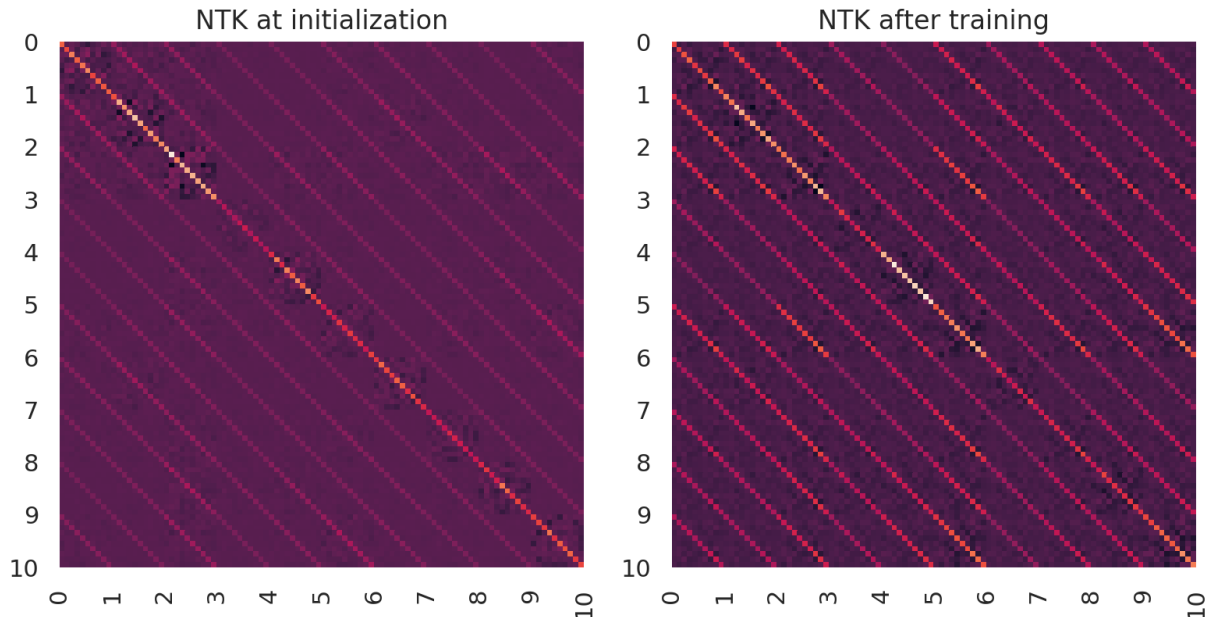


Figure 5: Neural Tangent Kernel of the MLP at initialization (Left) and after fully training the network (Right). The plot shows the entries of the first  $10 \times 10$  blocks of the neural tangent kernel. Recall that each block represent  $\text{NTK}(x, x')$  and it is a  $10 \times 10$  matrix, whose entries  $k, l$  correspond to the scalar product of the gradients of  $f_k$  and  $f_l$  with respect to the model parameters. Both images use a colour code to visualize the matrix and represent values close to zero in dark purple, but they have different scales. In fact, white values on the left are close to 24, while on the right they are close to 24000. This shows that the kernel does indeed move during training in the finite case.

## 5.2 Multilayer Perceptron

We start with the artificial neural network on which all theoretical results have been proved. In particular, we test an architecture with three fully-connected hidden layers and a readout layer. The input dimension is of  $784 = 28 \cdot 28$  nodes and all hidden layers have the same number of nodes. Combined with the readout layer, this gives a total of 99710 parameters.

We proceed as explained beforehand. We estimate the limiting NTK at initialization on a big network with 200 nodes. Then we initialize an MLP with the same architecture but only 100 nodes and train it as explained beforehand. Computing the tangent kernel of the latter gives us two different kernels  $\text{NTK}_\infty$  and  $\text{NTK}_{tr}$ , whose first entries we illustrate in Figure 5.

We see that with 100 nodes the kernel still moves consistently during training. We then test if this change affects the performance of Kernel regression. As we did with the ANN, we train the coefficients of the two kernel regressor,  $f_{\text{NTK}_\infty}$  and  $f_{\text{NTK}_{tr}}$ . Table 1 reports the results for the classification task. We see that on training all regressors perfectly fit the data, while the accuracy on test differs. In particular, none of the kernel regressors performs as well as the actual network, even though the kernel of the trained, smaller, network is closer to the expected result than  $\text{NTK}_\infty$ .

## 5.3 Convolutional Neural Network

We now test the theoretical results on a convolutional neural network. In particular, we define an architecture with three convolutional layers, the first two with filters of size  $5 \times 5$  and the last

	Test accuracy	Train accuracy
MLP	0.844	1.0
NTK <sub>∞</sub>	0.73	1.0
NTK <sub>tr</sub>	0.804	1.0

Table 1: Accuracy scores for the handwritten-digit recognition task. MLP is the fully trained multilayer perceptron, while NTK<sub>∞</sub> and NTK<sub>tr</sub> refers to the two kernel regressors using the respective kernels.

one with a filter of size  $20 \times 20$ , so that the output will be a tensor of size  $1 \times 1 \times 10$ , each channel corresponding to the prediction for the corresponding digit. Again, the input dimension is a tensor of size  $28 \times 28 \times 1$  and the number of channels passes from 1 to 15, to 15 and finally to 10. All the weights and biases considered, we have a total of 90550 parameters.

We proceed as in previous section, but this time, for computational limitation, we use the same dimensions for both the network estimating the CNTK<sub>∞</sub> and that which we train. Again, we end up with two different kernels CNTK<sub>∞</sub> and CNTK<sub>tr</sub>, whose first entries we illustrate in Figure 6.

We see that with this setup the kernel still moves during training. It is interesting to note that the magnitude of the entries increases notably, as for the MLP, but the structure of the matrix, in terms of which values are close to zero and which are far from it, remains very similar.

As before, we study how this change affects the performance of Kernel regression. We train the coefficients of the two kernel regressor,  $f_{\text{CNTK}_\infty}$  and  $f_{\text{CNTK}_{tr}}$  and Table 2 reports the results for the classification task. We see that on training the network and the trained kernel perfectly fit the data, while the regressor using CNTK<sub>∞</sub> does not fully train with 5000 epochs. As for the MLP case, the accuracies on the test set differ. Even though all performances are better than the MLP-related ones, none of the kernel regressors performs as well as the actual network. Nonetheless, it is notable that the differences presented are of the same order as before, with the initialization kernel scoring 10% worse than the actual network and 5% worse than the kernel obtained after training.

	Test accuracy	Train accuracy
CNN	0.894	1.0
CNTK <sub>∞</sub>	0.802	0.928
CNTK <sub>tr</sub>	0.84	1.0

Table 2: Accuracy scores for the handwritten-digit recognition task. MLP is the fully trained multilayer perceptron, while CNTK<sub>∞</sub> and CNTK<sub>tr</sub> refers to the two kernel regressors using the respective kernels.

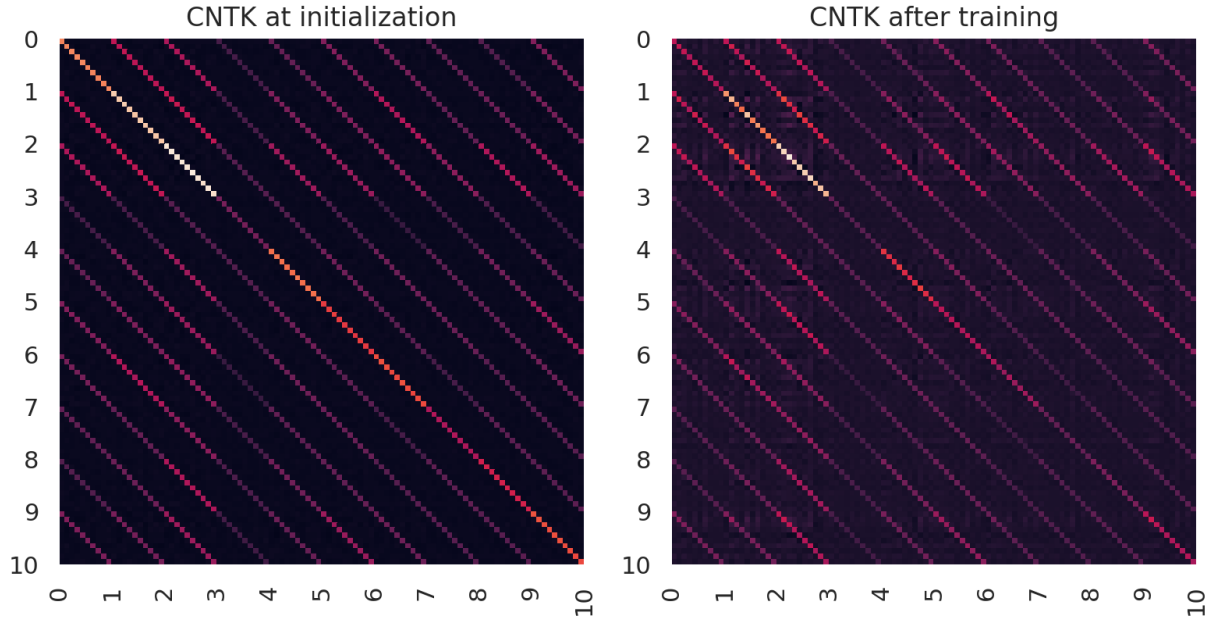


Figure 6: Convolutional Neural Tangent Kernel at initialization (Left) and after fully training the CNN (Right). The plot shows the entries of the first  $10 \times 10$  blocks of the neural tangent kernel. Recall that each block represent  $\text{CNTK}(x, x')$  and it is a  $10 \times 10$  matrix, whose entries  $k, l$  correspond to the scalar product of the gradients of  $f_k$  and  $f_l$  with respect to the model parameters. Both images use a colour code to visualize the matrix and represent values close to zero in dark purple, but they have different scales. In fact, white values on the left are close to 300, while on the right they are close to  $10^5$ . This shows that the kernel does indeed move during training in the finite case.



## References

- [1] S. Arora, S. S. Du, W. Hu, Z. Li, R. Salakhutdinov, and R. Wang. On exact computation with an infinitely wide neural net. *arXiv preprint arXiv:1904.11955*, 2019.
- [2] F. Fleuret. Deep learning - ee-559, February 2019.
- [3] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.
- [4] A. Jacot, F. Gabriel, and C. Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, pages 8571–8580, 2018.
- [5] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [6] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist>, 2, 2010.
- [7] R. M. Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.
- [8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [9] G. 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.
- [10] G. Yang. Tensor programs i: Wide feedforward or recurrent neural networks of any architecture are gaussian processes. *arXiv preprint arXiv:1910.12478*, 2019.