Exploring the Computational Properties of Infinite Width Neural Networks

Theories of Deep Learning

Candidate Number: 1061996

Contents

1 Introduction

The rapid growth of deep learning across a range of domains has lead to many open problems in the area [12]. Two of the most challenging problems in deep learning are how to characterise the training and generalisation of deep neural networks. Networks are often vastly overparameterised and have non-convex activation functions forcing the loss landscapes of the resulting neural networks to be complex and non-convex [9]. Nonetheless, these networks can often be trained to produce global minima of the loss function when applied to the training data [1, 5]. Not only that, but they then generalise well when applied to "unseen" test data [13].

An important feature that has allowed us to make advancements in recent years is that increasing the widths of deep neural networks increases their regularity, which in turn makes them easier to analyse. A natural question to ask is whether continually increasing this width such that the network becomes infinitely wide will give us a deeper insight into the behaviour of networks. This turns out to be the case and introduces us to an a great theoretical tool for understanding deep networks: the neural tangent kernel (NTK). The insights we gain from studying infinite networks can often be applied to finite networks, moreover they are often useful models in their own right [12].

It can be shown that neural networks in the infinite width limit simplify to linear models via Taylor expansion about the network's initialisation parameters. The neural tangent kernel at initalisation governs this expansion and, in certain cases, allows us to find closed form solutions for the evolution of the output of the neural network throughout training [9].

Here, we analyse infinite width neural networks and the neural tangent kernel, we show how they can be used to approximate the outputs of neural networks during and after training and we bound how accurate those approximations are. We then train some neural networks and compare the predictions of infinitely wide networks to the predictions of finite networks throughout training.

2 Linearisation of a Neural Network

2.1 Definitions and Notation

To start with we will define the type of neural networks that we will discuss in this essay and give the notation we will use for them. We will be following the defintions and notation given in Wide Neural Networks of Any Depth Evolve as Linear Models Under Gradient Descent [9] as we want to define our network in the same way.

We will mainly be discussing fully connected feedforward neural networks with L hidden layers with widths n_l, for $l = 1, 2, \dots, L$ and a readout layer with $n_{L+1} = k$. We use $h^l(x), x^l(x) \in \mathbb{R}^{n_l}$ to represent the pre- and post-activation functions at layer l where $x \in \mathbb{R}^{n_0}$ is an input. The recurrence relation for our feedforward network is defined by

$$\begin{cases} h^{l+1} &= x^l W^{l+1} + b^{l+1} \\ x^{l+1} &= \phi(h^{l+1}) \end{cases}$$

and,

$$\begin{cases} W^l_{i,j} &= \frac{\sigma_w}{\sqrt{n_l}} w^l_{ij} \\ b^l_j &= \sigma_b \beta^l_j \end{cases}$$

where ϕ is a pointwise activation function, $W^{l+1} \in \mathbb{R}^{n_l \times n_{l+1}}$ and $b^{l+1} \in \mathbb{R}^{n_{l+1}}$ are the weights and biases, w^l_{ij} and β^l_j are the trainable variables, drawn i.i.d. from a standard Gaussian $w^l_{ij}, \beta^l_j \sim \mathcal{N}(0, 1)$ at initialisation, and σ^2_w and σ^2_b are weight and bias variances.

We note that our definition of the parameters of the neural network are slightly different to the more common LeCun initialisation, where everything is the same except that the σ_b, and σ_w terms are absent. The factors $\frac{1}{\sqrt{n_l}}$ in our initialisation are essential if we are to gain a consistent asymptotic behaviour of neural networks as the width of the hidden layers goes to infinity. The factors scale the derivatives with respect to the weights of the function generated by the neural network. This greatly reduces the effect of weights during training as n_l grows, but we make up for this by introducing σ_b so the influence of the bias terms is not too great. Note that we can represent all the same functions with both parameterisations [8].

Furthermore, we define the $((n_{l-1} + 1)n_l) \times 1$ vector of all network parameters at layer l by $\theta^l := \text{vec}(W^l, b^l)$ and the vector of all network parameters by $\theta = \text{vec}(\cup_{l=1}^{L+1} \theta^l)$. We will talk about training as a function of time hence $\theta(t)$ will represent all network parameters at time t and $\theta_0 := \theta(0)$ will represent the all network parameters at initialisation. We will use $f(x; \theta(t)) := h^{L+1}(x) \in \mathbb{R}^k$ to denote the output of the neural network at time t when applied to an input $x \in \mathbb{R}^{n_0}$. We denote the loss function by $\ell(\hat{y}, y) : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$, where the first argument is the prediction and the second argument is the true label. We want to minimise the empirical loss $\mathcal{L}(\theta) = \sum_{(x,y)\in\mathcal{D}} \ell(f(x; \theta), y)$, where $\mathcal{D} \subset \mathbb{R}^{n_0} \times \mathbb{R}^k$ denotes the set of training data. Lastly, we denote the inputs and labels by $\mathcal{X} = \{x : (x, y) \in \mathcal{D}\}$ and $\mathcal{Y} = \{y : (x, y) \in \mathcal{D}\}$, respectively, this allows us to define $f(\mathcal{X}; \theta(t)) = \text{vec}([f(x; \theta(t))]_{x\in\mathcal{X}}$, the $k|\mathcal{D}| \times 1$ vector of concatenated outputs for all the training data. We can now move on to the theory of finite networks [9].

2.2 Linearising a Finite Network

Consider a fully connected feedforward neural network being trained. The wider the network the more neurons and therefore the more weights and biases there are to affect the output. If we define the relative change in parameters at time t by $\frac{||\theta(t)-\theta_0||_2}{||\theta_0||_2}$, we can intuitively see that the wider the network the less the relative change in parameters during training. This is because if each parameter only changes a small amount we could still end up with a large difference in the overall network function for a wide network, hence the parameters do not have to change much in order to fit the data. Therefore if we continue to make our network wider we would see very little relative change in the parameter values over the course of training [6].

This would suggest that we might be able to get a good approximation of our trained neural network by Taylor expanding $f(x; \theta(t))$ around its initialisation and ignoring terms greater than first order [6]. This gives

$$f^{lin}(x; \theta(t)) := f(x; \theta_0) + \nabla_\theta f(x; \theta_0)\omega(t)$$

where $\omega(t) := \theta(t) - \theta_0$ is the change of parameters from their initialisation to time t and $\nabla_\theta f(x; \theta_0)$ is the Jacobian of f at initialisation when applied to an input $x \in \mathbb{R}^{n_0}$. This has changed our entire nonlinear network function into a linear function of its parameters [9].

After initialisation, this formula has one constant term and one term governed by the change in $\omega(t)$ therefore significantly simplifying the dynamics of our neural network. We have to ask when this is valid since for many nonlinear neural networks this clearly is not the case.

The phenomenon of models behaving like their linearisation around initialisation is called lazy training. We will here follow the paper On Lazy Training in Differentiable Programming [3] to gain some insight on when this happens.

In this paper, when optimising a neural network we will always use gradient descent or some variation of it, so consider a gradient decsent step $\theta_1 = \theta_0 - \eta\nabla_\theta\mathcal{L}(\theta_0)$ where η is the step size (also known as the learning rate). Assuming that we are not already at a minimum so $\mathcal{L}(\theta_0) > 0$ and we are not at a critical point so $\nabla_\theta\mathcal{L}(\theta_0) \neq 0$ then, provided the learning rate is small enough, this should cause $\mathcal{L}(\theta)$ to decrease.

Here, lazy training refers to when the first-order derivative of f does not change much, but $\mathcal{L}$ significantly decreases in the relative sense. That is, $\Delta(\mathcal{L}) \gg \Delta(\nabla_\theta f)$ where $\Delta(\mathcal{L}) := \frac{|\mathcal{L}(\theta_1) - \mathcal{L}(\theta_0)|}{\mathcal{L}(\theta_0)}$, $\Delta(\nabla_\theta f) := \frac{||\nabla_\theta f(\theta_1) - \nabla_\theta f(\theta_0)||}{||\nabla_\theta f(\theta_0)||}$ and $||\cdot||$ is the operator norm. We can estimate the relative change in $\mathcal{L}$ using its Taylor expansion and then the gradient descent equation like so

$$\Delta(\mathcal{L}) = \frac{|\mathcal{L}(\theta_1) - \mathcal{L}(\theta_0)|}{\mathcal{L}(\theta_0)} \approx \frac{|\nabla_\theta \mathcal{L}(\theta_0) \cdot (\theta_1 - \theta_0)|}{\mathcal{L}(\theta_0)} \tag{1}$$

$$= \eta \frac{||\nabla_\theta \mathcal{L}(\theta_0)||^2}{\mathcal{L}(\theta_0)}. \tag{2}$$

We do the same for the relative change in $\nabla_\theta f$ then get an inequality by splitting the operator norm of a product of functions into the product of two operator norms of functions. Thus we see that lazy training is guaranteed when

$$\frac{||\nabla_\theta \mathcal{L}(\theta_0)||}{\mathcal{L}(\theta_0)} \gg \frac{||\nabla_\theta^2 f(\theta_0)||}{||\nabla_\theta f(\theta_0)||}.$$

This can be simplified further when using the mean squared loss function $\ell(\hat{y}, y) = \frac{1}{2}||\hat{y} - y||^2$. In this case $||\nabla_\theta \mathcal{L}(\theta_0)|| = ||\nabla_\theta f(\theta_0)^T (f(\theta_0) - \hat{y})|| \approx ||\nabla_\theta f(\theta_0)|| \cdot ||f(\theta_0) - \hat{y}||$ so we get the condition

$$\kappa_f(\theta_0) := ||f(\theta_0) - \hat{y}|| \frac{||\nabla_\theta^2 f(\theta_0)||}{||\nabla_\theta f(\theta_0)||^2} \ll 1$$

and we call $\kappa_f(\theta_0)$ the inverse relative scale of the model f at θ_0.

Now that we have a quantifiable result to show when linearisation is accurate, we want to know what happens to $\kappa_f(\theta_0)$ as the width of the hidden layers tends to infinity. It is important to remember that $\kappa_f(\theta_0)$ is actually a random variable as the initialisation of the parameters θ_0 is random given that each of the weights and biases are Gaussian random variables. As such, when we talk about what κ_f converges to in the infinite width limit we will talk about what its expectation converges to.

Unfortunately it is not as easy as we might hope to use this to say that all neural networks of large enough width can be treated as their linearisations. However, suppose that we have a two layer fully connected neural network defined like ours from Section 2.1 but with no bias terms. More specifically, suppose we have a function of the form

$$f(x; \theta) = \frac{1}{\sqrt{m}} \sum_{j=1}^{m} b_j \cdot \phi(a_j \cdot x)$$

where $m \in \mathbb{N}$ is the size of the hidden layer and $a_i, b_j \in \mathbb{R}$ for $i, j \in \{1, \ldots, m\}$ are the weights of the network. It can be shown that

$$\mathbb{E}[\kappa_f(\theta_0)] \lesssim 2m^{-\frac{1}{2}}.$$

for large m. Therefore we have a result that tells us that these two layer networks can be treated as their linearisations when the width of the hidden layer is large and therefore backs up our intuition that the weights of the network do not change much when training large width networks [3, 6].

Although this is only shown to be true in this specific case, this suggests that we may also be able to treat deeper networks as their linearisation and so we will investigate this further. However, first we will look into how we can use these linearisations to make predictions of the network function.

3 Gradient Flow Training of Finite Networks

Now we have seen that certain networks in this infinite width limit can be linearised we will investigate how to get predictions using the linearised form. The developments here will allow us see the training of neural networks in a different light. More specifically, they will allow us to study the training of neural networks in function space rather than just parameter space. In this section we will mainly be following the paper Wide Neural Networks of Any Depth Evolve as Linear Models Under Gradient Descent [9].

To start with we consider the evolution of our parameters θ and neural network function f through the continuous time version of gradient descent known as gradient flow. Intuitively, we would want $\theta(t)$ to change in the direction of steepest descent for the loss function $\mathcal{L}(\theta(t))$, that is

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = -\eta\nabla_\theta\mathcal{L}(\theta(t)) = -\eta\nabla_\theta f(\mathcal{X};\theta(t))^T\nabla_{f(\mathcal{X};\theta(t))}\mathcal{L} \tag{3}$$

using the chain rule for the second equality and where η is a scaling constant, the learning rate.

This enables us to find a formula for the rate of change of the neural network through gradient flow involving a new tool: the neural tangent kernel as follows

$$\begin{aligned}\frac{\mathrm{d}f(\mathcal{X};\theta(t))}{\mathrm{d}t} &= \nabla_\theta f(\mathcal{X};\theta(t))\frac{\mathrm{d}\theta}{\mathrm{d}t} &(4)\\ &= -\eta\nabla_\theta f(\mathcal{X};\theta(t))\nabla_\theta f(\mathcal{X};\theta(t))^T\nabla_{f(\mathcal{X};\theta(t))}\mathcal{L} &(5)\\ &= -\eta\hat{\Theta}(t,\mathcal{X},\mathcal{X})\nabla_{f(\mathcal{X};\theta(t))}\mathcal{L}. &(6)\end{aligned}$$

The neural tangent kernel is thus the $k|\mathcal{D}| \times k|\mathcal{D}|$ dimensional matrix defined by

$$\hat{\Theta}(t) = \nabla_\theta f(\mathcal{X};\theta(t))\nabla_\theta f(\mathcal{X};\theta(t))^T = \sum_{l=1}^{L+1}\nabla_{\theta^l} f(\mathcal{X};\theta(t))\nabla_{\theta^l} f(\mathcal{X};\theta(t))^T \tag{7}$$

in this case.

More generally, the dimension of it depends on the output dimension of the neural network $k|\mathcal{D}|$ and for $n_0|\mathcal{D}|$ dimensional inputs to the neural network $x, y \in \mathbb{R}^{n_0|\mathcal{D}|}$ we get the $k|\mathcal{D}| \times k|\mathcal{D}|$ dimensional matrix given by

$$\hat{\Theta}(x, y; \theta(t)) = \nabla_\theta f(x; \theta(t)) \nabla_\theta f(y; \theta(t))^T \tag{8}$$

and we define $\hat{\Theta}_0(x, y) := \hat{\Theta}(x, y; \theta(0))$.

What it represents looks complex when the output of the neural network is a vector and we are working with more than one data point but when a neural network outputs a scalar and only one training point is used the formula simplifies considerably. In fact equation (8) becomes a dot product. Intuitively, looking at equation (5) the neural tangent kernel represents the influence of the loss gradient on the evolution of $f(x)$. Now, if we consider one test point and more than one training point equation (5) simplifies to

$$\frac{\mathrm{d}f(x; \theta(t))}{\mathrm{d}t} = -\eta \nabla_\theta f(x; \theta(t)) \nabla_\theta f(\mathcal{X}; \theta(t))^T \nabla_{f(\mathcal{X}; \theta(t))} \mathcal{L}.$$

Thus, each entry of the neural tangent kernel represents how the loss gradient influences $f(x)$ with respect to each of the training points, so the neural tangent kernel captures how each training point influences the output $f(x)$ throughout training. This then generalises again when you have higher dimensional data [7].

Although this does not look particularly useful at the moment, we will see that in the infinite width limit the neural tangent kernel converges in probability allowing us to calculate a deterministic solution to these equations in certain cases. Note that the neural tangent kernel is actually a random variable as it depends on the initialisation of the parameters.

To gain closed form solutions for the evolution of the neural network through training we will first have to linearise. Specifically we will be working with $f^{lin}(x; \theta(t)) = f(x; \theta_0) + \nabla_\theta f(x; \theta_0) \omega_t$ again.

Following similar steps to last time we can derive the equations for the training dynamics of the linear network, getting

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = -\eta \nabla_\theta f(\mathcal{X}; \theta_0)^T \nabla_{f^{lin}(\mathcal{X}; \theta_0)} \mathcal{L} \tag{9}$$

$$\frac{\mathrm{d}f^{lin}(x; \theta(t))}{\mathrm{d}t} = -\eta \hat{\Theta}_0(x, \mathcal{X}) \nabla_{f^{lin}(\mathcal{X}; \theta(t))} \mathcal{L}. \tag{10}$$

Comparing these to equations (3) and (6) we see that we gain simpler dynamics this time as once the network has been initialised $\nabla_\theta f_0(x)$, and therefore $\hat{\Theta}_0(x, \mathcal{X})$, will stay constant throughout training. We can now use the previously defined mean squared loss function to solve these ODEs for closed form solutions of ω_t and f^{lin}, giving f^{lin} as a sum of two functions for any arbitrary $x \in \mathbb{R}^{n_0}$ and $t \geq 0$ by

$$\omega_t = -\nabla_\theta f(\mathcal{X};\theta_0)^T \hat{\Theta}_0^{-1}(I - e^{-\eta\hat{\Theta}_0 t})(f(\mathcal{X};\theta_0) - \mathcal{Y}), \tag{11}$$

$$f^{lin}(\mathcal{X};\theta(t)) = (I - e^{-\eta\hat{\Theta}_0 t})\mathcal{Y} + e^{-\eta\hat{\Theta}_0 t} f(\mathcal{X};\theta_0) \tag{12}$$

$$\mu_t(x) = \hat{\Theta}_0(x,\mathcal{X})\hat{\Theta}_0^{-1}(I - e^{-\eta\hat{\Theta}_0 t})\mathcal{Y} \tag{13}$$

$$\gamma_t(x) = f(x;\theta_0) - \hat{\Theta}_0(x,\mathcal{X})\hat{\Theta}_0^{-1}(I - e^{-\eta\hat{\Theta}_0 t}) f(\mathcal{X};\theta_0) \tag{14}$$

$$f^{lin}(x;\theta(t)) = \mu_t(x) + \gamma_t(x). \tag{15}$$

These may still look random, but once we initialise the neural network we will be able to compute $\hat{\Theta}_0$ and $f(x;\theta_0)$ for all $x \in \mathbb{R}^{n_0}$. This gives us a closed form, deterministic solution for the linearisation of the neural network f^{lin} at any time during training.

Although we would now like to state a bound on the accuracy of the linear approximation there is still more work to do to get to that point. These equations still depend on the initialisation of the network parameters therefore pre-initialisation there is still a random element to our solution. To combat this we will use a trick we have discussed before: letting the width of the hidden layers of the network tend to infinity.

4 Infinite Networks

It was previously mentioned that increasing the width of hidden layers was known to increase the regularity of the network's loss surface [12], and we saw in Section 2 that in certain cases, this allows us to linearise our network function. This suggests it is a good step to do next in order to gain some insight into the training of neural networks.

4.1 The Neural Tangent Kernel in the Infinite Width Limit

Here, we look at a fundamental paper which generated huge interest when it was released as it was one of the first research papers to analyse the training dynamics of deep neural networks. The paper, Neural Tangent Kernel: Convergence and Generalization in Neural Networks [8], offers some great insight, however it is very advanced and would likely take another page or two just to write out the notation for. We will instead discuss this material at a lower level then, once we have the idea, we will follow other papers which apply it to different circumstances. It is important to note that in this paper their neural network is set up slightly different to ours, but the results can be modified to work for our network too.

The first thing we learn is that despite the initialisation of a neural network being random, in the infinite width limit the output functions at intialisation $\{f(x;\theta_0)\}_{x\in\mathcal{X}}$ converge to a multivariate Gaussian in distribution. This is implied by the Central Limit Theorem and is basically a result of the pre-activation functions being the sums of the weights and biases: these are Gaussian random variables hence the pre-activations are Gaussian random variables [8, 9].

This fact will enable us to study the behaviour of neural networks in terms of their distribution, that is without initialising them. Similarly, we find that at initialisation the neural tangent kernel converges in probability to an explicit deterministic limit Θ in the infinte width limit. This kernel only depends on the variance of the parameters at initialisation, the depth of the network and the choice of activation function. We can compute this kernel explicitly for certain activation functions, such as ReLu and erf. We note that this theory can also be extended to different architectures such as convolutional neural networks with the convolutional neural tangent kernel (CNTK) [2, 8, 9].

The paper then goes on to discuss that the neural tangent kernel will stay asymptotically constant during training (in the infinite width limit), this removes the parameter dependence on t during training. Unfortunately it defines a more general version of training using inner products so we will leave this paper here but continue to follow other papers which use this idea [7].

4.2 Describing Neural Networks in the Infinite Width limit

The fact that the neural tangent kernel stays constant during training (in the infinte width limit) is very powerful. We will show this now by analysing a neural network function in a way similar to the linearised network before, but this time gaining a kernel regression problem in the process.

Keeping the notation consistent, we will be discussing minimising the mean squared loss of a fully connected neural network. Here, we will only consider when the output of the neural network function $f(x;\theta)$ is a real number but this can be easily generalised to higher dimensions, the notation is just more complex. We will be working from a derivation given in On Exact Computation with an Infinitely Wide Neural Net [2].

To simplify notation slightly let $u(t) = f(\mathcal{X}; \theta(t)) \in \mathbb{R}^n$ with $n = |\mathcal{D}|$. Similarly to last time we will analyse the training of the network using gradient flow, but this time ignoring the learning rate. Using the chain rule on the mean squared loss, we get

$$\frac{\mathrm{d}\theta(t)}{\mathrm{d}t} = -\nabla_\theta \mathcal{L}(\theta(t)) = -\sum_{i=1}^{n}(f(x_i; \theta(t)) - y_i)\frac{\partial f(x_i; \theta(t))}{\partial \theta}, \tag{16}$$

where the x_i and y_i with $i = 1, \dots, n$ are the inputs and labels of our training data, respectively. Using the chain rule on f and then substituting in in a similar way to equation (3), we get

$$\frac{\mathrm{d}f(x_i; \theta(t))}{\mathrm{d}t} = -\sum_{j=1}^{n}(f(x_j; \theta(t)) - y_j)\left\langle \frac{\partial f(x_i; \theta(t))}{\partial \theta}, \frac{\partial f(x_j; \theta(t))}{\partial \theta} \right\rangle \tag{17}$$

for all $i \in \{1, 2, \dots, n\}$. This can then be written using our definition of u as

$$\frac{\mathrm{d}u(t)}{\mathrm{d}t} = -\hat{\Theta}(t)(u(t) - \boldsymbol{y}) \tag{18}$$

with $\boldsymbol{y} = (y_i)_{i \in \{1,2,\dots,n\}}$ being the vector of all training data outputs, and $\hat{\Theta}(t) = \hat{\Theta}(\mathcal{X}, \mathcal{X}, t)$ is the neural tangent kernel applied to our training data (written only as a function of time) as defined earlier. Hence we now see that the neural tangent kernel governs the gradient flow and therefore the training dynamics of this system.

We can now apply what we learned at the start of this section. As the widths of the hidden layers of the neural network tend to infinity the neural tangent kernel will converge in probability to a deterministic function Θ at initialisation and remain constant throughout training. This gives us $\hat{\Theta}(t) = \Theta$ for all $t \geq 0$ and

$$\frac{\mathrm{d}u(t)}{\mathrm{d}t} = -\Theta(u(t) - \boldsymbol{y}). \tag{19}$$

Remember, these theorems are only applicable if the network is initialised in a certain way. The initialisation we gave at the beginning of this essay will suffice.

The dynamics here are identical to the dynamcis of kernel regression under gradient flow with ker defined by $\ker(x, x') = \mathbb{E}_{\theta \sim \mathcal{W}} \left\langle \frac{\partial f(x; \theta)}{\partial \theta}, \frac{\partial f(x'; \theta)}{\partial \theta} \right\rangle$ where $x, x' \in \mathbb{R}^{n_0}$ and $\mathcal{W}$ is an initialisation distribution over θ. This gives the final prediction

$$f^*(x) = (\ker(x, x_1), \dots, \ker(x, x_n)) \cdot (\Theta)^{-1}\boldsymbol{y} \tag{20}$$

at $t \to \infty$, when $u(0) = \mathbf{0}$ [2, 4].

Following on from this there are two brilliant results in this paper [2], both with very advanced proofs so we will just state the results then move on to something more accessible. These results are derived using a parameter initialisation slightly different to ours in that there are no bias terms and the activation functions are multiplied by some constants at each layer. Nonetheless, these are important results, one of which being a generalisation of a result previously stated at the start of this section.

This result says that for a fully connected neural network, initialised as just described with ReLu activation function we can find a bound on the hidden widths that shows convergence of the neural tangent kernel at initialisation. More specifically:

Theorem 4.1. *Fix $\epsilon > 0$ and $\delta \in (0,1)$. Suppose $\sigma(z) = \max(0,z)$ and $\min_{h\in[L]} n_h \geq \Omega(\frac{L^6}{\epsilon^4}\log(L/\delta))$. Then for any inputs $x, x' \in \mathbb{R}^{n_0}$ such that $||x|| \leq 1, ||x'|| \leq 1$, with probability at least $1-\delta$ we have:*

$$\left|\left\langle \frac{\partial f(x;\theta)}{\partial \theta}, \frac{\partial f(x';\theta)}{\partial \theta} \right\rangle - \Theta(x,x')\right| \leq (L+1)\epsilon.$$

Note that this theorem is an improvement on the one given at the start of this section as the previous result was asymtotic whereas this provides a non-asymtotic bound on the widths of the hidden layers. This theorem only requires the minimum hidden layer width to be sufficiently large, which is the weakest notion of a limit.

The next result is the first we have seen that sheds light on how well we can approximate fully trained infinite neural networks.

For some test data $x_{te} \in \mathbb{R}^{n_0}$ we let $[\ker_{ntk}(x_{te}, \mathcal{X})]_i = \Theta(x_{te}, x_i)$. The prediction we get from kernel regression using the neural tangent kernel on the test data is $f_{ntk}(x_{te}) = (\ker_{ntk}(x_{te}, \mathcal{X}))^T \Theta^{-1} \boldsymbol{y}$. Lastly, let $f_{nn}(x_{te}) = \lim_{t\to\infty} f_{nn}(x_{te}; \theta(t))$ be the network function at the end of training, then we can state the theorem.

Theorem 4.2. *Suppose $\sigma(z) = \max(0,z)$, $1/\kappa = poly(1/\epsilon, \log(n/\delta))$ and $n_1 = n_2 = \cdots = n_L = n$ with $n \geq poly(1/\kappa, L, 1/\lambda_0, n, \log(1/\delta))$. Then for any $x_{te} \in \mathbb{R}^{n_0}$ with $||x_{te}|| = 1$, with probability at least $1-\delta$ over the random initialisation, we have*

$$|f_{nn}(x_{te}) - f_{ntk}(x_{te})| \leq \epsilon.$$

Now we return to the linearised problem from Section 3, this time with more knowledge of how infinite networks behave under training.

5 The Linearised Infinite Network

We left the linearised problem having just determined a closed form solution for the evolution of the linearised network function during training when using the mean squared loss. This was deterministic once the parameters had been initialised but we can also treat them as random variables and consider the resulting distribution we would get from training. As stated earlier, in the infinite width limit the network functions at initialisation $f(x;\theta_0)$ and $f(\mathcal{X};\theta_0)$ are Gaussians. Equations (13) and (14) are just applying affine transforms to them therefore $f^{lin}(x;\theta(t))$ converges in distribution to a Gaussian in the infinite width limit and we can calculate the mean and covariance of this distribution using these equations. We just need to define one last thing before we can state the theorem.

If we let f^i denote the i-th entry in the network function then we can define the sample-to-sample kernel function of the outputs in the infinite width setting as

$$\mathcal{K}^{i,j}(x,x') = \lim_{\min(n_1,\dots,n_L)\to\infty} \mathbb{E}[f^i(x;\theta_0)\cdot f^j(x';\theta_0)]. \tag{21}$$

Then $\mathcal{K}^{i,j}(x,x')$ gives the covariance between the i-th input of x and the j-th input of x' and, as a point of interest, $f(\mathcal{X};\theta_0) \sim \mathcal{N}(0,\mathcal{K}(\mathcal{X},\mathcal{X}))$. Each of the covariances can be computed recursively [9].

Theorem 5.1. *For every test point $x \in \mathcal{X}_T$, and $t \geq 0$, $f^{lin}(x;\theta(t))$ converges in distribution as width goes to infinity to a Gaussian with mean and covariance given by*

$$\mu(\mathcal{X}_T) = \Theta(\mathcal{X}_T,\mathcal{X})\Theta^{-1}(I-e^{-\eta\Theta t})\mathcal{Y}, \tag{22}$$

$$\begin{aligned}\Sigma(\mathcal{X}_T,\mathcal{X}_T) = {} & \mathcal{K}(\mathcal{X}_T,\mathcal{X}_T) + \Theta(\mathcal{X}_T,\mathcal{X})\Theta^{-1}(I-e^{-\eta\Theta t})\mathcal{K}(I-e^{-\eta\Theta t})\Theta^{-1}\Theta(\mathcal{X},\mathcal{X}_T) \\ & - (\Theta(\mathcal{X}_T,\mathcal{X})\Theta^{-1}(I-e^{-\eta\Theta t})\mathcal{K}(\mathcal{X},\mathcal{X}_T) + h.c.).\end{aligned} \tag{23}$$

Therefore, over random initialisation, $\lim_{t\to\infty}\lim_{n\to\infty} f^{lin}(x,\theta(t))$ *has distribution*

$$\begin{aligned}&\mathcal{N}(\Theta(\mathcal{X}_T,\mathcal{X})\Theta^{-1}\mathcal{Y}, \\ &\quad \mathcal{K}(\mathcal{X}_T,\mathcal{X}_T) + \Theta(\mathcal{X}_T,\mathcal{X})\Theta^{-1}\mathcal{K}\Theta^{-1}\Theta(\mathcal{X},\mathcal{X}_T) - (\Theta(\mathcal{X}_T,\mathcal{X})\Theta^{-1}\mathcal{K}(\mathcal{X},\mathcal{X}_T) + h.c.)),\end{aligned} \tag{24}$$

where "+h.c." is an abbreviation of "plus the hermitian conjugate" [9].

We can see that first formula is derived simply by plugging $\mathcal{X}_T$ into equation (13).

For ReLu and erf activation functions we can compute Θ exactly making this theorem an extremely powerful method for predicting the evolution of a network function through gradient descent training. We will call this method NTK-GP (short for neural tangent kernel Gaussian process) and use it to compare results with trained finite networks in the next section. First however, we are going to compare the dynamics of linear and nonlinear networks in the infinite width limit with some theory so we know that we are getting accurate predictions of our actual nonlinear infinite network function when using the linearisation.

Let $\eta_{critical} = 2(\lambda_{min}(\Theta) + \lambda_{max}(\Theta))^{-1}$, where $\lambda_{min/max}(\Theta)$ is the min/max eigenvalue of Θ. Then we we get the following theorem [9].

Theorem 5.2. *Let $n_1 = \cdots = n_L = n$ and assume $\lambda_{min}(\Theta) > 0$. Applying gradient descent with learning rate $\eta < \eta_{critical}$ (or gradient flow), for every $x \in \mathbb{R}^{n_0}$ with $||x||_2 \leq 1$, with probability arbitrarily close to 1 over random initialisation,*

$$\sup_{t\geq 0} ||f(x;\theta(t)) - f^{lin}(x;\theta(t))||_2, \ \sup_{t\geq 0} \frac{||\theta(t)-\theta_0||_2}{\sqrt{n}}, \ \sup_{t\geq 0} ||\hat{\Theta}(t) - \hat{\Theta}_0||_F = \mathcal{O}(n^{-\frac{1}{2}}), \ \text{as } n \to \infty. \tag{25}$$

Hence we see that in the infinite width limit, and for the mean squared loss, the gradient descent dynamics of the nonlinear neural network fall into its linearised dynamics regime. Given that the network function and the parameters were governed by equations (3) and (6) in the nonlinear model with Θ being a function of t, the equations would have been very difficult, if not impossible, to solve. It is therefore incredible that we have found this. This is of course to do with the convergence of the neural tangent kernel in the infinite width limit.

We can combine Theorem 5.1 and Theorem 5.2 to get

Theorem 5.3. *If $\eta < \eta_{critical}$ then for every $x \in \mathbb{R}^{n_0}$ with $||x||_2 \leq 1$, as $n \to \infty$, $f(x;\theta(t))$ converges in distribution to the Gaussian with mean and variance given by equation (22) and (23) [9].*

6 Experiments

We now have a wealth of theory which describes how infinite neural networks can be modelled. In this section we will apply what we have learned and evaluate the performance of our infinite width neural networks in a range of different ways.

In this section we will heavily rely on the module Neural Tangents [10] for Python, developed by a group of researchers at Google Brain and the University of Cambridge. It enables relatively easy programming and training of finite and infinite neural networks with some of the theory we have just dicussed being used in the implementation.

For example, Theorem 5.3 is used when we want to compute the distribution of an infinite neural network function.

Not only have they developed this package, they also released a paper to go alongside it, Neural Tangents: Fast and Easy Infinite Neural Networks in Python [11], which uses the module to produce some interesting results. However, here we will just use Neural Tangents to conduct some experiments of our own.

6.1 Finite Width Networks Versus Infinite Width Networks

The first and most obvious thing we would like to compare is how an infinite width network trained via gradient descent performs against a finite width network trained via gradient descent. To do this we have followed some of the code used in the Jupyter notebook provided with Neural Tangents but we have changed the function we are approximating and also provided approximations of the function using two networks with different hidden widths. We do this to get a good picture of the role the width of the hidden layers play in the evolution of a network through gradient decent training and to see how these networks compare to the infinte width case. To train the finite networks we will actually use stochastic gradient descent to reduce the computational cost. Nonetheless, stochastic gradient descent will provide a good approximation of the network being trained by full batch gradient descent.

We want to use the neural networks to approximate the function $f(x) = x^3$ for $x \in [0, 1]$. We do this by taking 5 uniformly distributed points from the domain, computing $f(x)$ then adding some Gaussian random noise to each of them. That is, $y_i = x_i^3 + \epsilon_i$ where $x_i \sim \text{Uniform}(0, 1)$, $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ are independent and identically distributed for all $i \in \{1, \cdots, 5\}$. These are our training data. In the finite case we then take a random initialisation of the network and train it using this data with stochastic gradient descent. In the infinite case the program uses the result of Theorem 5.3 to approximate f while also calculating the standard deviation of the uncertainty in the prediction.

We use a 3 layer fully connected feedforward neural network with erf activation functions in all of these cases. The feedforward network is defined as in Section 2 with $\sigma_w = 1.5$ and $\sigma_b = 0.05$ and, in the finite case the hidden layers are of width 512. Also, we always use the mean squared loss function and set the variance of the noise to be $\sigma = 0.01$.

We can now see empirically that wide neural networks trained by stochastic gradient descent can be approximated very well by infinte width networks.

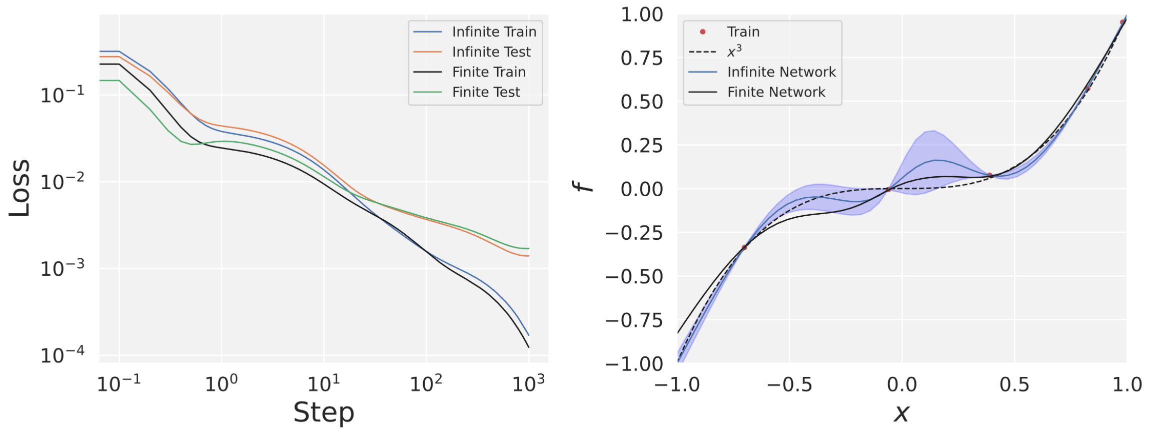

Figure 1: This figure compares the training dynamics of a finite neural network with an infinite width neural network with all parameters of the networks as described above. The graph on the left shows the evolution of the mean squared loss of the finite network and mean squared loss of the mean of the infinite network throughout training. This is shown on the training and the test data. The graph on the right compares the prediction of the infinite width network at infinite time and the finite network after 10,000 steps of stochastic gradient descent. The shaded region denotes two standard deviations of uncertainty on either side of the mean prediction of the infinte network.

We will now consider a network with smaller widths of its hidden layers and see if the infinite network provides a good approximation of it. Consider a network defined exactly the same as above but with hidden layers of width 50 rather than 512. Figure 2 shows that despite the network getting similar loss function as the wider network after training, the path that it takes to get there is quite different as shown by the evolution of the loss function through training. As opposed to the larger width case where the losses of the infinite and finite network followed a similar shape, here they evolve quite differently to each other but end up being quite close by the end of training. This would suggest that infinite networks are less good at modelling smaller width networks which our theory would also agree with.

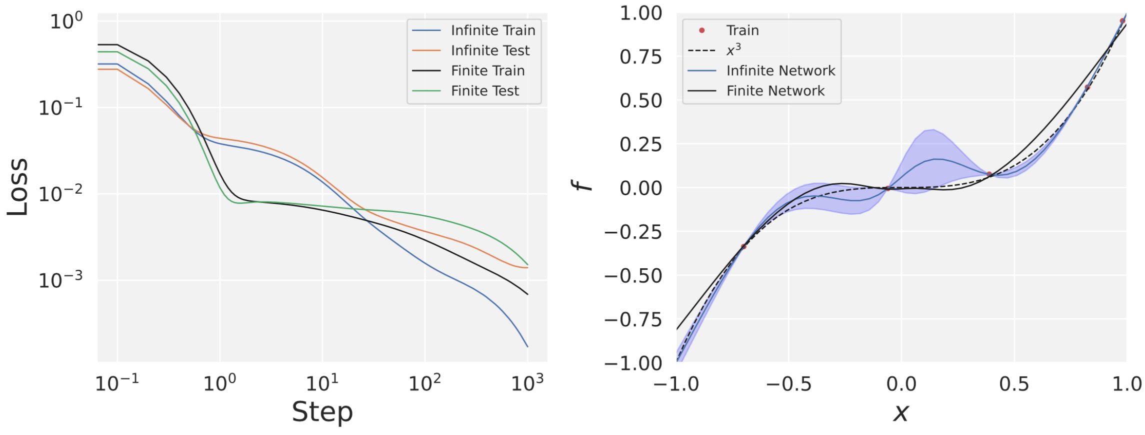

Figure 2: This figure compares the training dynamics of a finite neural network with an infinite width neural network with all parameters of the networks as described above. The graph on the left shows the evolution of the mean squared loss of the finite network and mean squared loss of the mean of the infinite network throughout training. This is shown on the training and the test data. The graph on the right compares the prediction of the infinite width network at infinite time and the finite network after 10,000 steps of stochastic gradient descent. The shaded region denotes two standard deviations of uncertainty on either side of the mean prediction of the infinte network.

Notice how in figure 1 and 2 we see that the infinite network has an uncertainty surrounding the mean. This is because the prediction on the infinite network for each data point is given by a Gaussian distribution. We can gain a similar uncertainty for our finite networks by using an ensemble of neural networks as is often done in real world applications. What I mean is, we program an ensemble of finite neural networks which are each initialised differently to be trained on the same task then average their predictions. We can also use the variance in these predictions to estimate uncertainty in the prediction [12].

6.2 Ensembles of Finite Width Networks Versus Infinite Width Networks

Here, we train 100 neural networks to approximate the function $f(x) = x^3$. We again use 3 layer feedforward fully connected networks with erf activation function and the same parameters for σ_w and σ_b as before. This time we will set the hidden layer widths to 512.

The first thing to note in figure 3 is the similarity of the predictions between the finite and infinite neural networks. The single networks we trained before offered a similar approximation to the infinite neural network but the approximation given by the ensemble is almost identical to that of the infinite network.

The second thing to note is the region bounded between the black dashed lines and how similar it is to the shaded blue region. Both of these regions represent two standard deviations of uncertainty on either side of the mean of their prediction. Not only do they both give similar shaped regions, but also they are of similar size.

Hence we see empirically that an infinite network can provide an excellent approximation to an ensemble of finite networks trained via stochastic gradient descent.

6.3 Convergence of the Neural Tangent Kernel

As we know, in the infinite width limit the neural tangent kernel converges to an explicit deterministic kernel at initialisation. Not only this, but the neural tangent kernel will also stay constant throughout training in the infinite width limit (Section 4.1). In Theorem 4.1 we then went on to bound the convergence of the neural tangent kernel to this deterministic kernel. Now we will investigate the convergence of the neural tangent kernel empirically using an experiment and figure from the paper Neural Tangent Kernel: Convergence and Generalization in Neural Networks [8].

In this experiment Jacot uses networks of depth $L = 4$ and sets all the widths of the hidden layers to be either $n = 500$ or $n = 10000$. To illustrate the distribution of the neural tangent kernel the network is initialised 10 times for each width and the neural tangent kernel $\hat{\Theta}(x_0, x)$ is plotted with $x_0 = (1, 0)$ and $x = (\cos(\gamma), \sin(\gamma))$. To get a picture of how the neural tangent kernel evolves with time Jacot plots the kernels at initialisation and then after 200 steps of gradient descent with learning rate

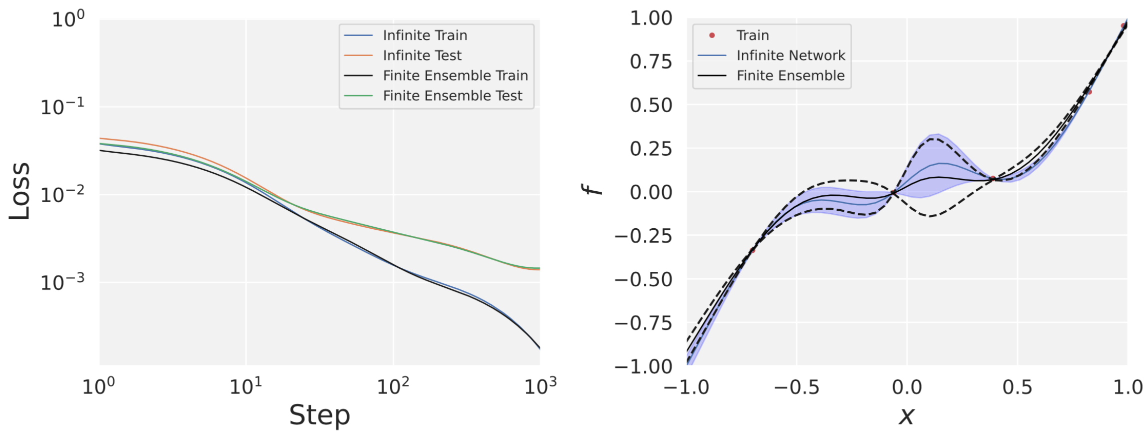

Figure 3: This figure compares the training dynamics of an ensemble of finite neural networks with an infinite width neural network with all parameters of the networks as described above. The graph on the left shows the evolution of the mean squared loss of the average of the ensemble of finite networks and mean squared loss of the mean of the infinite network throughout training. This is shown on the training and the test data. The graph on the right compares the prediction of the infinite width network at infinite time and the prediction of the ensemble where each finite network is trained with 10,000 steps of stochastic gradient descent. The shaded region denotes two standard deviations of uncertainty on either side of the mean prediction of the infinte network and the region between the black dashed lines denotes two standard deviations of uncertainty on either side of the mean prediction of the ensemble of networks.

1.0. Jacot uses the mean squared loss function and trains the network to approximate $f^*(x) = x_1 x_2$ using inputs from $\mathcal{N}(0, 1)$.

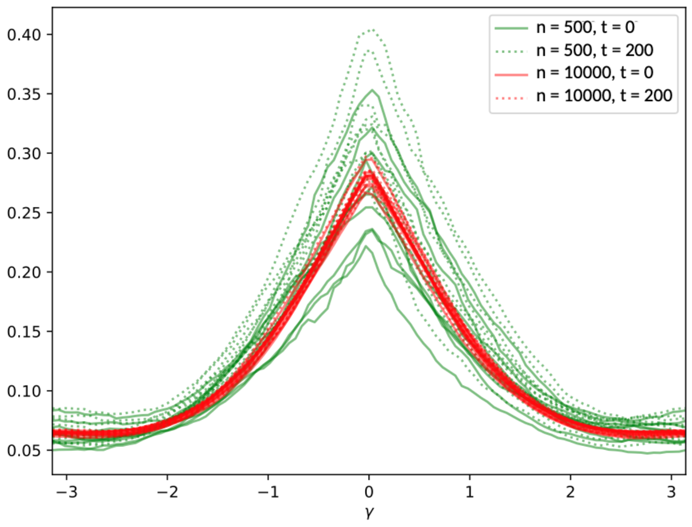

Figure 4: This figure shows the neural tangent kernel and how it varies with respect to time and the network's width as described above. It is taken from the paper Neural Tangent Kernel: Convergence and Generalization in Neural Networks [8].

The first important feature we see is that the smaller width network varies much more at initialisation, the second is that it also varies much more through time. Meanwhile the large width network appears almost fixed. This backs up our theory that the neural tangent kernel converges to a deterministic kernel as with widths of the hidden layers tend to infinity.

7 Conclusion

In this paper we started by analysing finite width fully connected feedforward networks, how their parameters change and in what circumstances they behave like their linearisation. We then moved on to the training of finite width networks by gradient flow. Here is where we first got introduced to the neural tangent kernel as an aide to describe the evolution of neural networks during training. We then went on to linearise the finite network giving us a closed form solution for f^{lin} at any time during training when using the mean squared loss. All that is required is that we know $\hat{\Theta}_0$ and $f(x; \theta_0)$.

By letting the width of the hidden layers tend to infinity we uncovered two fundamental features of the neural tangent kernel in this limit: that it is determined at initialisation and that it stays constant thoughout training. Furthermore, for certain loss functions, we can calculate it. This enables us to describe the training of an infinite width neural network precisely in certain situations. First, we analysed an infinite width network with mean squared loss and showed how in doing so we get a kernel regression problem. Then we returned to the linearised network and saw that the linearised network function converges in distribution in the infinite width limit when viewing the parameters as random variables. We found that for each test point $x \in \mathbb{R}^{n_0}$, $f^{lin}(x;\theta(t))$ converges in distribution to a Gaussian with a formula derived for the mean and covariance. We then went on to compare the prediction of the linearised model to the actual nonlinear network in the infinite width limit and showed that they converge to each other with probability arbitrarily close to 1 over random initialisations provided that the learning rate is small enough. Now we can make predictions of the output of an infinite neural network without even having to initialise it.

We backed up these results empirically using the Neural Tangents package to train infinite and finite networks and compared the results. Finally, we used experiments to show how the neural tangent kernel changes during training and how it changes with the width of a neural network.

References

[1] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. 5, 2019.

[2] Sanjeev Arora, Simon S. Du, Wei Hu, Zhiyuan Li, Ruslan Salakhutdinov, and Ruosong Wang. On exact computation with an infinitely wide neural net. *Advances in Neural Information Processing Systems*, 2:1–13, 2019.

[3] Lenaic Chizat, Edouard Oyallon, and Francis Bach. On lazy training in differentiable programming. 5, 2020.

[4] Simon Du and Wei Hu. Ultra-wide deep nets and the neural tangent kernel (ntk). `https://blog.ml.cmu.edu/2019/10/03/ultra-wide-deep-nets-and-the-neural-tangent-kernel-ntk/`, October 2019. Accessed on 3rd January 2022.

[5] Simon S. Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. *International Conference on Learning Representations*, 2:1–11, 2019.

[6] Rajat Vadiraj Dwaraknath. Understanding the neural tangent kernel. `https://rajatvd.github.io/NTK/`, November 2019. Accessed on 3rd January 2022.

[7] Ferenc Huszár. Some intuition on the neural tangent kernel. `https://www.inference.vc/neural-tangent-kernels-some-intuition-for-kernel-gradient-descent/`, November 2020. Accessed on 3rd January 2022.

[8] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *In Advances in neural information processing systems*, 1:1–8, 2018.

[9] Jaehoon Lee, Lechao Xiao, Samuel S Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. *Journal of Statistical Mechanics: Theory and Experiment*, 4:1–9, Dec 2019.

[10] Roman Novak. Neural tangents. `https://github.com/google/neural-tangents`, November 2021. Accessed on 15th January 2022, version 0.3.9.

[11] Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A. Alemi, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. Neural tangents: Fast and easy infinite neural networks in python. 1, 2019.

[12] Samuel S. Schoenholz and Roman Novak. Fast and easy infinitely wide networks with neural tangents. `https://ai.googleblog.com/2020/03/fast-and-easy-infinitely-wide-networks.html`, March 2020. Accessed on 3rd January 2022.

[13] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *In Proceedings of the International Conference on Learning Representations (ICLR)*, 2, 2017.