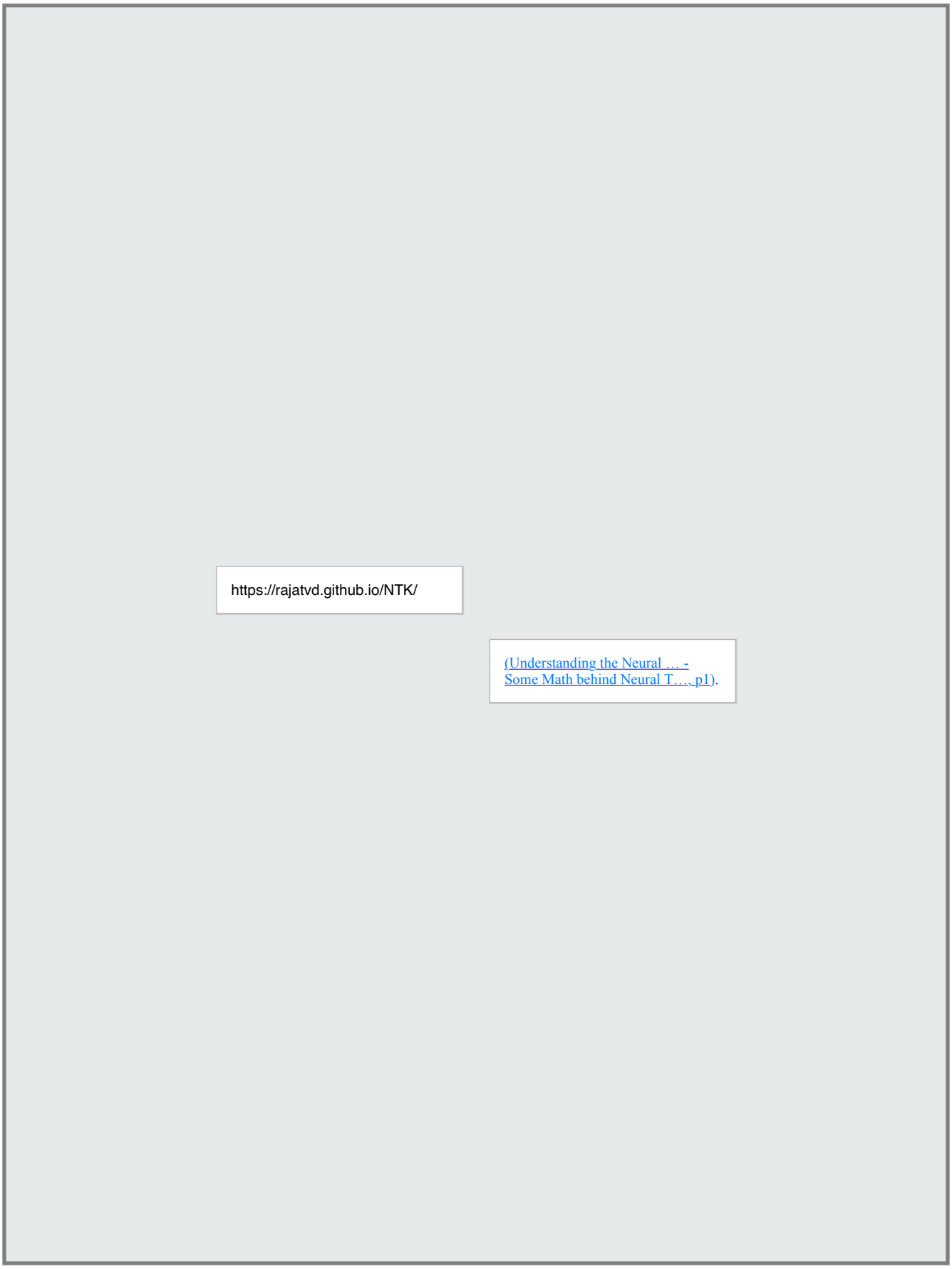
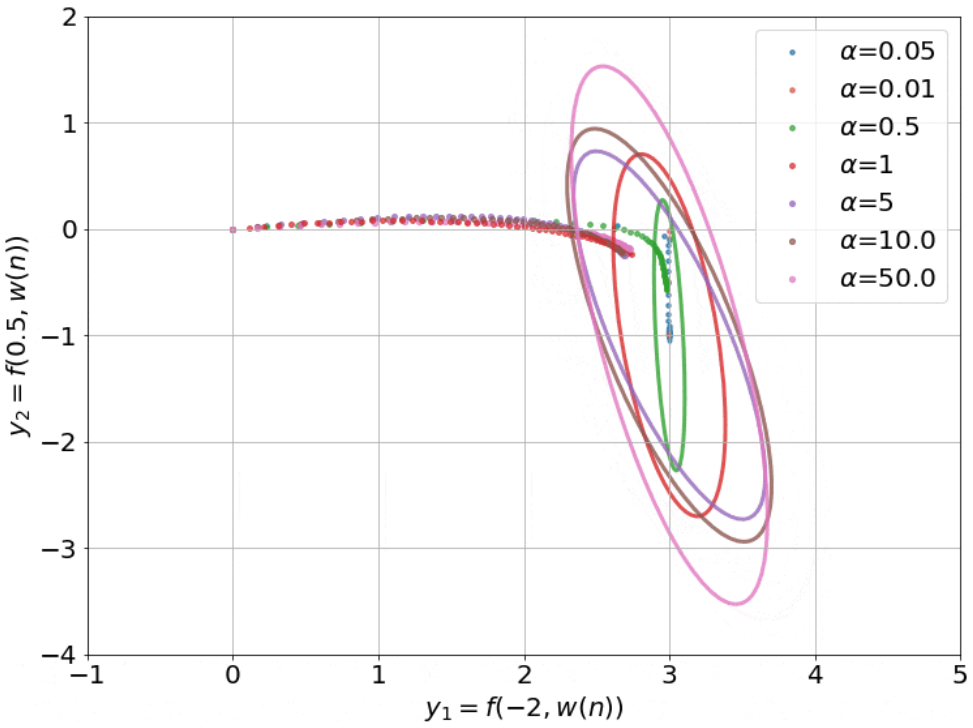

https://rajatvd.github.io/NTK/
(Understanding the Neural … - Some Math behind Neural T…, p1).

Understanding the Neural Tangent Kernel



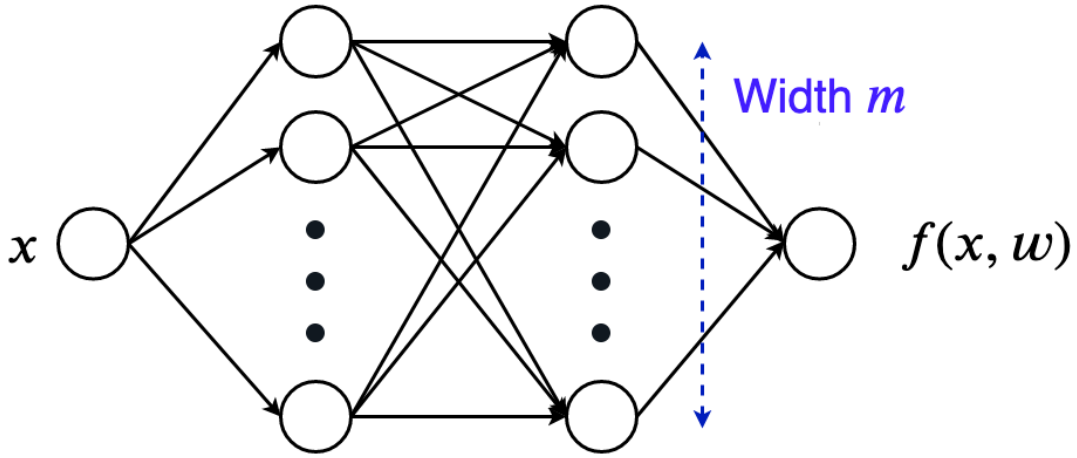
This gif depicts the training dynamics of a neural network. Find out how by reading the rest of this post.

A flurry of recent papers in theoretical deep learning tackles the common theme of analyzing neural networks in the **infinite-width** limit. At first, this limit may seem impractical and even pointless to study. However, it turns out that neural networks in this regime simplify to **linear models** with a kernel called the **neural tangent kernel**. Gradient descent is therefore very simple to study. While this may seem promising at first, empirical results show that neural networks in this regime perform worse than practical over-parameterized networks. Nevertheless, this still provides theoretical insight into some aspects of neural network training, so it is worth studying.

Additionally, this kernel regime can occur under a more general criterion which depends on the **scale** of the model, and does not require infinite width per se. In this post, I'll present a simple and intuitive introduction to this theory that leads to a **proof of convergence of gradient descent** to 0 training loss. I'll make use of a simple 1-d example which lends itself to neat visualizations to help make the ideas easier to understand.

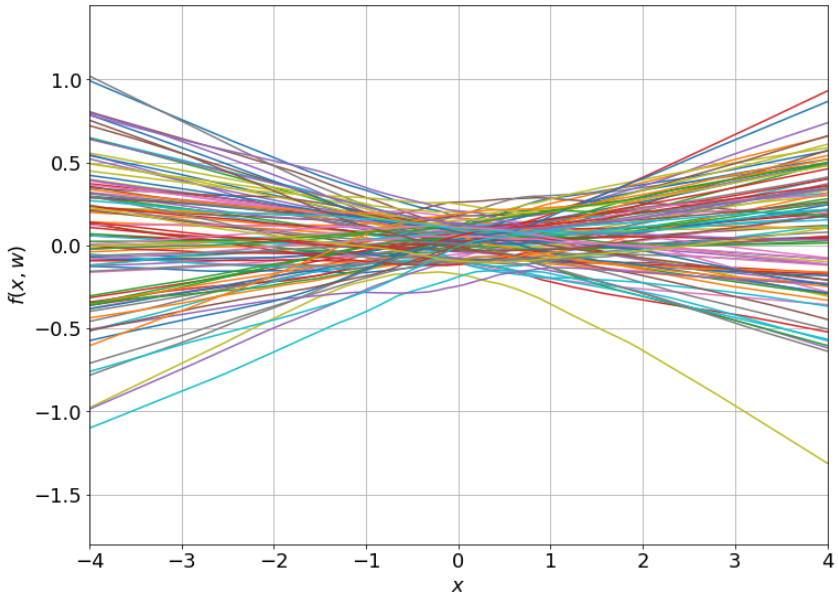
Setup

Let's start with an extremely simple example, and generalize our understanding once we get comfortable with this toy example. Let's work with a 1-D input and 1-D output network. A simple 2-hidden layer relu network with width m will do the trick for now.



The simple 1-D model we will use for initial experiments.

We can nicely plot how the network function looks for a bunch of random initializations:



100 randomly initialized 2-hidden layer, width-100 relu networks.

It turns out that these functions are equivalent to samples drawn from a Gaussian process as you take the hidden layer width to infinity, but that is a whole other can of worms best left for another post.

Lilian's blog explains this well.

Let's bring in some notation:

- Call the neural network function $f(x, \boldsymbol{w})$ where x is the input and $\boldsymbol{w}$ is the combined vector of weights **(say of size p)**.
- In this 1-D example, our dataset will just be points (x, y). Let's say we have N of them, so our dataset is: $\{\bar{x}_i, \bar{y}_i\}_{i=1}^N$.

For learning the network, we'll take a simple approach again: just **perform full-batch gradient descent on the least squares loss**. Now, you might be familiar with writing this loss as:

$$L(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} (f(\bar{x}_i, \boldsymbol{w}) - \bar{y}_i)^2$$

But we can simplify this using some vector notation.

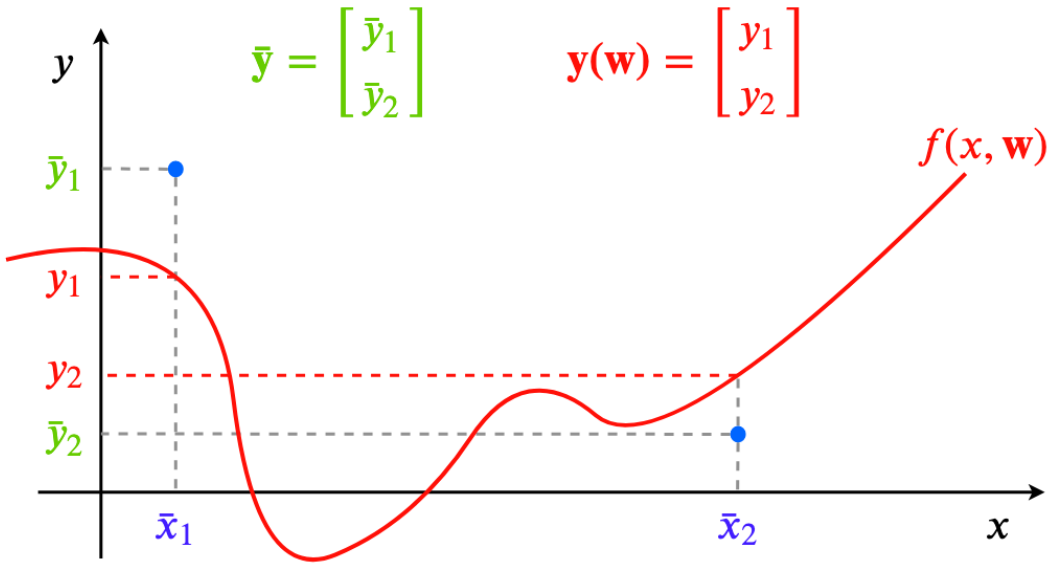
- First, stack all the output dataset values $\bar{y}_i$ into a single vector of size N, and call it $\bar{\boldsymbol{y}}$.
- Similarly, stack all the model outputs for each input, $f(\bar{x}_i, \boldsymbol{w})$ into a single **prediction vector $\boldsymbol{y}(\boldsymbol{w}) \in \mathbb{R}^N$**. We basically have $\boldsymbol{y}(\boldsymbol{w})_i = f(\bar{x}_i, \boldsymbol{w})$. This is similar in flavor to looking at the neural network function $f(\cdot, \boldsymbol{w})$ as a single vector belonging to a **function space**.

- So, our loss simplifies to this:

$$L(\boldsymbol{w}) = \frac{1}{N} \cdot \frac{1}{2} \|\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}\|_2^2$$

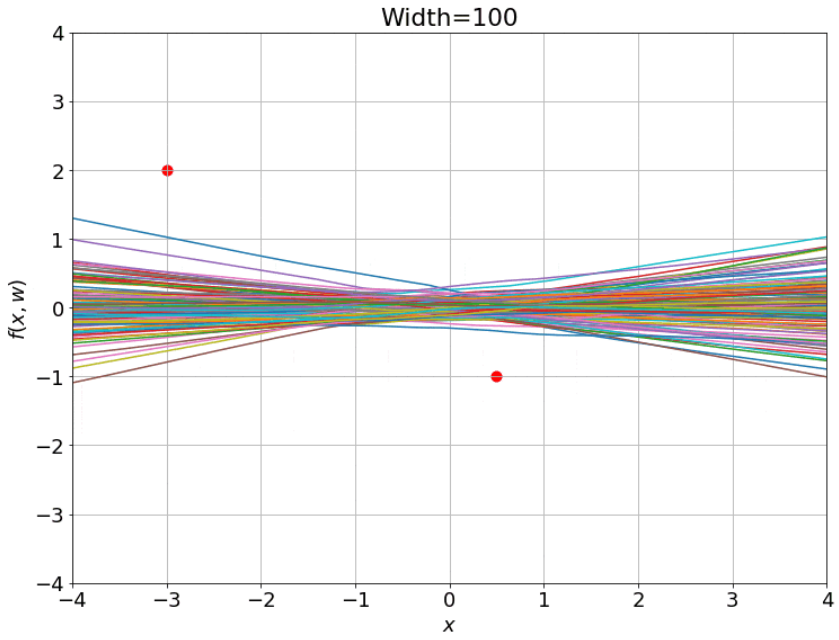
Now, we won't be changing the dataset size N anywhere, and it's an unnecessary constant in the loss expression. So, we can just drop it without affecting any of the further results, while making the algebra look less cluttered (we'll keep the half because it will make the derivatives nicer).

$$L(\boldsymbol{w}) = \frac{1}{2} \|\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}\|_2^2$$



This should clarify the vector notation. We are effectively vectorizing everything over the whole dataset, which is of size 2 in this simple case (the two blue dots).

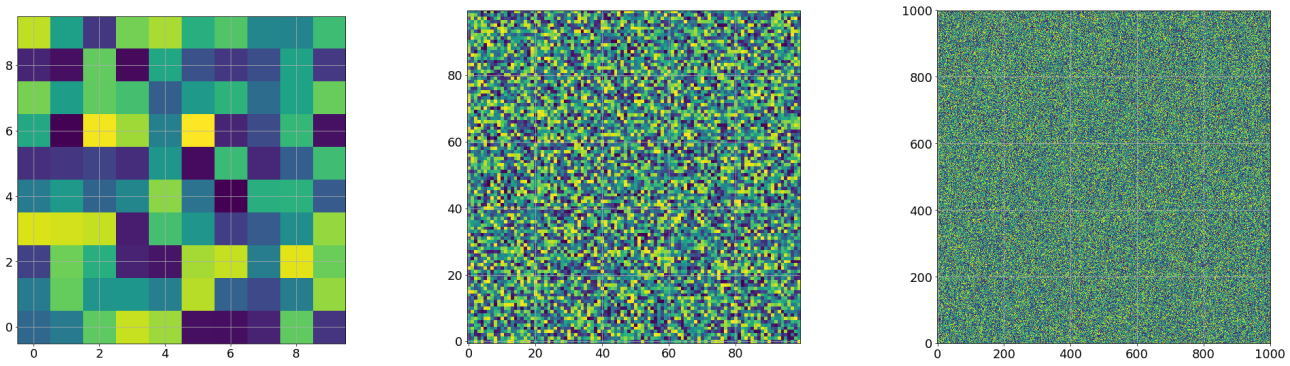
Training the network is now just minimizing this loss using gradient descent.



GIF in original blog

Training 100 ReLU nets using gradient descent on square loss.

If we make a gif of the weights between the two hidden layers (an $m \times m$ matrix) as the training progresses, we observe something very intriguing:



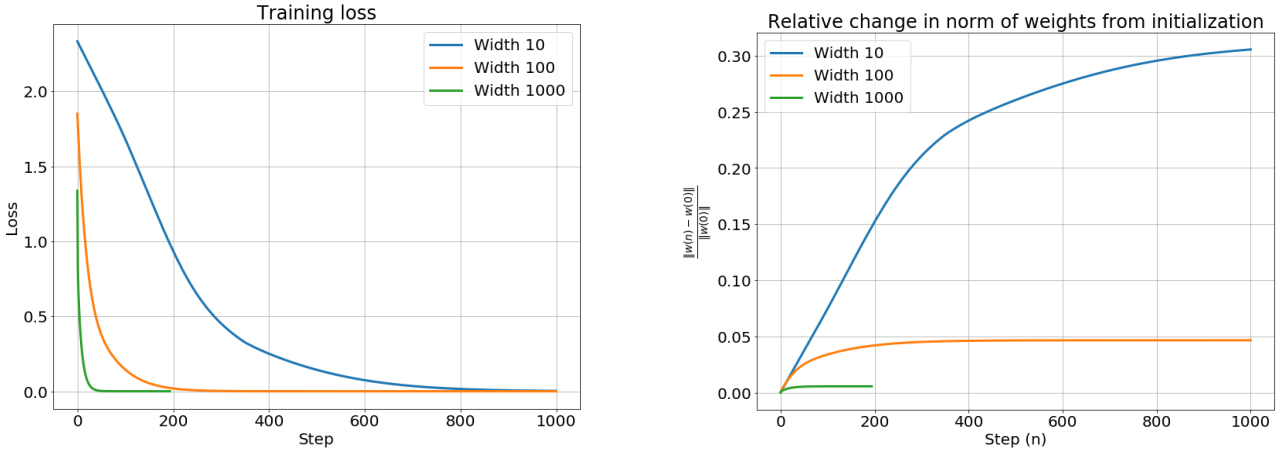
Animation of weight matrices during training. You might be able to see some changes only for width 10, but greater widths look completely static.

Yes, these are actually animated gifs, but the **weights don't even budge** for larger widths! It seems that these models are quite *lazy*! We can see this quantitatively, by looking at the *relative change* in the norm of the weight vector from initialization:

Lazy Training also mentioned in Lilian's blog

$$\frac{\|\boldsymbol{w}(n) - \boldsymbol{w}_0\|_2}{\|\boldsymbol{w}_0\|_2}$$

If the weights double, this relative change will be 2 irrespective of the size of the hidden layers. Now if we plot this for the nets we trained above:



Loss curves and how much the weights budge as training progress. The weight norms are calculated using ALL parameters in the model stacked into a single vector. Also, all other hyperparameters (like learning rate) are kept constant.

The weights don't change much at all for larger hidden widths. Well, what's a good thing to do if some variable doesn't change much? *Hint: Calc 101*

Just Taylor Expand

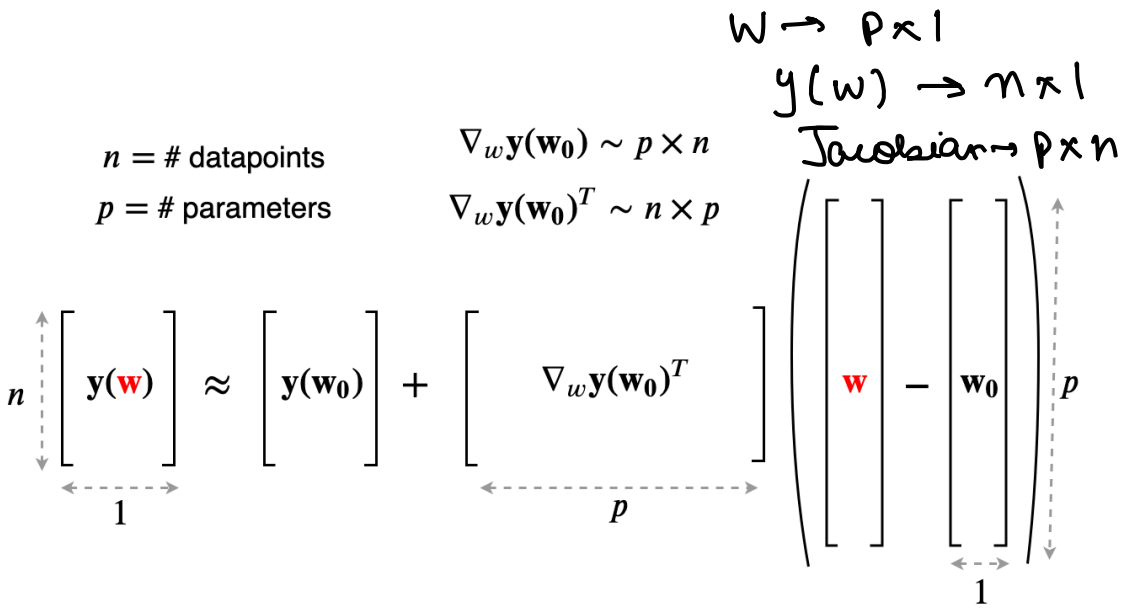
That's right, we can just Taylor expand the network function ***with respect to the weights*** around its ***initialization***.

$$f(x, \boldsymbol{w}) \approx f(x, \boldsymbol{w_0}) + \nabla_{\boldsymbol{w}} f(x, \boldsymbol{w_0})^T (\boldsymbol{w} - \boldsymbol{w_0})$$

We've turned the crazy non-linear network function into a simple **linear** function of the weights. Using our more concise vector notation for the model outputs on a specific dataset we can rewrite as:

$$\boldsymbol{y}(\boldsymbol{w}) \approx y(\boldsymbol{w_0}) + \nabla_{\boldsymbol{w}} \boldsymbol{y}(\boldsymbol{w_0})^T (\boldsymbol{w} - \boldsymbol{w_0})$$

With matrix sizes made explicit:



This picture should make the sizes of all vectors and matrices clear.

A lot of the things here like the initial output $y(\boldsymbol{w_0})$ and the model Jacobian $\nabla_{\boldsymbol{w}} \boldsymbol{y}(\boldsymbol{w_0})$ are just constants. Focus on the dependence on $\boldsymbol{w}$ – the approximation is a **linear model in the weights**, so minimizing the least squares loss reduces to just doing **linear regression**! But, notice that the model function is still **non-linear in the input**, because finding the gradient of the model is definitely not a linear operation. In fact, this is just a linear model using a **feature map** $\boldsymbol{\phi}(x)$ which is the gradient vector at initialization:

$$\boldsymbol{\phi}(x) = \nabla_{\boldsymbol{w}} f(x, \boldsymbol{w_0})$$

Defined in LLW's blog as well

This feature map naturally *induces a kernel* on the input, which is called the **neural tangent kernel**. We'll dive into the details of this kernel later when we look at gradient flows, but first, let's try to theoretically justify this linear approximation.

When is the Approximation Accurate?

The linearized model is great for analysis, only if it's actually an accurate approximation of the non-linear model. Let's try to derive a quantitative criterion for when this approximation works well. This is adapted from Chizat and Bach's paper on Lazy Training in Differentiable Programming (the **lazy regime** is just another name for when this linear approximation holds). You might need to brace yourself, because the derivation I'm presenting is intuitive and involves a bit of handwaving mixed with a salad of matrix and operator norms.

Firstly, we only care about the model being linear as we optimize it using gradient descent:

$$\boldsymbol{w}_1 = \boldsymbol{w}_0 - \eta \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_0)$$

With a small enough learning rate η, we know that the loss always decreases as we run gradient descent. So, the model outputs $\boldsymbol{y}(\boldsymbol{w_n})$ always get closer to the true labels $\bar{\boldsymbol{y}}$. Using this, we can bound the net change as follows:

$$\text{net change in } \boldsymbol{y}(\boldsymbol{w}) \lesssim \|\boldsymbol{y}(\boldsymbol{w_0}) - \bar{\boldsymbol{y}}\|$$

Max. possible change

Now we can quantify the "distance" we move in parameter space with a first order approximation,

$$\text{distance } d \text{ moved in } w \text{ space} \approx \frac{\text{net change in } \boldsymbol{y}(\boldsymbol{w})}{\text{rate of change of } \boldsymbol{y} \text{ w.r.t } \boldsymbol{w}} = \frac{\|\boldsymbol{y}(\boldsymbol{w_0}) - \bar{\boldsymbol{y}}\|}{\|\nabla_{\boldsymbol{w}} \boldsymbol{y}(\boldsymbol{w_0})\|}$$

Also in Lazy Training section of LLW blog

The norm of the Jacobian $\|\nabla_{\boldsymbol{w}} \boldsymbol{y}(\boldsymbol{w_0})\|$ is an operator norm. If you aren't familiar with operator norms, you can just think of it as a measure of the maximum stretching that a matrix can perform on a vector (or even simpler, just think of it as a measure of the 'magnitude' of the matrix).

To get the *change in the Jacobian*, we can use this distance d along with the **Hessian** $\nabla^2_{\boldsymbol{w}} \boldsymbol{y}(\boldsymbol{w_0})$ of the model outputs with respect to the weights. This Hessian is a rank-3 tensor (still a linear operator) in our case, and you can think of it as stacking the Hessian matrices for each output together to get a 3-d Hessian "cuboid". Again, think of its norm as just some measure of the magnitude of the Hessian, or the *rate of change* of the Jacobian.

$$\text{change in model Jacobian} \approx \text{distance } d \times \text{rate of change of the Jacobian} = d \cdot \|\nabla^2_{\boldsymbol{w}} \boldsymbol{y}(\boldsymbol{w_0})\|$$

What we really care about when we want the model to be linear is not just the change in its Jacobian, but the **relative change**. Specifically, we want this relative change to be **as small as possible** ($\ll 1$):

$$\text{relative change in model Jacobian} \approx \frac{d \cdot \text{rate of change of Jacobian}}{\text{norm of Jacobian}} = \frac{d \cdot \|\nabla_w^2 \boldsymbol{y}(\boldsymbol{w_0})\|}{\|\nabla_w \boldsymbol{y}(\boldsymbol{w_0})\|} = \|(\boldsymbol{y}(\boldsymbol{w_0}) - \bar{\boldsymbol{y}})\| \frac{\|\nabla_w^2 \boldsymbol{y}(\boldsymbol{w_0})\|}{\|\nabla_w \boldsymbol{y}(\boldsymbol{w_0})\|^2}$$

Let's call the quantity in blue $\kappa(\boldsymbol{w_0})$. The condition $\kappa(\boldsymbol{w_0}) \ll 1$ can be **intuitively summarized** as follows:

The amount of change in $\boldsymbol{w}$ required to produce a change of $\|(\boldsymbol{y}(\boldsymbol{w_0}) - \bar{\boldsymbol{y}})\|$ in $\boldsymbol{y}$ causes a negligible change in the Jacobian $\nabla_w \boldsymbol{y}(\boldsymbol{w})$.

This means that the model is very **close to its linear approximation**.

We now need to understand how $\kappa(\boldsymbol{w_0})$ changes with the hidden width m of our neural network. It turns out that $\kappa \to 0$ as the width $m \to \infty$. We need to be a bit careful here, because $\kappa(\boldsymbol{w_0})$ is a random variable as the initialization $\boldsymbol{w_0}$ is itself random. What I mean when I say $\kappa \to 0$ is that its *expectation* goes to 0. This only holds if the weights are **properly initialized**. Specifically, they need to be independent zero-mean Gaussian random variables with a variance that goes inversely with the size of the input layer. (This is called *LeCun initialization*, more on this in the next section). This result explains why the weights changed less while training for larger width neural networks.

Understanding why this result is true for the general case is quite complicated, so I'll present a derivation for the simpler case of only **one** hidden layer.

An intuitive explanation for why this happens is as follows: a large width means that there are a lot more neurons affecting the output. A small change in all of these neuron weights can result in a very large change in the output, so the neurons need to move very little to fit the data. If the weights move less, the linear approximation is more accurate. As the width increases, this amount of neuron budging decreases, so the model gets closer to its linear approximation. If you are happy with just believing this intuition, you can skip the proof and go to the next section.

▶ **One Hidden Layer Network proof (click to expand)**

While this is a nifty result for when a neural net behaves linearly in its parameters, it turns out that a much **simpler and general** condition is easy to derive.

Scaling the Output

Let's stare at the expression for $\kappa(\boldsymbol{w_0})$ again:

$$\kappa(\boldsymbol{w_0}) = \|(\boldsymbol{y}(\boldsymbol{w_0}) - \bar{\boldsymbol{y}})\| \frac{\|\nabla_w^2 \boldsymbol{y}(\boldsymbol{w_0})\|}{\|\nabla_w \boldsymbol{y}(\boldsymbol{w_0})\|^2}$$

I'm going to do a very simple thing, just ***multiply the output of our model by some factor*** α.

$$\kappa_\alpha(\boldsymbol{w_0}) = \|(\alpha\boldsymbol{y}(\boldsymbol{w_0}) - \bar{\boldsymbol{y}})\| \frac{\|\nabla_w^2 \alpha\boldsymbol{y}(\boldsymbol{w_0})\|}{\|\nabla_w \alpha\boldsymbol{y}(\boldsymbol{w_0})\|^2}$$

Yup, this is literally just **rescaling the model**. The $\|(\boldsymbol{y}(\boldsymbol{w_0}) - \bar{\boldsymbol{y}})\|$ term is annoying, and we can get rid of it by assuming that our model **always outputs 0 at initialization**, i.e. $\boldsymbol{y}(\boldsymbol{w_0}) = 0$. (We can make any model have this behavior by subtracting a copy of it at initialization.)

$$\implies \kappa_\alpha(\boldsymbol{w_0}) \sim \frac{\|\nabla_w^2 \alpha\boldsymbol{y}(\boldsymbol{w_0})\|}{\|\nabla_w \alpha\boldsymbol{y}(\boldsymbol{w_0})\|^2} = \frac{\alpha\|\nabla_w^2 \boldsymbol{y}(\boldsymbol{w_0})\|}{\alpha^2\|\nabla_w \boldsymbol{y}(\boldsymbol{w_0})\|^2} = \frac{1}{\alpha} \frac{\|\nabla_w^2 \boldsymbol{y}(\boldsymbol{w_0})\|}{\|\nabla_w \boldsymbol{y}(\boldsymbol{w_0})\|^2}$$

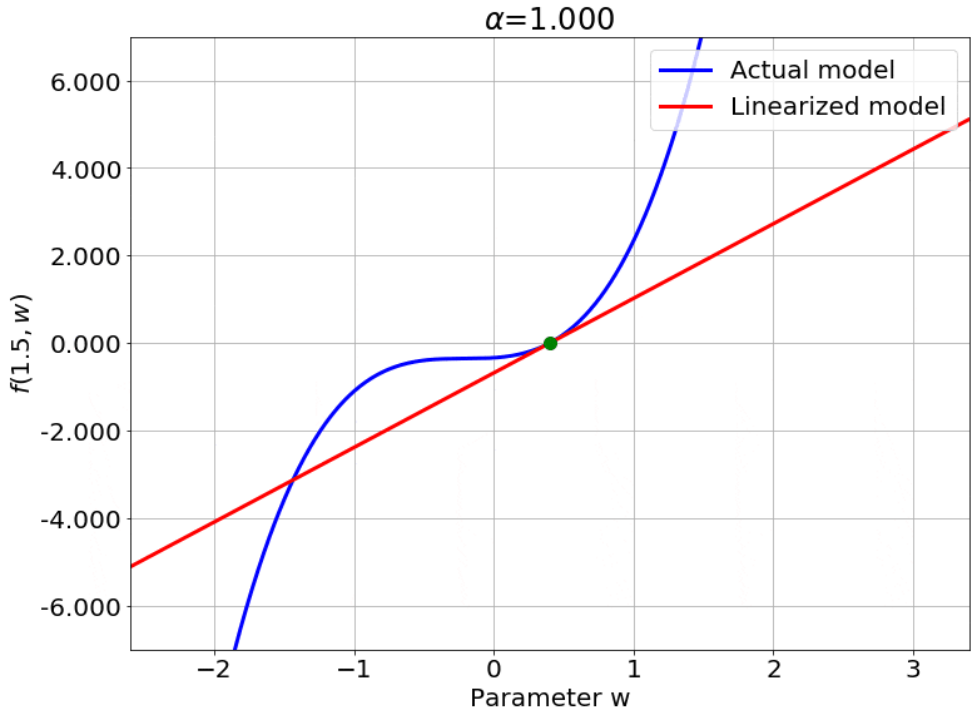
We can make the model linear (take $\kappa(\boldsymbol{w_0}) \to 0$) by simply jacking up $\alpha \to \infty$! Following this, the authors of the paper on lazy training called this quantity $\kappa(w_0)$ the *inverse relative scale* of the model.

One important thing to notice is that this works for **any non-linear model**, and is not specific to neural networks (needs to be twice differentiable though). So, to visualize this, I cooked up a crazy 1-D example model. This model has *one weight* w initialized to $w_0 = 0.4$.

$$f(x, w) = (wx + w^2 x + \sin(e^{0.1w}))(w - 0.4)$$

It also satisfies our requirement that the output be 0 at initialization. To see the effect of α on the linearity of the model, we can just look at the value of the function at *one specific value of* x, say $x = 1.5$. Varying α, we get:

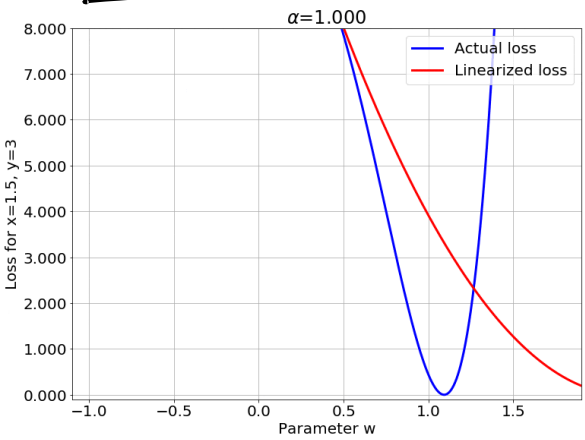
GIF



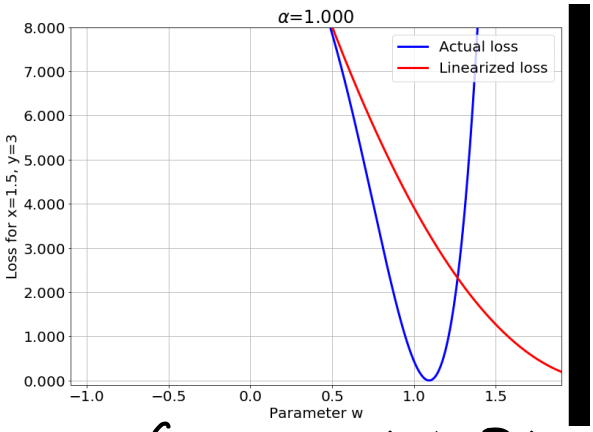
Varying α (the output scaling factor) for our 1-parameter example model evaluated at a specific x. Observe that the linearization (basically the tangent line in this case) approaches the actual function as the scale gets larger.

We can also look at the loss surface for a single data point for the two models. The linearized loss would be a nice parabola, and we would expect the actual loss to get closer to this parabola with increasing α:

GIF



GIF



zooms into 0.4

The *normalized* losses. Normalization is done by simply dividing by α^2 so that the actual loss doesn't change with α. The left and right gifs show the same things, but the right one also zooms in close to $w = 0.4$ so that we can see the small differences more clearly.

A couple of important observations:

- The linearized loss does get closer to the actual loss as we expected.
- The **minima** of the two loss surfaces also get closer, and more importantly, they **get closer to the initialization** as well. This perfectly agrees with our earlier observations that the weights barely budge as we train the model.

Gradient Flow

We've settled the question of when neural networks and more general non-linear models are accurately approximated by their linearizations. Let's now get into their training dynamics under gradient descent.

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \eta \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_k)$$

Rewriting this equation a bit, we get:

$$\frac{\boldsymbol{w}_{k+1}-\boldsymbol{w}_k}{\eta}=-\nabla_{\boldsymbol{w}} L(\boldsymbol{w}_k)$$

The term on the left hand side looks like a *finite-difference* approximation to a derivative. And this equation is like a difference equation approximation to a *differential equation*. If we take the learning rate to be infinitesimally small, we can look at the evolution of the weight vectors over **time**, and write down this differential equation:

$$\frac{d\boldsymbol{w}(t)}{dt}=-\nabla_{\boldsymbol{w}} L(\boldsymbol{w}(t))$$

Add temporal component when η small. LLW blog

This is called a *gradient flow*. In essence, it is a continuous time equivalent of standard gradient descent. The main point is that the trajectory of gradient descent in parameter space closely approximates the trajectory of the solution of this differential equation if the learning rate is small enough. To simplify notation, I will denote time derivatives with a dot: $\frac{d\boldsymbol{w}(t)}{dt}=\dot{\boldsymbol{w}}(t)$. The gradient flow is:

$$\boxed{\dot{\boldsymbol{w}}(t)=-\nabla L(\boldsymbol{w}(t))}$$

Additionally, let's drop the time variable as it can be inferred from context. Substituting for the loss and taking the gradient, we get:

$$\dot{\boldsymbol{w}}=-\nabla \boldsymbol{y}(\boldsymbol{w})(\boldsymbol{y}(\boldsymbol{w})-\bar{\boldsymbol{y}})$$

(time component still there)

We can now derive the dynamics of the model outputs $\boldsymbol{y}(\boldsymbol{w})$ (this is basically the **dynamics in function space**) induced by this gradient flow using the chain rule:

$$\boxed{\dot{\boldsymbol{y}}(\boldsymbol{w})=\nabla \boldsymbol{y}(\boldsymbol{w})^T\dot{\boldsymbol{w}}=-\nabla \boldsymbol{y}(\boldsymbol{w})^T\nabla \boldsymbol{y}(\boldsymbol{w})(\boldsymbol{y}(\boldsymbol{w})-\bar{\boldsymbol{y}})}$$

The quantity in red, $\nabla \boldsymbol{y}(\boldsymbol{w})^T\nabla \boldsymbol{y}(\boldsymbol{w})$ is called the **neural tangent kernel** (NTK for short). Let's give it a symbol $\boldsymbol{H}(\boldsymbol{w})$.

If you go back to the part where we first Taylor expanded the network function, we saw that the linearized model has a feature map $\boldsymbol{\phi}(\boldsymbol{x})=\nabla_{\boldsymbol{w}} f(\boldsymbol{x},\boldsymbol{w_0})$. The kernel matrix corresponding to this feature map is obtained by taking **pairwise inner products** between the feature maps of all the data points. This is exactly $\boldsymbol{H}(\boldsymbol{w_0})$!

$(p\times n)^T \cdot p\times n = n\times n$

$$\underset{n}{\left[\mathbf{H}(\mathbf{w_0})\right]}=\left[\nabla_w \mathbf{y}(\mathbf{w_0})^T\right]\left[\nabla_w \mathbf{y}(\mathbf{w_0})\right]$$

($\mathbf{H}(\mathbf{w_0})$: n by n; $\nabla_w \mathbf{y}(\mathbf{w_0})^T$: n by p; $\nabla_w \mathbf{y}(\mathbf{w_0})$: p by n)

NTK

Can be represented using ϕ which takes weights at initialization.

$$=\begin{bmatrix} — & \phi(\bar{x}_1)^T & — \\ & \vdots & \\ — & \phi(\bar{x}_n)^T & — \end{bmatrix}\begin{bmatrix} | & & | \\ \phi(\bar{x}_1) & \cdots & \phi(\bar{x}_n) \\ | & & | \end{bmatrix}$$

The neural tangent kernel at initialization consists of the pairwise inner products between the feature maps of the data points. Notice that this can also be interpreted as an outer product over data points as well (recall that $\bar{x}_i$ are the inputs in our dataset).

If the model is close to its linear approximation, ($\kappa(\boldsymbol{w_0}) \ll 1$), the Jacobian of the model outputs **does not change as training progresses**. In other words,

$$\nabla \boldsymbol{y}(\boldsymbol{w}(t)) \approx \nabla \boldsymbol{y}(\boldsymbol{w}_0) \implies \boldsymbol{H}(\boldsymbol{w}(t)) \approx \boldsymbol{H}(\boldsymbol{w}_0)$$

When $\kappa(w_0) \ll 1$

This is referred to as the **kernel regime**, because the tangent kernel stays constant during training. The training dynamics now reduces to a very simple **linear ordinary differential equation**:

$$\dot{\boldsymbol{y}}(\boldsymbol{w}) = -\boldsymbol{H}(\boldsymbol{w_0})(\boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}})$$

Clearly, $\boldsymbol{y}(\boldsymbol{w}) = \bar{\boldsymbol{y}}$ is an equilibrium of this ODE, and it corresponds to a train loss of 0 – exactly what we want. We can change state variables to get rid of $\bar{\boldsymbol{y}}$ by defining $\boldsymbol{u} = \boldsymbol{y}(\boldsymbol{w}) - \bar{\boldsymbol{y}}$. The flow simplifies to:

$$\dot{\boldsymbol{u}} = -\boldsymbol{H}(\boldsymbol{w_0})\boldsymbol{u}$$

The solution of this ODE is given by a matrix exponential:

$$\boldsymbol{u}(t) = \boldsymbol{u}(0)e^{-\boldsymbol{H}(\boldsymbol{w_0})t}$$

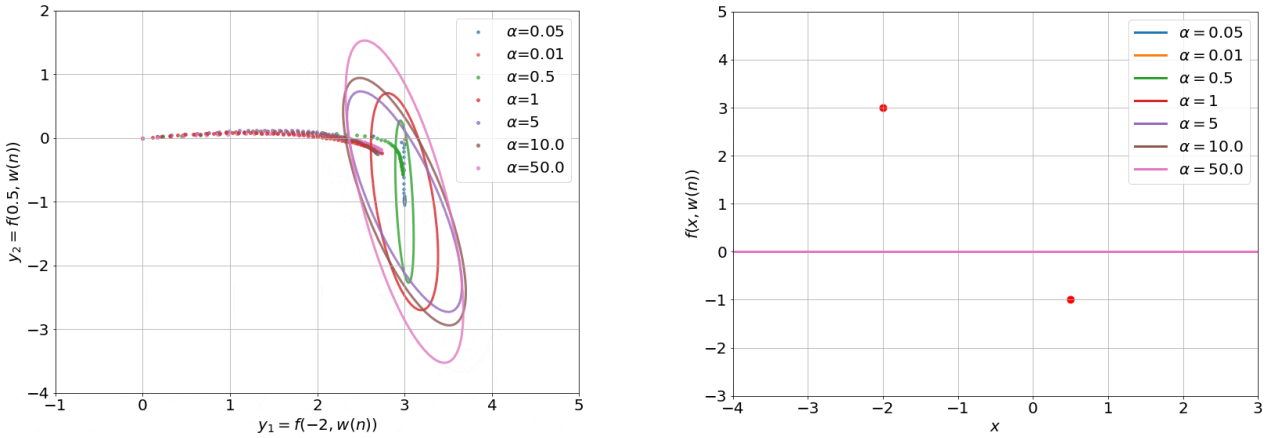
Because our model is **over-parameterized** ($p > n$), the NTK $\nabla \boldsymbol{y}(\boldsymbol{w_0})^T \nabla \boldsymbol{y}(\boldsymbol{w_0})$ is always *positive definite* (ignoring any degeneracy in the dataset that would cause $\nabla \boldsymbol{y}(\boldsymbol{w_0})$ to not have full column rank). By performing a spectral decomposition on the positive definite NTK, we can decouple the trajectory of the gradient flow into independent 1-d components (the eigenvectors) that decay at a rate proportional to the corresponding eigenvalue. The key thing is that they all **decay** (because all eigenvalues are positive), which means that the gradient flow **always converges** to the equilibrium where train loss is 0.

Through this sequence of arguments, we have shown that gradient descent converges to 0 training loss for any non-linear model as long as it is close to its linearization (which can be achieved by just taking the scale $\alpha \to \infty$)! This is the essence of most of the proofs in the recent papers which show that gradient descent achieves 0 train loss.

Seeing the Kernel Regime

The gradient flow math can be a bit overwhelming, so let's try to grab a handle on what's going on by going back to our 2-dataset example. For 2 data points, the NTK is just a 2x2 positive definite matrix. We can visualize such matrices as ellipses in a 2-D plane, where the major and minor axes are the eigenvectors, and their lengths are inversely proportional to the square root of the eigenvalues. The ellipse is basically a contour of the time-varying quadratic $(\boldsymbol{y} - \bar{\boldsymbol{y}})^T \boldsymbol{H}(\boldsymbol{w}(t))(\boldsymbol{y} - \bar{\boldsymbol{y}})$.

On top of this, we can visualize the trajectory of $\boldsymbol{y}(\boldsymbol{w}(t))$ induced by gradient descent (which approximates gradient flow) on this 2-D plane as well. This trajectory is equivalent to performing gradient descent on the same time-varying or *instantaneous* quadratic. This is how it looks:



Visualizing the neural tangent kernel as training progresses for different model scales (starting from different initializations). I've normalized the gradient flow for different α's so that the time-scales and the scale of the tangent kernels match. We can clearly see that larger α (pink) leads to a more constant NTK, while the NTK for smaller α (red) changes significantly as the model trains. On the right is the model functions as the training progresses.

- I intentionally centered the ellipses on the target data $\bar{\boldsymbol{y}}$ so that the trajectories would approach the center. This way, you can see that the component corresponding to the shorter axis (larger eigenvalue) of the ellipse converges faster than the longer axis (smaller eigenvalue).

- The smaller values of α also converge to 0 train loss (this is called the 'deep' or 'rich' regime). We don't have any theoretical justification for this, though. Our proof only works when α is large enough for the ellipse to stay constant throughout training (kernel regime).
- The smallest α's converge so fast it's barely possible to notice in the gif.
- I used a two hidden layer network with a width of 100 for these runs. I also zeroed the outputs of these networks by subtracting a copy at initialization (that is why all the trajectories start from the origin). While we can't visualize how the 10000 or so weights of this network move as we train, we can visualize the networks outputs, and that is what this gif does.

The main takeaway is this: the kernel regime does occur as we expected for large alpha, but the behavior for small alpha escapes a theoretical explanation.

An additional point:

The NTK regime in these plots occur for large α. We could also approach this regime by taking the width to infinity as we showed earlier instead of just scaling the output. However, we get a special bonus in this case. Notice that the NTKs at initialization are different, this is because the initialization is random. But, as we take the hidden widths to infinity in a neural network, it turns out that the kernel at initialization becomes **deterministic**! There is one fixed NTK for a given depth and activation function. In fact, this NTK can actually be computed exactly, effectively simulating an infinite width neural network (see this paper On Exact Computation in an Infinitely Wide Net). This result is very similar to the deterministic gradient norm we saw in the one hidden layer network proof above, but I won't go into further details, because this post is long enough already.

↳ See "Deterministic NTK" section in LLW blog.

Generalization

So far, we only talked about what happens for the training data. But what about the test set, i.e. the points in between our training samples?

Firstly, training the model in the kernel regime is equivalent to solving a linear system. Over-parameterization ($p > n$) just means that this linear system is **under-determined** – so it has infinite solutions. Since we are using gradient descent to solve this system, it is **implicitly biased** towards a solution of **minimum norm**. That is, gradient descent chooses that solution which has minimum $\|\boldsymbol{w}\|_2$ (as long as we start from an initialization with low norm).

If we now think of extending the $\boldsymbol{y}$-space to a **function space** of infinite dimension, then this minimum ℓ_2 norm in w-space translates to choosing a function that minimizes a specific *functional norm*. What determines this norm? The kernel which describes the linear problem does, which in this case is the NTK. This can then be interpreted as a form of regularization, and you can use this to talk about generalization. I won't delve into further details, as this post is mainly aimed at understanding optimization results as opposed to generalization. If you want to learn more about this, I'd suggest looking up stuff on *reproducing kernel Hilbert spaces* and the generalizing properties of kernel machines.

Concluding Remarks

The NTK theory is great, but the above visualizations show that there is more to neural networks than just this kernel regime. Experiments show that practically successful neural networks definitely **do not operate in the kernel regime**. In fact, even the best linearized neural networks (exactly computed NTKs for infinite width) actually **perform worse** on standard benchmarks like MNIST and CIFAR by around ~7% (although recent developments might have bridged this gap, see this paper on Enhanced Convolutional NTKs).

However, this NTK regime theory is not the only theory for understanding neural network training dynamics. There is also a line of work (see references) that uses ideas from optimal transport and mean field theory to characterize the dynamics when the model scaling is not large ($\kappa \sim 1$) through a non-linear PDE, but to my knowledge, these theories don't extend beyond networks with a single hidden layer.

Nevertheless, these NTK results are still extremely interesting, because they offer a new perspective on neural network learning. Trying to understand how the kernel changes as training progresses seems like an important question to answer in the search for a better theory of neural networks.

You can find the code for this post in this repo.

Special thanks to my brother Anjan and my guide Prof. Harish Guruprasad for their insightful comments and suggestions.

References

Related to NTK:

- This nice YouTube video by Arthur Jacot summarizes the NTK paper and also has some more visualizations which I didn't present in this post.
- Jacot, Arthur, Franck Gabriel, and Clément Hongler. "Neural tangent kernel: Convergence and generalization in neural networks." Advances in neural information processing systems. 2018.
- Chizat, Lenaic, and Francis Bach. "A note on lazy training in supervised differentiable programming." arXiv preprint arXiv:1812.07956 (2018).
- Arora, Sanjeev, et al. "On exact computation with an infinitely wide neural net." arXiv preprint arXiv:1904.11955 (2019).
- Li, Zhiyuan, et al. "Enhanced Convolutional Neural Tangent Kernels." arXiv preprint arXiv:1911.00809 (2019).

Mean field dynamics and optimal transport point of view:

- Chizat, Lenaic, and Francis Bach. "On the global convergence of gradient descent for over-parameterized models using optimal transport." Advances in neural information processing systems. 2018.
- Mei, Song, Theodor Misiakiewicz, and Andrea Montanari. "Mean-field theory of two-layers neural networks: dimension-free bounds and kernel limit." arXiv preprint arXiv:1902.06015 (2019).

Written on November 10, 2019

ALSO ON **RAJAT'S BLOG**

Exploring Adversarial Reprogramming
4 years ago • 2 comm…
Google brain recently published a paper titled Adversarial …

Neural Ordinary Differential …
4 years ago • 4 comm…
I experiment with Neural ODEs and touch on parallels between …

Visualizing Tensor Operations with …
3 years ago • 2 comm…
The factor graph is a beautiful tool for visualizating complex …

Animating Doodles with Autoencoders …
4 years ago • 4 comm…
I train autoencoders to identify components of doodles using a synthetic …

Sandesh Ghimire • 3 years ago
I am not clear how NTK theory helps in understanding generalization of neural networks? Or does it? The only relation to generalization seems to be that the kernel gradient is defined for data points outside the training set. Could you point me to a good reference in this direction?
5 | Reply • Share ›

ChaSphere • 2 years ago • edited
It's not clear to me how you draw a connection to linear regression:

> ...training the model in the kernel regime is equivalent to solving a linear system. (...) Since we are using gradient descent to solve this system, it is implicitly biased towards a solution of minimum norm.

I can see how in the kernel regime we're basically computing x_{t+1} = (I - cH)_t, for some constant c. And I can see how this amounts to an iterative method to find a solution to Hu = 0. However, I don't know what you mean when you say it will converge to a solution of minimum norm. If H is positive definite, there's only one solution to Hu = 0 anyway, and that's u = 0.
| Reply • Share ›

Rajat V D Mod ➚ ChaSphere • 2 years ago • edited
You are correct that since H is positive definite, the solution to Hu = 0 is u = 0. However, note that the vector u is in the output space (function space in general). If you were to write Hu = 0 in terms of the weight vector w, you would get a linear equation which does not have a single solution in terms of w. When I said gradient descent is biased towards solutions of minimum norm, I meant minimum **weight** norm (I clarified this in the line after your quote as well).
| Reply • Share ›

ChaSphere ➚ Rajat V D • 2 years ago
I haven't done the calculation for the weights, but what's strange is it seems to me that if H is positive definite, then in function space

I haven't done the calculation for the weights, but what's strange is it seems to me that if H is positive definite, then in *function space*, there should only be one fit for the training data, so that the minimum norm question becomes irrelevant again. Maybe I've made some mistake:

If x is *any* input (not just a training input) then you can show that f(x, t) converges to f(x, 0) + h(x) A u(0), where A is the inverse of H, h(x) is the vector of kernel values k(x, x_i), where x_i is in the training set, and u(0) is the difference between the predicted training labels at initialization and the correct training labels. In short, the function f(x, t), as a function defined on the entire input domain, converges to f(x, 0) + g(x), where g is a member of the RKHS such that f(x_i, t) = y_i for each (x_i, y_i) in the training set. In other words, the kernel regime basically finds the best fit to the training data in an affine space of the RKHS plus the function at initialization.

My interpretation of a statement like "gradient descent is biased towards the minimum functional norm solution" is that we converge to not just to any g in the RKHS, but to the minimum norm g. However, if H is positive definite, there is only one g in the RKHS that fits the training data. So counterintuitively, it seems like in the limit of large width (when H is more likely to be positive definite, and the kernel regime holds), the network has *less* choice as to how it interpolates the training data. Do you see what I mean? Maybe I'm completely barking up the wrong tree or have simply screwed up the math, I'm very new to this area.

Anyway, thanks for the article.

Rajat V D Mod ➚ ChaSphere • 2 years ago

The statements you wrote are all indeed mathematically correct, I think you are just missing a small technicality about the norm of the initialization itself.

> ... then in function space, there should only be one fit for the training data ...

Notice that this fit is found by gradient descent and depends on the initialization. For the optimization problem "find a function which interpolates the data", we don't prescribe that this must be done by specifically by gradient descent or any other means. This particular problem has infinite solutions when you are in the overparameterized setting. However, if you choose to solve the problem by running gradient descent from some particular initialization, you will converge to one fixed solution as you correctly pointed out.

> In other words, the kernel regime basically finds the best fit to the training data in an affine space of the RKHS plus the function at initialization.

You can think of the above statement in terms of the norm of the solution as well:

The particular solution gradient descent converges to is the "closest solution" to the initialization.

To go from here to the actual minimum norm solution, you need to impose an additional condition that your initialization itself have 0 norm (i.e. init = 0) -- so now the closest solution will naturally be the one of minimum norm. Alternatively, you can also choose an initialization with a very small norm, and you will reach a solution that is very close to the minimum norm solution. This is what I meant by "gradient descent is biased towards the minimum norm solution". However, you are correct that this business of minimum norm solution will not hold without this technicality.

ChaSphere ➚ Rajat V D • 2 years ago • edited

I think I see, so you're saying there are multiple fits to the training data *in weight space*, i.e. in the space of of all functions representable by the network. However, in the kernel regime the network can only actually reach a thin cross-section of that space: the plane parallel to the RKHS passing through the initial function. And the RKHS itself depends on the tangent kernel and therefore on the initial weights. If we identify weight vectors and functions, then we can write this "feasible space" as w0 + RKHS(w0). And you claim that the (unique) element of w0 + RKHS(w0) that fits the data is also the closest point (in Euclidean norm) to w0 in the entire space that fits the data, is that right?

Rajat V D Mod ➚ ChaSphere • 2 years ago

Yeah exactly, you got it.

A couple additions:

> And the RKHS itself depends on the tangent kernel and therefore on the initial weights.

In the specific case of neural networks, if you choose your initial weights to be Gaussian random variables with the right scaling (this is the initialization that is commonly used by neural networks in practical cases), the tangent kernel actually becomes a deterministic constant in the limit of infinite width, so the dependence on the initial weight also disappears. This only applies for the specific case of neural networks, but the statement you wrote holds in general.

> the (unique) element of w0 + RKHS(w0) that fits the data is also the closest point (in Euclidean norm) to w0 in the entire space that fits the data

Yup, this is correct. If you want to see why this is true, you can look at the problem of solving an overparameterized linear equation Ax = b using gradient descent. The solution space can be represented as x* + nullspace(A) where x* is any particular solution. If you minimize the error term norm(Ax - b)^2 over x using gradient descent, you can find with a bit of math that every gradient update lies in the rowspace of A. Since the rowspace is orthogonal to the nullspace, you can then infer that every gradient update is orthogonal to the solution space. Now it should be easy to intuitively see that gradient descent will reach the closest solution to the initialization. In the current setting, we are essentially doing exactly the same thing because our model is linear in the weights (rewrite Hu = 0 in terms of w if this isn't clear). You can then translate these arguments to the RKHS as well.

Reply • Share ›

ChaSphere ➚ Rajat V D • 2 years ago

I see now, thank you, and thank you for your time answering all my questions. One thing I'm still wondering about if you don't mind: when you write

> If we now think of extending the y-space to a function space of infinite dimension, then this minimum ℓ2 norm in w-space translates to choosing a function that minimizes a specific functional norm. What determines this norm? The kernel which describes the linear problem does, which in this case is the NTK.

Which functional norm are you referring to? Once we converge to the unique solution wT in the space w0 + RKHS(w0), my understanding of this passage is that you mean there is a functional norm on *the entire weight space* (after identifying weight vectors with functions), and that wT is the closest point to w0, in the sense of that functional norm, that fits the data. You just explained how the same thing is true if "functional norm" is replaced with "l2 norm in weight space", however I don't see what functional norm would make it true. It can't be the RKHS norm on RKHS(w0), since that can't be extended to the entire weight space, or at least not in a way that's obvious to me. Is it just that the l2 norm on the weights can be interpreted as a functional norm? If so, does this "interpretation" have a name?

Reply • Share ›

raghav gupta • 3 years ago

Great post

Reply • Share ›

Rémi Le Priol • 3 years ago

I really enjoyed reading your post. I got a sense of easy understanding, especially thanks to your illustrations. Which library do you use to make these gifs ?

Reply • Share ›

Rajat V D Mod ➚ Rémi Le Priol • 3 years ago

Hi,

I used matplotlib in python to make the gifs. You can find the code in this repo: https://github.com/rajatvd/NTK

Reply • Share ›

Calimero • 3 years ago

Thank you so much for taking the time to write this great post! I've been trying to understand NTK and the lazy regime for a while now and this really helps a lot.