Toy Model II: Kernel Regression

Mark Rhee

Chapter 3

In the previous chapter we studied deep linear networks, i.e., networks that are a linear function of their input data. What made the analysis interesting was their nonlinear training dynamics. Such nonlinear training dynamics are a result of a subtlety in the word "*linear*" in deep linear network. It's true that the network is linear with respect to each individual weight matrix W_i. However, if we consider the weights collectively, stacking all the scalar weights in one ginormous *parameter vector* θ, then a deep linear network is decidedly *not* linear in θ. In fact, a deep linear network with L layers is an Lth order polynomial of θ. This is essentially what results in coupled dynamics and why we had to solve nonlinear ODEs to obtain the optimization trajectories of the weights. This is also what made deep linear networks a decent toy model of neural network training dynamics.

In this chapter, instead of networks that are a linear function of their input data, we will study networks that are approximately a linear[1] function of their parameter vector θ. A good way to think about such networks is that their weights θ deviate so little from initialization throughout training that the network as a whole is well approximated by its first-order (linear) Taylor expansion centered at its weights at initialization $\theta^{(0)}$:

$$\hat{f}^{(t)}(x) \approx \hat{f}^{(t)}_{\text{lin}}(t) := \hat{f}^{(0)}(x) + \left(\nabla_\theta \hat{f}(x)|_{\theta=\theta^{(0)}}\right)^\top (\theta^{(t)} - \theta^{(0)}) \tag{1}$$

$$= \text{const.} + (\text{const.})^\top (\text{change in weights}) \tag{2}$$

for all $t \geq 0$. Crucially, the terms $\hat{f}^{(0)}(x)$ and $\nabla_\theta \hat{f}(x)|_{\theta=\theta^{(0)}}$ are determined at initialization and stay constant throughout training. The weights of this kind of network undergo very simple, exactly solvable optimization trajectories. For neural networks in the infinite-width limit, there exist hyperparameter settings that result in this kind of linear training.[2] Because the weights undergoing this kind of training barely move, we say that such hyperparameter settings put the network in the *lazy* regime. It's also worth noting that a recent paper from Meterez et al. (2026)[3] found that for LLMs, such a first-order Taylor expansion well-approximates the full model for meaningful windows of time during pretraining.

1 Lazy training in the infinite-width limit

When dealing with coupled systems that have many moving parts, physicsts love to take some parameter(s) to infinity and see if things simplify. A classic example of this techinque is the thermodynamic limit. The key idea is that macroscopic behavior becomes stable and easier to analyze when micro-level fluctuations vanish. Might something like that be possible for neural networks?

[1] Technically affine.

[2] There are also hyperparameter settings that result in non-lazy training in the infinite-width limit. More on that in the next chapter.

[3] A Defense of the Quadratic Model, https://arxiv.org/pdf/2607.21716

Consider a 2-layer neural network:

$$\hat{f}(x) = W_2\sigma(W_1 x). \tag{3}$$

We can rewrite this in a suggestable way as follows:

$$\hat{f}(x) = \sum_{i=1}^{H} u_i \sigma(w_i^\top x) \tag{4}$$

where the $u_i \in \mathbb{R}^{n_{\text{out}}}$ are columns of $W_2 \in \mathbb{R}^{n_{\text{out}} \times H}$ and the $w_i \in \mathbb{R}^{n_{\text{in}}}$ are rows of $W_1 \in \mathbb{R}^{H \times n_{\text{in}}}$. Written this way, it's clear that a 2-layer neural network is just an input-dependent linear combination of output vectors u_i. Furthermore, at initialization, the pairs (u_i, w_i) are drawn independently across i from some fixed distribution. This means that for any fixed input x, the H summands $u_i\sigma(w_i^\top x) =: X_i$ are themselves i.i.d. random vectors in $\mathbb{R}^{\text{out}}$. In other words,

$$\hat{f}(x) = \sum_{i=1}^{H} X_i \tag{5}$$

is precisely a sum of H i.i.d. random vectors at initialization. This is exactly the kind of object the central limit theorem (CLT) governs.

To cleanly make a CLT-style concentration argument, we need to send the number of summands to infinity, i.e., $H \to \infty$. However, there's a problem with absent-mindedly taking this infinite-width limit: the value of the function $\hat{f}(x)$ will also go to infinity! One way to prevent $\hat{f}(x)$ from blowing up is to scale the output inversely proportional to the size of the width H. Such output scaling allows us to initialize weights as standard Gaussians ($u_{ij} \sim \mathcal{N}(0,1)$ and $w_{ij} \sim \mathcal{N}(0,1)$) and use a width-independent learning rate η. By and large, there are two classical ways of scaling a sum of i.i.d. random variables: a $\frac{1}{H}$ scaling and a $\frac{1}{\sqrt{H}}$ scaling. The former is associated with the law of large numbers (LLN) while the latter is associated with the central limit theorem (CLT). The latter is what leads to lazy training, so we'll study that first:

$$\hat{f}^{(t)}(x) = \frac{1}{\sqrt{H}} \sum_{i=1}^{H} u_i^{(t)} \sigma({w^{(t)}}_i^\top x), \tag{6}$$

where $u_{ij}^{(0)} \sim \mathcal{N}(0,1)$ and $w_{ij}^{(0)} \sim \mathcal{N}(0,1)$ and the learning rate η is some small width-independent number (e.g., 0.1). We'll see that this setup not only allows us to leverage the CLT at initialization, it also results in the weights moving so little from initialization that for large H, the network $\hat{f}^{(t)}(x)$ is well-approximated by its first-order Taylor expansion $\hat{f}_{\text{lin}}^{(t)}(x)$, i.e., it undergoes lazy training.

Concretely, we'll try to understand why the following statement might be true:

$$\sup_{t \geq 0} \left\| \hat{f}^{(t)}(x) - \hat{f}_{\text{lin}}^{(t)}(x) \right\|_2 = \mathcal{O}\left(\frac{1}{\sqrt{H}}\right). \tag{7}$$

In the $H \to \infty$ limit, the above equation implies that the trajectory of $\hat{f}^{(t)}(x)$ is exactly the trajectory of $\hat{f}_{\text{lin}}^{(t)}(x)$. To make sense of the above statement, we will now introduce one of the central objects of study in learning mechanics: the neural tangent kernel (NTK).

1.1 The neural tangent kernel

At its core, the neural tangent kernel is a tool for reasoning about networks undergoing gradient descent in *function space* instead of *weight space*. Recall that weight space is the collection of all possible weight configurations. It can be thought of as a high dimensional vector space $\mathbb{R}^M$ where M is the total number of tunable weights. Every point in this space of weights is mapped to a scalar loss value by the loss function $\mathcal{L} : \mathbb{R}^M \to \mathbb{R}$. These loss values make up the loss landscape. In chapters 1 and 2, we mostly thought about networks' weight space dynamics; i.e., how the weights θ traverse the loss landscape according to gradient flow:

$$\dot{\theta}^{(t)} = -\eta \nabla_\theta \mathcal{L}(\theta^{(t)}). \tag{8}$$

However, at the end of the day, the only reason we care about the dynamics of the weights θ is because they tell us about the dynamics of the network function $\hat{f}(x)$—we want to see how the predictions for each data point $x \in \mathbb{R}^{n_{\text{in}}}$ evolve over time. Could we bypass the middleman θ[4] and directly study the dynamics of the time-evolving prediction matrix $\hat{f}^{(t)}(X) \in \mathbb{R}^{P \times n_{\text{out}}}$ (where $X \in \mathbb{R}^{P \times n_{\text{in}}}$ and P is the number of data points in the training set)?

For the rest of this chapter we take $n_{\text{out}} = 1$ so that $\hat{f}(x) \in \mathbb{R}$ and $\hat{f}(X) \in \mathbb{R}^P$[5]. It will be convenient to name the Jacobian of the network's predictions with respect to its weights,

$$J^{(t)} := \frac{\partial \hat{f}^{(t)}(X)}{\partial \theta} \in \mathbb{R}^{P \times M}, \qquad J^{(t)}_{ik} = \frac{\partial \hat{f}^{(t)}(x_i)}{\partial \theta_k}, \tag{9}$$

whose ith row is the gradient of the network's prediction at x_i with respect to all M weights.

Now apply the chain rule to see how the predictions move:

$$\dot{\hat{f}}^{(t)}(X) = J^{(t)} \dot{\theta}^{(t)} = -\eta J^{(t)} \nabla_\theta \mathcal{L} = -\eta \underbrace{J^{(t)} J^{(t)\top}}_{=:K^{(t)}} \nabla_{\hat{f}(X)} \mathcal{L}\left(\hat{f}^{(t)}(X)\right), \tag{10}$$

The matrix that appeared is the *empirical neural tangent kernel*:

$$K^{(t)} := J^{(t)} J^{(t)\top} \in \mathbb{R}^{P \times P}, \qquad K^{(t)}_{ij} = \sum_{k=1}^{M} \frac{\partial \hat{f}^{(t)}(x_i)}{\partial \theta_k} \frac{\partial \hat{f}^{(t)}(x_j)}{\partial \theta_k}. \tag{11}$$

For now, we'll refer to the empirical neural tangent kernel as simply the NTK. The following are three important observations we can make about the NTK. First, $K^{(t)}$ is symmetric and positive semi-definite. Second, it has no information about the loss or the labels: it is built entirely from the architecture and the current weights, and the loss enters Equation (10) only through the separate factor $\nabla_{\hat{f}(X)} \mathcal{L}$. Third, Equation (10) is exact. We have made no approximation, taken no limit, and assumed nothing about the architecture; every neural network trained by gradient flow obeys it. Gradient flow was used only to get a clean ODE, and the definition of $K^{(t)}$ itself is perfectly sound under discrete gradient descent.

[4] There are pros and cons to studying weight space dynamics. On the one hand, it's the most visceral way to think about what's going on during gradient descent. On the other hand, there's a lot of stuff to keep track of and there exist many symmetries in weight space that create annoying redundancies.

[5] Nothing conceptual changes for $n_{\text{out}} > 1$; the kernel simply becomes an $Nn_{\text{out}} \times Nn_{\text{out}}$ matrix of blocks, one $n_{\text{out}} \times n_{\text{out}}$ block per pair of training points.

1.1.1 Linear regression revisited

Take a hard look at Equation (10). Hopefully it looks familiar. It is almost exactly the differential equation for overparameterized linear regression from chapter 1:

$$\dot{\hat{f}}^{(t)}_{\text{linreg}}(X) = X\dot{w}^{(t)} = -\eta XX^\top \left(Xw^{(t)} - y\right) \tag{12}$$

$$= -\eta XX^\top \nabla_{\hat{f}_{\text{linreg}}(X)} \mathcal{L}\left(\hat{f}^{(t)}_{\text{linreg}}(X)\right). \tag{13}$$

Linear regression is the special case in which the Jacobian is the data matrix itself, $J = X$, so that the kernel is just the matrix of inner products between data points, $K_{ij} = x_i^\top x_j$. Being linear in its weights, linear regression has a Jacobian that does not depend on w at all, which is precisely why its kernel is constant and its dynamics are solvable in closed form.

Define the *feature map*

$$\phi^{(t)}(x) := \frac{\partial \hat{f}^{(t)}(x)}{\partial \theta} \in \mathbb{R}^M, \qquad \text{so that} \qquad K^{(t)}_{ij} = \left\langle \phi^{(t)}(x_i), \phi^{(t)}(x_j) \right\rangle. \tag{14}$$

We can think of the NTK as first projecting each data point into an M-dimensional feature space, then taking inner products there. Thus, a neural network can be thought of as doing linear regression in some high-dimensional feature space using a data-dependent, time-evolving feature map $\phi^{(t)}$! This is a powerful way to think about feature learning.

At this point, it might seem like we've cracked neural networks: they're just a more data-dependent version of linear regression.[6] A full understanding feels within reach. But this is an illusion. All we've really done is push the problem of feature learning up one level of abstraction; from evolving weights to an evolving kernel. Studying the latter is not necessarily any easier than studying the former. That being said, the NTK isn't a dead end. It gives us a natural place to start: the lazy regime, where the kernel stays (approximately) frozen and the dynamics become tractable (essentially the same dynamics as linear regression). Furthermore, we'll find that there are interesting insights to be gained about generalization from networks in the lazy regime.

For a network that is linear in its weights, such as the linearization $\hat{f}_{\text{lin}}$ from the start of this chapter, the NTK will stay constant throughout training. By construction $\phi^{(t)}_{\text{lin}}(x) = \nabla_\theta \hat{f}(x)|_{\theta=\theta^{(0)}}$ is independent of t, so $K^{(t)}_{\text{lin}} = K^{(0)}$ exactly, for all t. Conversely, a fixed NTK implies that the network equals its linearization throughout training. The dynamics of the linearized network are exactly those of a classical, well-studied statistical method: *kernel regression*. And we can solve it exactly like how we solved overparameterized linear regression in chapter 1. Given MSE loss,

$$\mathcal{L}\left(\hat{f}(X)\right) = \left\|y - \hat{f}(X)\right\|^2, \qquad \nabla_{\hat{f}(X)}\mathcal{L} = -2\left(y - \hat{f}(X)\right), \tag{15}$$

it will be convenient to again track the residual $r^{(t)} := y - \hat{f}^{(t)}_{\text{lin}}(X) \in \mathbb{R}^P$. Since the linearized network has a constant Jacobian $J^{(t)}_{\text{lin}} = J^{(0)}$, its kernel is the constant matrix $K^{(0)} = J^{(0)}J^{(0)\top}$, and Equation (10) becomes

$$\dot{\hat{f}}^{(t)}_{\text{lin}}(X) = 2\eta K^{(0)} r^{(t)} \qquad \Longrightarrow \qquad \dot{r}^{(t)} = -2\eta K^{(0)} r^{(t)}, \tag{16}$$

[6] In this sense, I lied a little when I said that linear regression is a terrible toy model of neural networks in chapter 1.

which is the same first order homogeneous linear ODE we met in chapter 1, with $XX^\top$ replaced by $K^{(0)}$. Its solution is

$$r^{(t)} = e^{-2\eta K^{(0)} t}\, r^{(0)}, \qquad \text{i.e.,} \qquad \hat{f}^{(t)}_{\text{lin}}(X) = y - e^{-2\eta K^{(0)} t}\left(y - \hat{f}^{(0)}(X)\right). \tag{17}$$

Equation (17) is the entire training trajectory of the network's predictions on the training set, in closed form, for all $t \geq 0$. Assuming $K^{(0)}$ is positive definite (the analogue of assuming the rows of X are linearly independent, and generically true for wide networks with $M \gg P$), the residual decays exponentially to zero and the network interpolates the training data.

If we want the weights too, we can integrate exactly as before. In weight space $\dot{\theta}^{(t)} = -\eta \nabla_\theta \mathcal{L} = 2\eta {J^{(0)}}^\top r^{(t)}$, so

$$\theta^{(t)} - \theta^{(0)} = 2\eta {J^{(0)}}^\top \left(\int_0^t e^{-2\eta K^{(0)} s}\, ds \right) r^{(0)} \tag{18}$$

$$= {J^{(0)}}^\top \left(K^{(0)}\right)^{-1} \left(I_P - e^{-2\eta K^{(0)} t}\right) \left(y - \hat{f}^{(0)}(X)\right), \tag{19}$$

which is Equation (**??**) verbatim under the substitution $X \to J^{(0)}$, $w \to \theta$. The implicit regularization story carries over as well: as $t \to \infty$ the weight displacement lies entirely in the row space of $J^{(0)}$, so among all weight vectors that interpolate the data, gradient flow finds the one minimizing $\|\theta - \theta^{(0)}\|$. Note the shift in reference point—in chapter 1 we needed $w_0 = 0$ to get a minimum-norm solution, whereas here the bias is toward *initialization* rather than the origin. This is exactly the sense in which lazy training is lazy: the network moves as little as it can get away with.

Finally, we usually care about predictions on data we have not trained on. Feeding the above into the linearization at an arbitrary test point x and writing $K^{(0)}(x, X) \in \mathbb{R}^{1 \times P}$ for the row vector with entries $\langle \phi^{(0)}(x), \phi^{(0)}(x_i) \rangle$,

$$\hat{f}^{(t)}_{\text{lin}}(x) = \hat{f}^{(0)}(x) + K^{(0)}(x, X) \left(K^{(0)}\right)^{-1} \left(I_P - e^{-2\eta K^{(0)} t}\right) \left(y - \hat{f}^{(0)}(X)\right), \tag{20}$$

and at convergence,

$$\hat{f}^{(\infty)}_{\text{lin}}(x) = \underbrace{\hat{f}^{(0)}(x)}_{\text{random init. function}} + \underbrace{K^{(0)}(x, X) \left(K^{(0)}\right)^{-1} \left(y - \hat{f}^{(0)}(X)\right)}_{\text{kernel regression on the residual labels}}. \tag{21}$$

Equation (21) is the punchline of the lazy regime. Up to the random function the network happened to be at initialization, an infinitely wide network trained to convergence in the lazy regime performs *kernel regression with the NTK*. Training does not discover features; the features $\phi^{(0)}$ were fixed the moment we sampled $\theta^{(0)}$, and gradient descent merely finds the least-squares fit in that frozen feature space.

One last observation that we will lean on heavily when we discuss generalization: since $K^{(0)}$ is symmetric PSD, we may diagonalize it as $K^{(0)} = \sum_i \lambda_i v_i v_i^\top$ and decompose the residual along its eigenbasis. Each component then decays independently,

$$\langle v_i, r^{(t)} \rangle = e^{-2\eta \lambda_i t} \langle v_i, r^{(0)} \rangle, \tag{22}$$

so the network learns the top eigendirections of its kernel first and the bottom ones exponentially more slowly. The NTK spectrum tells us, before training begins, which functions the network finds easy and which it finds hard.

$$\dot{\hat{f}}^{(t)}(X) = \eta K(y - \hat{f}^{(t)}(X)) \tag{23}$$

$$\implies \dot{\hat{f}}^{(t)}(X) = \eta K e^{-\eta K t}(y - \hat{f}^{(0)}(X)) \tag{24}$$

$$\hat{f}^{(T)}(X) - \hat{f}^{(0)}(X) = \int_0^T \left[\eta K e^{-\eta K t}(y - \hat{f}^{(0)}(X))\right] dt \tag{25}$$

$$= 2X^\top \left(\int_0^T e^{-2XX^\top t} dt\right)(y - Xw_0). \tag{26}$$

$$\lim_{T\to\infty} w(T) = X^\top (XX^\top)^{-1} y =: w^*. \tag{27}$$

But before we go on to study kernel regression, let's return to the problem at hand: showing that $1/\sqrt{H}$ scaling actually yields lazy training.

1.2 $1/\sqrt{H}$ scaling yields lazy training

Let us compute the NTK for the network we introduced earlier in the chapter, $\hat{f}(x) = \frac{1}{\sqrt{H}} \sum_{i=1}^H u_i \sigma(w_i^\top x)$, whose weights are $\theta = \{u_i, w_i\}_{i=1}^H$. The two derivatives we need are

$$\frac{\partial \hat{f}(x)}{\partial u_k} = \frac{1}{\sqrt{H}} \sigma(w_k^\top x), \qquad \frac{\partial \hat{f}(x)}{\partial w_k} = \frac{1}{\sqrt{H}} u_k \sigma'(w_k^\top x)\, x, \tag{28}$$

and summing their products over all weights gives

$$K_{ij} = \underbrace{\frac{1}{H} \sum_{k=1}^H \sigma(w_k^\top x_i)\sigma(w_k^\top x_j)}_{\text{from the output weights}} + \underbrace{(x_i^\top x_j)\, \frac{1}{H} \sum_{k=1}^H u_k^2\, \sigma'(w_k^\top x_i)\sigma'(w_k^\top x_j)}_{\text{from the hidden weights}}. \tag{29}$$

Look at the scaling. The kernel is a sum of H i.i.d. terms each carrying a $\frac{1}{H}$ prefactor. That's LLN scaling. The two derivatives each contributed a factor of $\frac{1}{\sqrt{H}}$, and their product turned the CLT-scaled sum into an LLN-scaled average. Thus, the law of large numbers applies to Equation (29), and as $H \to \infty$ the random matrix $K^{(0)}$ converges to a deterministic limit:

$$K^\infty(x, x') = \mathbb{E}_w \left[\sigma(w^\top x)\sigma(w^\top x')\right] + (x^\top x')\, \mathbb{E}_w \left[\sigma'(w^\top x)\sigma'(w^\top x')\right], \tag{30}$$

where we used $\mathbb{E}[u_i^2] = 1$ and the independence of u_i from w_i at initialization. This object is a deterministic function of the architecture, the nonlinearity, and the initialization distribution, with no dependence on any particular draw of the weights. This object is the *analytical* NTK[7]. The width shows up only in how far $K^{(0)}$ strays from it, and the CLT tells us those fluctuations are $\mathcal{O}(1/\sqrt{H})$.

Concentration at initialization is only half of what we need; the kernel could still drift once training starts. The following is an intuitive explanation for why it doesn't. Under CLT scaling,

[7] Also called the limiting or infinite-width NTK. Terminology in the literature is not consistent: "the NTK" sometimes means K^∞ and sometimes means $K^{(t)}$ at finite width. We will always write $K^{(t)}$ for the empirical kernel and K^∞ for the analytical one.

the gradient of the loss with respect to any single weight carries a factor of $\frac{1}{\sqrt{H}}$, so over an $\mathcal{O}(1)$ amount of training time each individual weight moves by only $\mathcal{O}(1/\sqrt{H})$. Since the weights themselves are $\mathcal{O}(1)$ at initialization, this is a vanishing relative change, i.e., the weights barely move, which is what "lazy" means. Yet the function moves plenty. The network sums H terms each with a $\frac{1}{\sqrt{H}}$ prefactor, so H weights each budging by $\mathcal{O}(1/\sqrt{H})$ conspire to change $\hat{f}$ by $\frac{1}{\sqrt{H}} \cdot H \cdot \frac{1}{\sqrt{H}} = \mathcal{O}(1)$. This is the resolution of the apparent paradox of lazy training: the network learns even though no individual weight does anything appreciable. And the kernel, being a smooth average of quantities that each shift by $\mathcal{O}(1/\sqrt{H})$, inherits that same $\mathcal{O}(1/\sqrt{H})$ deviation rather than an $\mathcal{O}(1)$ one. Concretely,

$$\sup_{t \geq 0} \left\| K^{(t)} - K^{(0)} \right\|_2 = \mathcal{O}\left(\frac{1}{\sqrt{H}}\right). \tag{31}$$

In the $H \to \infty$ limit, since $K^{(t)} = K^{(0)}$ exactly, it's clear that $\hat{f}^{(t)}(X) = \hat{f}^{(t)}_{\text{lin}}(X)$ should be true for all t. However, a stronger claim is true as the fact that the kernel only moves an $\mathcal{O}(\frac{1}{\sqrt{H}})$ amount can be leveraged to show that the network's predictions also only deviate an $\mathcal{O}(\frac{1}{\sqrt{H}})$ amount from the predictions of its linearization:

$$\sup_{t \geq 0} \left\| \hat{f}^{(t)}(X) - \hat{f}^{(t)}_{\text{lin}}(X) \right\|_2 = \mathcal{O}\left(\frac{1}{\sqrt{H}}\right). \tag{32}$$

The handwavey intuition given in the past few paragraphs can be made rigorous as mathematical proof. Lee et al. (2019)[8] do exactly that in appendix G of their paper titled "Wide Neural Networks of Any Depth Evolve as Linear Models Under Gradient Descent." But for our purposes, the handwavey intuition shall suffice.

2 Kernel regression

In this section, we will learn about *Eigenlearning*: a theoretical framework for understanding *generalization* in kernel regression. Generalization is an important term that we haven't talked about much. Let's talk about it.

2.1 On generalization and inductive biases

Recall the setup for a standard supervised learning task. We are given training data consisting of P pairs of data points $\{(x_i, y_i)\}_{i=1}^{P}$ where the inputs x are sampled (usually assumed i.i.d.) from some data distribution $x \sim \mathcal{D}$, and the outputs y are generated by some target function $y = f^*(x)$. We learn a model $\hat{f}(x)$ such that $\hat{f}(x_i) \approx f^*(x_i) = y_i$ for all $i = 1, \cdots, P$ and hope that it agrees with $f^*(x)$ on the entire data distribution $\mathcal{D}$.

The space of all functions that fit the P training points well is enormous. So, to ensure that the model learns something that actually generalizes to the entire data distribution $\mathcal{D}$, we might consider using a combination of architecture, learning rule, and loss that has an *inductive bias*[9] toward learning a simple, well-generalizing function. While characterizing the inductive biases of neural networks in general is incredibly difficult, doing so for networks specifically in the lazy regime is not, as it directly relates to the frozen NTK determined at initialization. We'll find that different architectures have different inductive biases that determine how easy or hard it is to learn a particular target function. We'll also learn about once mysterious deep learning phenomena such as double descent and benign overfitting.

[8] https://arxiv.org/pdf/1902.06720

[9] The min-norm bias of overparameterized linear regression from chapter 1 and the greedy (larger singular value modes learned first) bias of deep linear networks from chapter 2 are simple examples of inductive biases.

2.2 Eigenlearning

Consider an arbitrary data point $x \sim \mathcal{D}$ drawn from the data distribution. Consider the evolution of the prediction $\hat{f}(x)$ throughout training:

$$\dot{\hat{f}}^{(t)}(x) = \frac{\partial \hat{f}^{(t)}(x_0)}{\partial \theta}\dot{\theta} = -\eta \underbrace{\frac{\partial \hat{f}^{(t)}(x_0)}{\partial \theta} J^{(t)^\top}}_{=:k^\top(x)} \nabla_{\hat{f}(X)} \mathcal{L}\left(\hat{f}^{(t)}(X)\right) \tag{33}$$

THOUGHT. the utility of having a frozen kernel comes from the fact that the kernel is determined at init. this allows us to use RMT/stat-mechy arguments to think about model predictions and generalization.

3 Why lazy learning is bad