Modern Topics in Statistical Learning Theory Spring 2023

Lecture 6 — Mar. 3, 2023

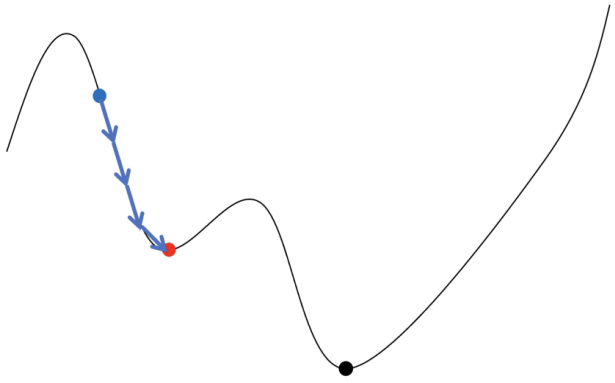
Prof.Qi Lei *Scribe: Noah Amsel, Tracy Zhu, Nikhil Vinay Sharma*

1 Recap

In the previous lecture, we introduced algorithms that optimize the objective in polynomial time. One important algorithm was Gradient Descent. There are two important points to remember about the algorithm.

- Gradient Descent converges to stationary points.
- Gradient Descent converges to the global minimum (when the objective function is convex).

The below illustration shows how gradient descent does not always find the global minimum. In the picture, gradient descent initialized at the blue point only makes it to the local minimum at the red point: it does not find the global minimum at the black point.



In this lecture, we introduce the neural tangent kernel approach which allows us to characterize the loss of general neural networks near a specific initialization (or under specific parameterization).

2 Linear Regression

Before discussing kernel methods, we review the simpler case of ordinary linear regression. We are given a dataset of points $\{(x_i, y_i)\}_{i=1}^n$, where $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. We want to fit a linear function

$f_\theta(x) = \theta^\top x$. Using the squared loss $l(z) = z^2$, the empirical risk is

$$\widehat{\mathcal{L}_n}(\theta) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2 = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \theta^\top x_i)^2$$

To fit the model, we use empirical risk minimization. Our goal is to solve the following optimization problem

$$\arg\min_\theta \widehat{\mathcal{L}_n}(\theta)$$

Since our loss is quadratic, this optimization problem is convex. Thus we can find the global minimum using Gradient Descent:

$$\begin{aligned}
\theta_{t+1} &= \theta_t - \eta_t \cdot \nabla\widehat{\mathcal{L}_n}(\theta) \\
&= \theta_t - \eta_t \cdot \frac{1}{n}\sum_{i=1}^{n}(y_i - f_\theta(x_i))\nabla_\theta f_\theta(x_i) \\
&= \theta_t - \eta_t \cdot \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta^\top x_i)x_i
\end{aligned}$$

Notice here that the gradient of the prediction function, $\nabla_\theta f_\theta(x) = x$, does not depend on θ. Of course, in many cases, linear functions are not expressive enough to fit the data.

3 Kernel Method

Again we are given $\{(x_i, y_i)\}_{i=1}^n$. We also make use of a non-linear function to make our prediction function more expressive

$$\phi : \mathbb{R}^d \to \mathbb{R}^D \qquad D >> d$$

and our predictor function is now $f_\theta(x) = \theta^\top\phi(x)$. For example, ϕ may be a polynomial kernel, like

$$\phi : \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \mapsto \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix}$$

To see the usefulness of this transformation, imagine a classification problem where the data is distributed on two circles centered at the origin of different radii. The label of each point is determined by its radius. Clearly linear models cannot solve this problem, since no straight line can separate one circle from the other. However, by setting $\theta = \begin{bmatrix}0 & 0 & 0 & 0 & 1 & 1\end{bmatrix}^\top$, our prediction function is $f_\theta(x) = x_1^2 + x_2^2$, which is the squared radius of x. Now it is simple to separate the two classes by simply thresholding this value. In fact, using this set of non-linear features, we can express any degree 2 polynomial in the entries of $x \in \mathbb{R}^2$.

Notice that $f_\theta(x)$ is non-linear in x, making it more expressive than linear functions. However, $f_\theta(x)$ is still linear in θ, making it easy to fit this model using gradient descent or other convex

optimization methods. A drawback of this method is that the non-linear features ϕ must be fixed in advance. It is usually difficult to know in advance what features are useful for fitting our dataset. To be safe, we could pick a very general ϕ. For example, we could choose a ϕ that allows to express any polynomial of the entries of x up to degree k, as above; however, if $x \in \mathbb{R}^d$, this requires our model to have $D = \Theta(d^k)$ features. This is prohibitively expensive in many cases. For example, images from the ImageNet dataset have $d \sim 10^5$, so even using relatively low-degree features like $k = 3$ would yield $D \sim 10^{15}$.

4 Kernel trick

Definition 1. *A kernel function K is a function that maps any pair (x_i, x_j) to an $n \times n$ matrix such that*

$$K(x_i, x_j) \triangleq \langle \phi(x_i), \phi(x_j) \rangle$$

The kernel matrix $K_m \in \mathbb{R}^{n \times n}$ is a positive semi-definite matrix.

$$\begin{pmatrix} (i,j)\text{-th element} \\ = \phi(x_i)^\top \phi(x_j) \end{pmatrix} \succcurlyeq 0$$

The kernel function enables us to compute $K(x_i, x_j)$ without an "explicit" composition of $\phi(x)$'s.

Kernel functions also enable us to compute solutions efficiently. One example is the polynomial kernel. The polynomial kernel $K_E(x_i, x_j) = \left(c + x_i^\top x_j\right)^k$ requires $O(d)$ in computational cost, whereas computing $\phi_E \in \mathbb{R}^D$ where $D = \Theta(d^k)$ is much more computationally inefficient.

Other examples of kernel tricks include kernel SVM, kernel ridge regularization, and kernel PCA. These methods use a kernel function to map data into a high-dimensional feature space, enabling them to achieve non-linear separability, more flexible decision boundaries, and and sometimes better performance compared to their non-kernel counterparts.

5 Neural network

Let us start with a simple 2-layer neural network.

Having parameters

$$\theta = (a_i, b_i)_{i=1}^m ,$$

a 2-layer neural network can be written as[1]

$$y = f_\theta(x) = \frac{1}{\sqrt{m}} \sum_{i=1}^m b_i \sigma\left(a_i^\top x\right)$$

with the (empirical) loss function

$$\hat{L}_n(\theta) = \frac{1}{2n} \sum_{i=1}^n \left(f_\theta\left(x_i\right) - y_i\right)^2 .$$

[1] we will explain why the scaling factor here is $\frac{1}{\sqrt{m}}$. This is to make sure the term does not vanish/explode.

Given $x \in \mathbb{R}^d$, it predicts the outcome $y \in \mathbb{R}$.

Our goal is

$$\min_{\theta} \hat{L}_n(\theta),$$

and the gradient descent can be derived as:

$$\theta_{t+1} \leftarrow \theta_t - \eta_t \cdot \frac{1}{n}\Big(\sum_{i=1}^{n} \left(f_\theta\left(x_i\right) - y_i\right) \cdot \underset{\text{change wrt } \theta \text{ and } x_i}{\nabla_\theta f_\theta(x_i)} \Big)$$

Note that when m is huge, the gradients updated each step become small and the parameters $\theta \in \mathbb{R}^{m(d+1)}$ are almost static. This is known as "lazy training".

Since the change in θ during training is small relative to the size of the initialization, we can use a first-order Taylor expansion about the initialization θ_0 to express the prediction function:

$$f(\theta; x) \approx f(\theta_0; x) + \nabla_\theta f(\theta_0; x)^\top (\theta - \theta_0) + O(\|\theta - \theta_0\|^2)$$

Notice that the right hand side is linear[2] in θ, though it is non-linear in x; this is similar to the kernel method above. This observation suggests defining the following kernel function:

Definition 2. *For a neural network $f(\theta; x)$ parameterized by θ with a random initialization θ_0, the neural tangent kernel (NTK) is*

$$\phi(x) \triangleq \nabla_\theta f(\theta_0; x)$$

The name comes to the fact that we are approximating a neural network prediction function by kernel regression, where the kernel is the gradient, which corresponds to the "tangent plane" to the prediction function. The corresponding kernel function is

$$K(x, x') = \langle \phi(x), \phi(x') \rangle$$

We will derive an expression for this kernel function and show that as as the width $m \to \infty$, $K(\cdot, \cdot)$ converges to an explicit function that we can write down.

Since we will be changing the number of neurons, we rewrite the neural network function so as to emphasize its dependence on m:

$$f_m(\theta; x) = \frac{1}{\sqrt{m}} \sum_{i=1}^{m} b_i \sigma(a_i^\top x) \qquad \theta = (a_i, b_i)_{i=1}^m$$

The corresponding kernel function will be denoted $K_m(\cdot, \cdot)$. To find an expression for this K_m, we begin by calculating the gradient of f with respect to each of the pieces of θ:

$$\nabla_{a_i} f_m(\theta_0; x) = \frac{1}{\sqrt{m}} b_i \sigma'(a_i^\top x) x$$
$$\nabla_{b_i} f_m(\theta_0; x) = \frac{1}{\sqrt{m}} \sigma(a_i^\top x)$$

[2]Actually this expression is affine in θ due to the constant $f(\theta_0; x)$. However, because we use a random initialization, the constant is nearly zero. Moreover, append a 1 to θ allows us to incorporate this constant into the inner product.

(Above, σ' is the derivative of σ.) Notice that since $\phi(x)$ is a concatenation of the gradient with respect to the as and the gradient with respect to the bs, we can decompose the kernel function into parts that depend only on the as and bs, respectively:

$$\begin{aligned} K_m(x, x') &= \langle \nabla_\theta f(\theta_0; x), \nabla_\theta f(\theta_0; x') \rangle \\ &= \langle \nabla_a f(\theta_0; x), \nabla_a f(\theta_0; x') \rangle + \langle \nabla_b f(\theta_0; x), \nabla_b f(\theta_0; x') \rangle \\ &=: K_m^{(a)}(x, x') + K_m^{(b)}(x, x') \end{aligned}$$

Using our expression above for the gradient with respect to a_i,

$$K_m^{(a)}(x, x') = \sum_{i=1}^{m} \langle \nabla_{a_i} f_m(\theta; x), \nabla_{a_i} f_m(\theta; x') \rangle = \frac{1}{m} \sum_{i=1}^{m} b_i^2 \sigma'(a_i^\top x) \sigma'(a_i^\top x') \langle x, x' \rangle$$

Similarly,

$$K_m^{(b)}(x, x') = \frac{1}{m} \sum_{i=1}^{m} \sigma(a_i^\top x) \sigma(a_i^\top x')$$

In the above expressions, the as and bs are components of θ_0. Recall that we initialized the network by drawing each parameter independently from a standard normal distribution:

$$a_{ij}, b_i \sim \mathcal{N}(0, 1)$$

So for fixed x and x', we can think of each term in the summations above as an instantiations of the random variables $b^2 \sigma'(a^\top x) \sigma'(a^\top x') \langle x, x' \rangle$ and $\sigma(a^\top x) \sigma(a^\top x')$, where $a \sim \mathcal{N}(0, I_d)$ and $b \sim \mathcal{N}(0, 1)$ are also random variables. Thus, as we take $m \to \infty$, these summations converge to expectations over the choice of a and b that do not depend on m (this explains the choice of normalization factor $1/\sqrt{m}$ that we made initially):

$$\begin{aligned} \lim_{m \to \infty} K_m^{(a)}(x, x') &= \mathbb{E}_{\substack{a \sim \mathcal{N}(0, I_d) \\ b \sim \mathcal{N}(0,1)}} b^2 \sigma'(a^\top x) \sigma'(a^\top x') \langle x, x' \rangle \\ \lim_{m \to \infty} K_m^{(b)}(x, x') &= \mathbb{E}_{\substack{a \sim \mathcal{N}(0, I_d) \\ b \sim \mathcal{N}(0,1)}} \sigma(a^\top x) \sigma(a^\top x') \end{aligned}$$

Of course, in reality, we cannot create a neural network with infinite width. However, we can still apply the NTK corresponding to an infinite width network using the kernel trick and the formulas above. If σ is the ReLU function, then σ' is the step function

$$\sigma'(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

This derivative is not defined at 0, but since we are taking expectations is okay; for $x \neq 0$, the probability that $a^\top x = 0$ is 0.

Notice that the distribution of a is rotationally symmetric. This means that if we rotated both x and x' by the same amount, the expectations above would not change. The directions of x and x' don't matter; only the angle between them, which we denote $\angle(x, x')$, does. To make this specific, assume σ is the ReLU function. We can rewrite

$$\sigma'(z) \sigma'(z') = \cdot \begin{cases} 1 & z, z' \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Thus,

$$\mathbb{E}_{a\sim\mathcal{N}(0,I_d)}\sigma'(a^\top x)\sigma'(a^\top x') = \mathbb{P}[(a^\top x > 0) \wedge (a^\top x' > 0)]$$

Thinking of a as a uniformly random direction and considering the geometry of the problem, it is possible to show that this probability is $(\pi - \angle(x, x'))/2\pi$. Since a and b are independent, we have

$$K_\infty^{(a)}(x, x') = \frac{\langle x, x'\rangle \mathbb{E}[b^2]}{2\pi} \cdot (\pi - \langle(x, x'))$$

A similar argument shows that

$$K_\infty^{(b)}(x, x') = \frac{\|x\| \cdot \|x'\| \cdot \mathbb{E}\|a\|^2}{2\pi d} \cdot \big((\pi - \angle(x, x'))\cos(\angle) + \sin(\angle)\big)$$

These formulas allow us to easily build the kernel matrix corresponding to the neural tangent kernel for an infinite width network.

Above, we approximated the neural network function by a first-order Taylor expansion. We now wish to show that this approximation is accurate. The second order Taylor expansion of the neural network function about θ_0 is

$$f(\theta; x) \approx f(\theta_0; x) + \langle \nabla_\theta f, \theta - \theta_0\rangle + \frac{1}{2}(\theta - \theta_0)^\top H(\theta - \theta_0)$$

where $H = \nabla_\theta^2 f(\theta_0; x)$ is the Hessian matrix of f at the point θ_0. This shows that the first-order approximation is accurate when the eigenvalues of H are bounded and $\theta \approx \theta_0$.

We have reduced our job to showing that $\theta \approx \theta_0$ throughout training. Since we are training with gradient descent, we know that $\theta \approx \theta_0$ when gradient descent converges quickly, since at each step we take only a small step away from θ_0. We have now reduced our job to showing that gradient descent converges quickly.

It would be difficult to prove that gradient descent converges quickly when applied to the neural network. However, during the first few training steps when $\theta \approx \theta_0$, we know that the neural networks acts like the NTK. It is possible to analyze gradient descent applied to the NTK, showing that it converges quickly. Thus, even for the real neural network, $\theta \approx \theta_0$ throughout the entire (short) training process. To summarize, the steps of our argument will be as follows:

1. When $\theta \approx \theta_0$, the neural network acts like the NTK.
2. Gradient descent applied to the NTK converges quickly.
3. Therefore, θ stays close to its initialization throughout training – both when training the NTK and when training the neural network.
4. Therefore the neural network trained with gradient descent acts like the NTK.

Instead of analyzing gradient descent directly, we analyze its continuous analogue, gradient flow. Recall the gradient descent update rule to minimize a function g:

$$\theta_{t+1} = \theta_t - \eta\nabla g(\theta_t)$$

We can rewrite this rule by parameterizing the learning rate in terms of a timestep Δt

$$\theta(t + \Delta t) = \theta(t) - \eta\Delta t\nabla g(\theta(t))$$

Rearranging yields

$$\frac{\theta(t+\Delta t)-\theta(t)}{\Delta t} = -\eta \nabla g(\theta(t))$$

Taking the limit as $\Delta t \to 0$, we get a differential equation describing a continuously evolving weight vector $\theta(t)$:

$$\frac{d\theta(t)}{dt} = -\nabla g(\theta(t))$$

(We can chose the units in which we measure time so as to get rid of the constant of proportionality η.) In our case, the function we are minimizing is the empirical risk

$$\widehat{\mathcal{L}_n}(\theta) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$$

Let $\widehat{y}_i(\theta) = f_\theta(x_i)$. Collect the y_is and $\widehat{y}_i$s into vectors $y = \begin{bmatrix} y_1 & \cdots & y_n \end{bmatrix}^\top$ and $\widehat{y}(\theta) = \begin{bmatrix} \widehat{y}_1(\theta) & \cdots & \widehat{y}_n(\theta) \end{bmatrix}^\top$. Then we can rewrite the empirical risk as

$$\widehat{\mathcal{L}_n}(\theta) = \frac{1}{2n}\|y - \widehat{y}(\theta)\|^2$$

Taking the gradient with respect to θ,

$$\nabla\widehat{\mathcal{L}_n}(\theta) = \frac{1}{n}[\nabla\widehat{y}(\theta)](\widehat{y}(\theta) - y)$$

The function $\widehat{y}(\theta)$ maps $\theta \in \mathbb{R}^D$ to $\widehat{y} \in \mathbb{R}^n$. Thus, the Jacobian of this function, $\nabla\widehat{y}(\theta)$, is a $D \times n$ matrix. The vector of errors $(\widehat{y}(\theta) - y)$ is of course n dimensional. Substituting this expression into the definition of gradient flow,

$$\frac{d\theta(t)}{dt} = -\frac{1}{n}[\nabla\widehat{y}(\theta)](\widehat{y}(\theta) - y)$$

Using the chain rule and plugging in the above expression, we get the derivative of the predictions $\widehat{y}$ made by the neural network as a function of training time:

$$\begin{aligned}\frac{d\widehat{y}(\theta(t))}{dt} &= [\nabla\widehat{y}(\theta)]^\top \frac{d\theta(t)}{dt} \\ &= [\nabla\widehat{y}(\theta)]^\top [\nabla\widehat{y}(\theta)](\widehat{y}(\theta) - y)\end{aligned}$$

Recalling the definition of $\widehat{y}$, we can rewrite

$$\nabla\widehat{y}(\theta) = \begin{bmatrix} | & & | \\ \nabla_\theta f(\theta; x_1) & \cdots & \nabla_\theta f(\theta; x_n) \\ | & & | \end{bmatrix}$$

This means that the i,jth entry of the $n \times n$ matrix $[\nabla\widehat{y}(\theta)]^\top[\nabla\widehat{y}(\theta)]$ is $\langle \nabla_\theta f(\theta; x_i), \nabla_\theta f(\theta; x_j)\rangle$. This matrix has the same form as the kernel matrix K. If we apply our assumption that $\theta \approx \theta_0$, then we can approximate these inner products as follows to make the analogy exact:

$$\langle \nabla_\theta f(\theta; x_i), \nabla_\theta f(\theta; x_j)\rangle \approx \langle \nabla_\theta f(\theta_0; x_i), \nabla_\theta f(\theta_0; x_j)\rangle = \langle \phi(x_i), \phi(x_j)\rangle$$

This implies $[\nabla \widehat{y}(\theta)]^\top [\nabla \widehat{y}(\theta)] \approx K$, the kernel matrix corresponding to the NTK and the dataset $x_1, \ldots, x_n$. Substituting this into the differential equation above,

$$\frac{d\theta(t)}{dt} \approx -K(\widehat{y}(\theta) - y)$$

Letting $u = \widehat{y} - y$, we can rewrite this as

$$\frac{du}{dt} \approx -Ku$$

Solving for u,

$$u(t) = u(0)e^{-Kt}$$

We know take the eigendecomposition of the kernel matrix

$$K = \sum_{i=1}^{n} \lambda_i v_i v_i^\top$$

Kernel matrices are always positive semidefinite. In fact since the model is overparameterized, it can be shown that the kernel is positive definite, that is $\lambda_i > 0$. Thus

$$u(t) = u(0) \prod_{i=1}^{n} e^{-\lambda_i v_i v_i^\top t}$$

converges to zero exponentially fast. That is, the model predictions $\hat{y}(\theta(t))$ converge to the ground truth labels y exponentially fast.

The NTK is a big step forward in understanding neural networks. But there is still much about neural networks' dynamics, optimization, and generalization behavior that is not captured by this analysis. For instance, the NTK does better than traditional kernel methods and can approach the performance of some neural networks, but even when the training and testing distributions are the same, state of the art neural networks outperform NTK methods in practice. This suggests that an important factor in neural networks' success is the ability to do feature learning, that is, to adapt the kernel to the data rather than fixing the kernel at initialization time. It is important to learn the lower layers of the network too, not just the top one.

References

[1] Jacot, Arughut, Franck Gabrield, and Clément Hongler. "Neural Tangent Kernel: Convergence and Generalization in neural networks." Advances in neural information processsing systems 31 (2018)

[2] Arora, Sanjeev, et al. "Fine-Grained Analysis of Optimization and Generalization for Overparameterized Two-Layer Neural Networks." International Conference on Machine Learning. PMLR, 2019.

[3] Chizat et al. "On Lazy Training in Differentiable Programming." NeurIPS, 2019.