7

Gradient-Based Learning

> *Of course, that's like saying Newton's second law $F = ma$, as it appears in textbooks on mechanics, is just a definition of what you mean by "force". That's true, strictly speaking, but we live in a landscape where there is an implicit promise that when someone writes that down . . . that they will give laws for the force, and not, say, for some quantity involving the 17th time derivative of the position.*
>
> Sidney Coleman, in his "Quantum Mechanics in Your Face" Dirac Lecture [56].

In the last chapter, we discussed Bayesian inference as a learning algorithm, which followed naturally from our study of networks at initialization. Starting from a description of a neural network architecture with parameters – weights and biases – we integrated out these parameters to find a distribution over preactivations $z^{(\ell)}(x)$ as a function of layer and input sample, which in particular included the output distribution $p\Big(z^{(L)}(x)\Big)$. This was interpreted as a *prior distribution* over an ensemble of such models, and then we explained how the logic of Bayes' rule lets us evolve the prior into a *posterior distribution* conditioned on observed data. Despite the theoretical elegance of Bayesian inference, the naive implementation quickly became computationally intractable as the number of conditioned data samples grew large.

Stepping back, there's actually something a little bit odd about this setup. Once we worked out the output distribution, the actual network itself was discarded, with the parameters long since integrated out. Since Bayesian inference only cares about the output distribution of a model, the starting point for inference can really be any ensemble of models as it isn't specifically tailored to neural networks at all. So why go through all the trouble of starting with neural-network models? How did we even know that these models are a good abstraction to begin with?

Deep neural networks are exciting because they work surprisingly well. We know this because in practice such networks are explicitly *trained* and used to perform useful tasks. Most commonly, learning occurs by repeatedly updating the model parameters via a gradient-based optimization procedure such as *gradient descent*.

In particular, gradient-based learning algorithms can efficiently process a large amount of training data by optimizing an auxiliary *loss* function that directly compares

https://doi.org/10.1017/9781009023405.009 Published online by Cambridge University Press

the network output $f(x;\theta) \equiv z^{(L)}(x)$ to some desired result or *label*. This optimization procedure involves sampling only a single set of network parameters from the initialization distribution, yielding just a single network trained for the task of interest rather than a full ensemble of networks. In this way, gradient-based learning methods offset their inability to express confidence in their predictions – due to the absence of an ensemble – with data efficiency and easy scalability.

Since gradient descent involves making explicit updates to the model parameters, the first step is to bring them back (from whatever place variables go when they are integrated out). In supervised learning, the adjustments of model parameters are directly proportional to the function-approximation error multiplied by the gradient of the model output with respect to the parameters. This decomposition motivates the study of the *neural tangent kernel* (NTK).[1] In short, the NTK is a type of Hamiltonian that controls the training dynamics of observables whenever gradient descent is used to optimize an auxiliary loss that scores a function approximation. As we detail in §10, §11, and §∞, understanding the NTK for a given neural-network architecture will enable us to effectively describe gradient-based learning for that model.

In this chapter, we give a short introduction to supervised learning in §7.1, followed by a discussion of gradient descent in §7.2 with a very general focus on how the NTK arises in supervised learning. In the next chapter, we'll incorporate the NTK into our effective theory of deep learning by exploiting the same layer-to-layer RG flow technique we used in §4.

7.1 Supervised Learning

One of the most basic modeling tasks at which neural networks excel is known as **supervised learning**. Given a **data distribution** $p(x, y) = p(y|x)p(x)$, the goal is to predict a **label** y given an input x, for any pair that is jointly sampled from the distribution.[2] To be precise, the model tries to learn the conditional distribution $p(y|x)$, and the resulting model is sometimes called a **discriminative model**. In one canonical example from computer vision, we might want to **classify** an image x_δ of a hand-written digit "3" according to its literal value $y_\delta = \texttt{3}$. Or, for a natural language processing example, given a sentence containing the word $x_\delta = \texttt{cat}$, we might want to identify the part of the speech as $y_\delta = \texttt{noun}$. The better the *probabilistic model* learns the distribution $p(y|x)$, the more accurately it can predict a true label y for a novel input example x. Generating these datasets generally requires human annotators to label the inputs, hence the name supervised learning.

[1] The NTK was first identified in the seminal work of Jacot *et al.* [57] in the context of infinite-width networks.

[2] In this section, we suppress *vectorial indices* on the inputs x_δ, labels y_δ, and model outputs $z(x_\delta;\theta)$, while often retaining *sample indices* $\delta \in \mathcal{D}$.

https://doi.org/10.1017/9781009023405.009 Published online by Cambridge University Press

In this setup, the supervised-learning model outputs a prediction $z(x_\delta;\theta)$. This notation emphasizes that the model output is a function of both the input x_δ and some adjustable parameters θ. This should already be familiar in the context of neural-network function approximation, where the model parameters consist of the biases and weights.

As discussed in §2.3, the model parameters are drawn from an easy-to-sample prior distribution over the parameters, which is also known as the *initialization distribution* in the context of gradient-based learning. Importantly, this parameter distribution knows nothing about the data distribution. Thus, in order for the model to make good predictions, its parameters will need to be adjusted somehow. Really, this is just a specific application of the function approximation that we discussed in §2.1 where the function to be approximated is a conditional distribution $p(y|x)$.

Before we understand how to adjust or *fit* the model parameters, we need to understand what we mean by making good predictions. What we want is, for a typical input x_δ and a label y_δ sampled from the data distribution $p(x,y)$, that the model output $z(x_\delta;\theta)$ is as close to the label y_δ as possible on average. In order to measure this proximity, for a prediction–label pair we need to define an auxiliary *objective function* or **loss**,

$$\mathcal{L}\Big(z(x_\delta;\theta),\, y_\delta\Big), \tag{7.1}$$

with the property that the closer $z(x_\delta;\theta)$ is to y_δ, the lower the value of the function is. One very intuitive choice for the loss is *MSE loss* (6.17),

$$\mathcal{L}_{\text{MSE}}\Big(z(x_\delta;\theta),\, y_\delta\Big) \equiv \frac{1}{2}\Big[z(x_\delta;\theta) - y_\delta\Big]^2, \tag{7.2}$$

which clearly has the required property, though this is not the most common choice in deep learning. The specific form of the loss will not matter for the rest of the chapter.

With the loss function in hand, the goal of training is to adjust the model parameters so as to minimize the loss for as many input–label pairs as possible. Ideally, we would like to minimize the loss averaged over the entire data distribution,

$$\mathbb{E}\left[\mathcal{L}(\theta)\right] = \int dxdy\, p(x,y)\, \mathcal{L}\Big(z(x;\theta),\, y\Big). \tag{7.3}$$

But since we almost never have access to the analytical form of the data distribution $p(x,y)$, in practice this would require the sampling of an infinite number of input–label pairs. Instead, as a proxy of the entire loss (7.3), we sample a large-but-finite number of pairs $(x_{\tilde\alpha}, y_{\tilde\alpha})_{\tilde\alpha\in\mathcal{A}}$ and try to minimize

$$\mathcal{L}_{\mathcal{A}}(\theta) \equiv \sum_{\tilde\alpha\in\mathcal{A}} \mathcal{L}\Big(z(x_{\tilde\alpha};\theta),\, y_{\tilde\alpha}\Big). \tag{7.4}$$

This set of examples $\mathcal{A}$ is referred to as the **training set**, and the estimate of the loss (7.4) is called the **training loss**; here we've also inherited from §6 our sample-index notation of alpha-with-tilde for the inputs in the training set $\tilde\alpha \in \mathcal{A}$, while denoting generic inputs as delta-with-no-decoration $\delta \in \mathcal{D}$, and soon we'll use beta-with-dot for

https://doi.org/10.1017/9781009023405.009 Published online by Cambridge University Press

inputs in the *test set* $\dot{\beta} \in \mathcal{B}$.[3] To train our model, we try to find a configuration of the model parameters that minimizes the training loss

$$\theta^\star = \arg\min_\theta \mathcal{L}_\mathcal{A}(\theta) = \arg\min_\theta \left[\sum_{\tilde{\alpha} \in \mathcal{A}} \mathcal{L}\Big(z(x_{\tilde{\alpha}}; \theta),\, y_{\tilde{\alpha}} \Big) \right]. \tag{7.6}$$

In the next section, we will present the gradient descent algorithm as a way to accomplish this goal.

Having set the minimization of the training loss (7.4) as our optimization problem, it is important to keep in mind that the true goal of supervised learning is the minimization of the loss over the entire data distribution in the sense of (7.3). Said another way, the question is not whether the model is able to memorize all the input–label pairs in the training set but rather whether it's able to generalize its predictions to additional input–label pairs not seen during training. One might then worry about whether a training set is *biased* in its sampling of the data distribution or whether there is high *variance* in a particular set of samples.

To explicitly assess this **generalization** property of a model, a separate set of input–label samples $(x_{\dot{\beta}}, y_{\dot{\beta}})_{\dot{\beta} \in \mathcal{B}}$ – known as the **test set** – is typically set aside and only used to evaluate a model after training is complete. To the extent that the training set $\mathcal{A}$ is representative of the full data distribution $p(x, y)$, decreasing the training loss will often decrease the entire loss (7.3), as estimated by the test loss $\mathcal{L}_\mathcal{B}$. We will address this question directly in §10.

7.2 Gradient Descent and Function Approximation

Considering the training loss minimization (7.6), we see that learning is a complicated optimization problem. Being entirely naive about it, in order to find extrema of a

[3] Note that our definition of the training loss (7.4) is a bit at odds with our definition of the expected loss (7.3). In particular, the expected loss is *intensive*, while the training loss is *extensive*, scaling linearly with the size of the training set $N_\mathcal{A} \equiv |\mathcal{A}|$. This latter choice is consistent with our first definition of this loss, (6.17), in the context of MLE as an approximate method for Bayesian model fitting in §6.2.1. There, the extensivity of the loss was natural according to the Bayesian framework: as the number of observed input–output pairs $N_\mathcal{A}$ increases, we want the likelihood to dominate the prior. As such, we will find it natural to follow that convention. (You also might more accurately call the extensive loss (6.17) the ~~*mean*~~ *squared error*.) However, from a non-Bayesian perspective, it is often customary to define a training loss as

$$\mathcal{L}_\mathcal{A}(\theta) \equiv \frac{1}{|\mathcal{A}|} \sum_{\tilde{\alpha} \in \mathcal{A}} \mathcal{L}\Big(z(x_{\tilde{\alpha}}; \theta),\, y_{\tilde{\alpha}} \Big), \tag{7.5}$$

which better corresponds to the expected loss (7.3). Since in the context of gradient-based learning the overall normalization can always be absorbed in a redefinition of the global learning rate η, to be introduced in the next section, the only advantage we see of this latter definition (7.5) is the better correspondence of the loss with its name.

https://doi.org/10.1017/9781009023405.009 Published online by Cambridge University Press

function, calculus instructs us to differentiate the training loss and find the value of the argument for which the resulting expression vanishes:

$$0 = \left.\frac{d\mathcal{L}_{\mathcal{A}}}{d\theta_\mu}\right|_{\theta=\theta^\star} . \tag{7.7}$$

Unfortunately this equation is exactly solvable only in special cases, for instance when the loss is quadratic in the model parameters. Rather than trying to find minima analytically, practitioners typically employ an iterative procedure to bring the loss closer and closer to a minimum.

Gradient descent is one such method that can be used to minimize nontrivial functions like the training loss (7.4), and so it's a natural candidate for model fitting. The algorithm involves the computation of the gradient of the loss and iteratively updates the model parameters in the (negative) direction of the gradient:

$$\theta_\mu(t+1) = \theta_\mu(t) - \eta \left.\frac{d\mathcal{L}_{\mathcal{A}}}{d\theta_\mu}\right|_{\theta_\mu=\theta_\mu(t)} , \tag{7.8}$$

where t keeps track of the number of steps in the iterative training process, with $t = 0$ conventionally being the point of initialization. Here, η is a positive **training hyperparameter** called the **learning rate**, which controls how large a step is taken in *parameter space*. Note that the computational cost of gradient descent scales linearly with the size of the dataset $\mathcal{A}$, as one just needs to compute the gradient for each sample and then add them up.

For sufficiently small learning rates, the updates (7.8) are guaranteed to decrease the training loss $\mathcal{L}_{\mathcal{A}}$. In order to see this, let us Taylor-expand the training loss around the current value of the parameters $\theta(t)$ and compute the change in the loss after making an update:

$$\Delta\mathcal{L}_{\mathcal{A}} \equiv \mathcal{L}_{\mathcal{A}}\Big(\theta(t+1)\Big) - \mathcal{L}_{\mathcal{A}}\Big(\theta(t)\Big) = -\eta \sum_\mu \left(\frac{d\mathcal{L}_{\mathcal{A}}}{d\theta_\mu}\right)^2 \Bigg|_{\theta=\theta(t)} + O(\eta^2). \tag{7.9}$$

As minus a sum of squares, this is strictly negative. Pretty typically, iterating these updates will eventually lead to (at least) a local minimum of the training loss. In practice, small variants of the gradient descent algorithm are responsible for almost all training and optimization in deep learning.[4]

[4] In particular, the most popular learning algorithm is **stochastic gradient descent** (SGD). SGD uses updates of the form

$$\theta_\mu(t+1) = \theta_\mu(t) - \eta \left.\frac{d\mathcal{L}_{\mathcal{S}_t}}{d\theta_\mu}\right|_{\theta_\mu=\theta_\mu(t)} , \tag{7.10}$$

where $\mathcal{S}_t$ is a subset of the training set, $\mathcal{S}_t \subset \mathcal{A}$. Each subset $\mathcal{S}_t$ is called a *mini-batch* or **batch**. Training is then organized by **epoch**, which is a complete pass through the training set. Typically, for each epoch

https://doi.org/10.1017/9781009023405.009 Published online by Cambridge University Press

Tensorial Gradient Descent

In one such variant, we can define a more general family of learning algorithms by modifying the update (7.8) as

$$\theta_\mu(t+1) = \theta_\mu(t) - \eta \sum_\nu \lambda_{\mu\nu} \frac{d\mathcal{L}_\mathcal{A}}{d\theta_\nu}\bigg|_{\theta=\theta(t)}, \tag{7.11}$$

where the tensor $\lambda_{\mu\nu}$ is a **learning-rate tensor** on parameter space; the original gradient-descent update (7.8) is a special case with the Kronecker delta as the tensor, $\lambda_{\mu\nu} = \delta_{\mu\nu}$. While in the original gradient descent (7.8) we have one global learning rate η, in the tensorial gradient descent (7.11) we have the freedom to separately specify how the ν-th component of the gradient $d\mathcal{L}_\mathcal{A}/d\theta_\nu$ contributes to the update of the μ-th parameter θ_μ via the tensor $\lambda_{\mu\nu}$. Repeating the same Taylor expansion in η (7.9) with the generalized update (7.11), we find

$$\Delta\mathcal{L}_\mathcal{A} = -\eta \sum_{\mu,\nu} \lambda_{\mu\nu} \frac{d\mathcal{L}_\mathcal{A}}{d\theta_\mu} \frac{d\mathcal{L}_\mathcal{A}}{d\theta_\nu} + O(\eta^2), \tag{7.12}$$

indicating that the training loss again is almost surely decreasing for sufficiently small learning rates, so long as the learning-rate tensor $\lambda_{\mu\nu}$ is a positive semidefinite matrix.

Neural Tangent Kernel

Everything we have said so far about gradient descent could be applied equally to the optimization of any function. However, in the context of function approximation there is additional structure: the optimization objective is a function of the model output.

To take advantage of this structure, first note that by the chain rule the gradient of the loss can be expressed as

$$\frac{d\mathcal{L}_\mathcal{A}}{d\theta_\mu} = \sum_{i=1}^{n_\text{out}} \sum_{\tilde{\alpha}\in\mathcal{A}} \frac{\partial\mathcal{L}_\mathcal{A}}{\partial z_{i;\tilde{\alpha}}} \frac{dz_{i;\tilde{\alpha}}}{d\theta_\mu}, \tag{7.13}$$

which means that the change in the loss (7.12) after an update can be nicely decomposed as

$$\Delta\mathcal{L}_\mathcal{A} = -\eta \sum_{i_1,i_2=1}^{n_\text{out}} \sum_{\tilde{\alpha}_1,\tilde{\alpha}_2\in\mathcal{A}} \left[\frac{\partial\mathcal{L}_\mathcal{A}}{\partial z_{i_1;\tilde{\alpha}_1}} \frac{\partial\mathcal{L}_\mathcal{A}}{\partial z_{i_2;\tilde{\alpha}_2}}\right] \left[\sum_{\mu,\nu} \lambda_{\mu\nu} \frac{dz_{i_1;\tilde{\alpha}_1}}{d\theta_\mu} \frac{dz_{i_2;\tilde{\alpha}_2}}{d\theta_\nu}\right] + O(\eta^2). \tag{7.14}$$

---

the training set is *stochastically* partitioned into subsets of equal size, which are then sequentially used to estimate the gradient.

The advantage of this algorithm is twofold: *(i)* the computational cost of training now scales with the fixed size of the sets $\mathcal{S}_t$ rather than with the size of the whole training set $\mathcal{A}$ and *(ii)* SGD is thought to have better generalization properties than gradient descent. Nevertheless, essentially everything we will say about gradient descent will apply to stochastic gradient descent as well.

https://doi.org/10.1017/9781009023405.009 Published online by Cambridge University Press

The quantity in the first square bracket is a measure of the function approximation error. For instance, for the MSE loss (7.2) we see that the gradient of the loss with respect to the model output is exactly the prediction error,

$$\frac{\partial \mathcal{L}_{\mathcal{A}}}{\partial z_{i;\tilde{\alpha}}} = z_i(x_{\tilde{\alpha}}; \theta) - y_{i;\tilde{\alpha}}. \tag{7.15}$$

More generally for other losses, the gradient of the loss or **error factor**

$$\epsilon_{i;\tilde{\alpha}} \equiv \frac{\partial \mathcal{L}_{\mathcal{A}}}{\partial z_{i;\tilde{\alpha}}} \tag{7.16}$$

is small when the model output is close to the label. Sensibly, the greater the error factor, the larger the update (7.13), and the greater the change in the loss (7.14). The quantity in the second square bracket is called the **neural tangent kernel** (NTK)

$$H_{i_1 i_2;\tilde{\alpha}_1\tilde{\alpha}_2} \equiv \sum_{\mu,\nu} \lambda_{\mu\nu} \frac{dz_{i_1;\tilde{\alpha}_1}}{d\theta_\mu} \frac{dz_{i_2;\tilde{\alpha}_2}}{d\theta_\nu}. \tag{7.17}$$

As is clear from (7.17), the NTK is independent of the auxiliary loss function.

Importantly, the NTK is the main driver of the function-approximation dynamics. To the point, it governs the evolution of a much more general set of observables than the training loss. Consider any observable that depends on the model's outputs:

$$\mathcal{O}(\theta) \equiv \mathcal{O}\Big(z(x_{\delta_1}; \theta), \ldots, z(x_{\delta_M}; \theta)\Big), \tag{7.18}$$

where $x_{\delta_1}, \ldots, x_{\delta_M} \in \mathcal{D}$ for some dataset $\mathcal{D}$. For example, if $\mathcal{D}$ is the test set $\mathcal{B}$, and $\mathcal{O}$ is the loss function, then this observable would be the test loss $\mathcal{L}_{\mathcal{B}}$. In addition to the test loss, one might want to observe the change in a particular component of the output $\mathcal{O} = z_i(x)$ or perhaps track correlations among different vectorial components of the output $\mathcal{O} = z_i(x)\, z_j(x)$ for a given input x. For any such observable (7.18), its change after an update is given by the expression

$$\mathcal{O}\Big(\theta(t+1)\Big) - \mathcal{O}\Big(\theta(t)\Big) = -\eta \sum_{i_1,i_2=1}^{n_{\text{out}}} \sum_{\tilde{\alpha}\in\mathcal{A}} \sum_{\delta\in\mathcal{D}} \left[\frac{\partial \mathcal{L}_{\mathcal{A}}}{\partial z_{i_1;\tilde{\alpha}}} \frac{\partial \mathcal{O}}{\partial z_{i_2;\delta}}\right] H_{i_1 i_2;\tilde{\alpha}\delta} + O(\eta^2). \tag{7.19}$$

As we see, the square bracket contains the function-approximation error as well as the particulars about how the observable depends on the model output. In contrast, the NTK contains all the dynamical information pertaining to the particular model, depending only on the model architecture and parameters.[5]

We can further understand the function-approximation dynamics under gradient descent by considering a particular vectorial component of the output for a particular

[5] As our discussion makes clear, the NTK can generally be defined for any function approximator. This means that its name masks its true generality. In addition to objecting to the "neural" part of the name, one could object to the "kernel" part. In particular, the NTK is more akin to a Hamiltonian than a kernel as it generates the evolution of observables; we'll fully justify this claim in §∞.2.2.

https://doi.org/10.1017/9781009023405.009 Published online by Cambridge University Press

sample as an observable, i.e., $\mathcal{O} = z_i(x_\delta)$. In this case, the derivative of $\mathcal{O}$ in (7.19) is a Kronecker delta on both the vectorial indices and sample indices, and the evolution reduces to

$$z_i\Big(x_\delta; \theta(t+1)\Big) - z_i\Big(x_\delta; \theta(t)\Big) = -\eta \sum_{j=1}^{n_{\text{out}}} \sum_{\tilde{\alpha} \in \mathcal{A}} H_{ij;\delta\tilde{\alpha}} \epsilon_{j;\tilde{\alpha}} + O(\eta^2). \tag{7.20}$$

This equation shows how the model output changes after a training update. Importantly, we see how the error factor $\epsilon_{j;\tilde{\alpha}}$ (7.16) from example $x_{\tilde{\alpha}}$ on the model output component j affects the updated behavior of the model output component i on a different example x_δ: it's mediated by the NTK component $H_{ij;\delta\tilde{\alpha}}$. This is what makes function approximation possible: the ability to learn something about one example, x_δ, by observing another, $x_{\tilde{\alpha}}$. We see that the off-diagonal components of the NTK in the sample indices determine the generalization behavior of the model, while the off-diagonal components in vectorial indices allow for one feature to affect the training of another feature. We will have more to say about the former property in §10 and the latter property in §∞.

Finally, let us note in passing that unlike the case of the training loss (7.14), for general observables (7.19) the term in the square bracket is not necessarily positive. While the training loss $\mathcal{L}_{\mathcal{A}}$ will always decrease for small enough learning rates, a given observable may not. In particular, nothing guarantees that the test loss will decrease, and – for models that overfit their training set – the test loss may even increase.

https://doi.org/10.1017/9781009023405.009 Published online by Cambridge University Press