7

Deep learning

The past chapters have sketched a path towards predictive modeling: acquire data, construct a set of features that properly represent data in a way such that relevant conditions can be discriminated, pose a convex optimization problem that balances fitting training data to managing model complexity, optimize this problem with a standard solver, and then reason about generalization via the holdout method or cross validation. In many ways this pipeline suffices for most predictive tasks.

However, this standard practice does have its deficiencies. Feature engineering has many moving pieces, and choices at one part of the pipeline may influence downstream decisions in unpredictable ways. Moreover, different software dependencies may be required to intertwine the various parts of this chain, making the machine learning engineering more fragile. It's additionally possible that more concise feature representations are possible if the pieces can all be tuned together.

Though motivated differently by different people, *deep learning* can be understood as an attempt to "delayer" the abstraction boundaries in the standard machine learning workflow. It enables holistic design of representation and optimization. This delayering comes at the cost of loss of convexity and some theoretical guarantees on optimization and generalization. But, as we will now describe in this chapter, this cost is often quite modest and, on many machine learning problems such as image classification and machine translation, the predictive gains can be dramatic.

Deep learning has been tremendously successful in solving industrial machine learning problems at many tech companies. It is also the top performing approach in most academic prediction tasks in computer vision, speech, and other domains.

The success of deep learning is not just a purely technical matter. Once the industry had embraced deep learning, an unprecedented amount of resources has gone into building and refining high quality software for practicing deep learning. The open source deep learning ecosystem is vast and evolving quickly. For almost any task, there is already some code available to start from. Companies are actively competing over open source frameworks with convenient high-level syntax and extensive documentation. An honest answer for why practitioners prefer deep learning at this point

over other methods is because it simply seems to work better on many problems and there is a lot of quality code available.

We now retrace our path through representation, optimization, and generalization, highlighting what is different for deep learning and what remains the same.

Deep models and feature representation

We discussed in the chapter on representation that template matching, pooling, and nonlinear lifting can all be achieved by affine transformations followed by pointwise nonlinearities. These mappings can be chained together to give a series of new feature vectors:

$$x_{\ell+1} = \phi(A_\ell x_\ell + b_\ell)\,.$$

Here, ℓ indexes the *layer* of a model. We can chain several layers together to yield a final representation x_L.

As a canonical example, suppose x_1 is a pixel representation of an image. Let's say this representation has size $d_1 \times d_1 \times c_1$, with d_1 counting spatial dimensions and c_1 counting the number of color channels. We could apply c_2 template matching convolutions to this image, resulting in a second layer of size $d_1 \times d_1 \times c_2$. Since we expect convolutions to capture local variation, we can compress the size of this second layer, averaging every 2×2 cell to produce x_2 of size $d_2 \times d_2 \times c_2$, with $d_2 < d_1$ and $c_2 > c_1$. Repeating this procedure several times will yield a representation x_{L-1} which has few spatial dimensions (d_{L-1} is small) but many channel dimensions (c_{L-1} is large). We can then map this penultimate layer through some universal approximator like a neural network.

A variety of machine learning pipelines can be thought of in this way. The first layer might correspond to edge detectors like in SIFT[1] or HOG.[2] The second layer may look for parts relevant to detection like in a deformable parts model.[3] The insight in deep learning is that we can declare the parameters of each layer A_ℓ and b_ℓ to be *optimization variables*. This way, we do not have to worry about the particular edge or color detectors to use for our problem, but can instead let the collected data dictate the best settings of these features.

This abstraction of "features" as "structured linear maps with tunable parameters" allows for a set of basic building blocks that can be used across a variety of domains.

1. **Fully connected layers.** Fully connected layers are simply unstructured neural networks that we discussed in the representation chapter.

For a fixed nonlinear function σ, a fully connected layer maps a vector x to a vector z with coordinates

$$z_i = \sigma\left(\sum_j A_{ij}x_j + b_i\right).$$

While it is popular to chain fully connected layers together to get deep neural networks, there is little established advantage over just using a single layer. Daniely et al. have backed up this empirical observation, showing theoretically that no new approximation power is gained by concatenating fully connected layers.[4] Moreover, as we will discuss below, concatenating many layers together often slows down optimization. As with most things in deep learning, there's nothing saying you can't chain fully connected layers, but we argue that most of the gains in deep learning come from the structured transforms, including the ones we highlight here.

2. **Convolutions**. Convolutions are the most important building block in all of deep learning. We have already discussed the use of convolutions as template matchers that promote spatial invariances. Suppose the input is $d_0 \times d_0 \times c_0$, with the first two components indexing space and the last indexing channels. The parameter A has size $q_0 \times q_0 \times c_0 \times c_1$, where q_0 is usually small (greater than 2 and less than 10. b typically has size c_1. The number of parameters used to define a convolutional layer is dramatically smaller than what would be used in a fully connected layer. The structured linear map of a convolution can be written as

$$z_{a,b,c} = \sigma\left(\sum_{i,j,k} A_{i,j,k,c}x_{a-i,b-j,k} + b_c\right).$$

3. **Recurrent structures.** Recurrent structures let us capture repeatable stationary patters in time or space. Suppose we expect stationarity in time. In this case, we expect each layer to represent a state of the system, and the next time step should be a static function of the previous:

$$x_{t+1} = f(x_t)$$

When we write f as a neural network, this is called a *recurrent neural network*. Recurrent neural networks *share weights* insofar as the f does not change from one time step to the next.

4. **Attention mechanisms.** Attention mechanisms have proven powerful tools in natural language processing for collections of vectors with dependencies that are not necessarily sequential. Suppose our layer

is a list of m vectors of dimension d. That is, x has shape $d \times m$. An attention layer will have two matrices U and V, one to operate on the feature dimensions and one to operate on the sequential dimensions. The transformation takes form

$$z_{a,b} = \sigma\left(\sum_{i,j} U_{a,i} x_{ij} V_{b,j}\right) .$$

Just as was the case with convolutions, this structured map can have fewer dimensions than a fully connected layer, and can also respect a separation of the feature dimensions from the sequential dimensions in the data matrix x.

Optimization of deep nets

Once we have settled on a feature representation, typically called *model architecture* in this context, we now need to solve empirical risk minimization. Let's group all of the parameters of the layers into a single large array of weights, w. We denote the map from x to prediction with weights w as $f(x; w)$. At an abstract level, empirical risk minimization amounts to minimizing

$$R_S[w] = \frac{1}{n} \sum_{i=1}^{n} loss(f(x_i; w), y_i) .$$

This is a nonconvex optimization problem, but we can still run gradient methods to try to find minimizers. Our main concerns from an optimization perspective are whether we run into local optima and how can we compute gradient steps efficiently.

We will address gradient computation through a discussion of automatic differentiation. With regards to global optimization, there are unfortunately computational complexity results proving efficient optimization of arbitrary neural networks is intractable. Even neural nets with a single neuron can have exponentially many local minimizers,[5] and finding the minimum of a simple two-layer neural network is NP-hard.[6,7] We cannot expect a perfectly clean mathematical theory guiding our design and implementation of neural net optimization algorithms.

However, these theoretical results are about the *worst case*. In practice, optimization of neural nets is often easy. If the loss is bounded below by zero, any model with zero loss is a global minimizer. As we discussed in the generalization chapter, one can quickly find a global optimum of a state-of-the-art neural net by cloning a GitHub repo and turning off the various regularization schemes. In this section, we aim to provide some insights

about the disconnect between computational complexity in the worst case and the results achieved in practice. We provide some partial insights as to why neural net optimization is doable by studying the convergence of the predictions and how this convergence can be aided by overparameterization.

Convergence of predictions in nonlinear models

Consider the special case where we aim to minimize the square-loss. Let $\widehat{y}_t$ denote the vector of predictions $\big(f(x_i; w_t)\big)_{i=1}^n \in \mathbb{R}^n$. Gradient descent follows the iterations

$$w_{t+1} = w_t - \alpha \mathsf{D}_w f(x; w_t)(\widehat{y}_t - y)$$

where $\mathsf{D}_w f(x; w_t)$ denotes the $d \times n$ Jacobian matrix of the predictions $\widehat{y}_t$. Reasoning about convergence of the weights is difficult, but we showed in the optimization chapter that we could reason about convergence of the predictions:

$$\widehat{y}_{t+1} - y = (I - \alpha \mathsf{D}_w f(x; w_t)^T \mathsf{D}_w f(x; w_t))(\widehat{y}_t - y) + \alpha \epsilon_t \,.$$

Here,

$$\epsilon_t = \frac{\alpha}{2} \Lambda \|\widehat{y}_t - y\|^2$$

and Λ bounds the curvature of the f. If ϵ_t is sufficiently small the predictions will converge to the training labels.

Hence, under some assumptions, nonconvexity does not stand in the way of convergence to an empirical risk minimizer. Moreover, control of the Jacobians $\mathsf{D}_w f$ can accelerate convergence. We can derive some reasonable ground rules on how to keep $\mathsf{D}_w f$ well conditioned by unpacking how we compute gradients of compositions of functions.

Automatic differentiation

For linear models it's not hard to calculate the gradient with respect to the model parameters analytically and to implement the analytical expression efficiently. The situation changes once we stack many operations on top of each other. Even though in principle we can calculate gradients using the chain rule, this gets tedious and error-prone very quickly. Moreover, the straightforward way of implementing the chain rule is computationally inefficient. What's worse is that any change to the model architecture would require us to redo our calculation and update its implementation.

Fortunately, the field of *automatic differentiation* has the tools to avoid all of these problems. At a high-level, automatic differentiation provides

efficient algorithms to compute gradients of any function that we can write as a composition of differentiable building blocks. Though automatic differentiation has existed since the 1960s, the success of deep learning has led to numerous free, well-engineered, and efficient automatic differentiation packages. Moreover, the dynamic programming algorithm behind these methods is quite instructive as it gives us some insights into how to best engineer model architectures with desirable gradients.

Automatic differentiation serves two useful purposes in deep learning. First, it lowers the barrier to entry, allowing practitioners to think more about modeling than about the particulars of calculus and numerical optimization. Second, a standard automatic differentiation algorithm helps us write down parseable formulas for the gradients of our objective function so we can reason about what structures encourage faster optimization.

To define this dynamic programming algorithm, called *backpropagation*, it's helpful to move up a level of abstraction before making the matter more concrete again. After all, the idea is completely general and not specific to deep models. Specifically, we consider an abstract computation that proceeds in L steps starting from an input $z^{(0)}$ and produces an output $z^{(L)}$ with $L-1$ intermediate steps $z^{(1)}, \ldots, z^{(L-1)}$:

$$\begin{aligned} \text{input} \quad & z^{(0)} \\ & z^{(1)} = f_1(z^{(0)}) \\ & \vdots \\ & z^{(L-1)} = f_{L-1}(z^{(L-2)}) \\ \text{output} \quad & z^{(L)} = f_L(z^{(L-1)}) \end{aligned}$$

We assume that each *layer* $z^{(i)}$ is a real-valued vector and that each function f_i maps a real-valued vector of some dimension to another dimension. Recall that for a function $f\colon \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian matrix $\mathsf{D}(w)$ is the $m \times n$ matrix of first-order partial derivatives evaluated at the point w. When $m = 1$ the Jacobian matrix coincides with the transpose of the gradient.

In the context of backpropagation, it makes sense to be a bit more explicit in our notation. Specifically, we will denote the Jacobian of a function f with respect to a variable x evaluated at point w by $\mathsf{D}_x f(w)$.

The backpropagation algorithm is a computationally efficient way of computing the partial derivatives of the output $z^{(L)}$ with respect to the input $z^{(0)}$ evaluated at a given parameter vector w, that is, the Jacobian $\mathsf{D}_{z^{(0)}} z^{(L)}(w)$. Along the way, it computes the Jacobians for any of the intermediate layers as well.

Backpropagation

- Input: parameters w
- Forward pass:
  - Set $v_0 = w$
  - For $i = 1, \ldots, L$:
    - Store v_{i-1} and compute $v_i = f_i(v_{i-1})$
- Backward pass:
  - Set $\Lambda_L = \mathsf{D}_{z^{(L)}} z^{(L)}(v_L) = I$.
  - For $i = L, \ldots, 1$:
    - Set $\Lambda_{i-1} = \Lambda_i \mathsf{D}_{z^{(i-1)}} z^{(i)}(v_{i-1})$.
- Output Λ_0.

First note that backpropagation runs in time $O(LC)$ where C is the computational cost of performing an operation at one step. On the forward pass, this cost correspond to function evaluation. On the backward pass, it requires computing the partial derivatives of $z^{(i)}$ with respect to $z^{(i-1)}$. The computational cost depends on what the operation is. Typically, the i-th step of the backward pass has the same computational cost as the corresponding i-th step of the forward pass up to constant factors. What is important is that computing these partial derivatives is an entirely *local* operation. It only requires the partial derivatives of a function with respect to its input evaluated at the value v_{i-1} that we computed in the forward pass. This observation is key to all fast implementations of backpropagation. Each operation in our computation only has to implement function evaluation, its first-order derivatives, and store an array of values computed in the forward pass. There is nothing else we need to know about the computation that comes before or after.

The main correctness property of the algorithm is that the final output Λ_0 equals the partial derivative of the output layer with respect to the input layer evaluated at the input w.

Proposition 1. ***Correctness of backpropagation.***

$$\Lambda_0 = \mathsf{D}_{z^{(0)}} z^{(L)}(w)$$

The claim directly follows by induction from the following lemma, which states that if we have the correct partial derivatives at step i of the backward pass, then we also get them at step $i - 1$.

Lemma 1. ***Backpropagation invariant.***

$$\Lambda_i = \mathsf{D}_{z^{(i)}} z^{(L)}(v_i) \quad \Longrightarrow \quad \Lambda_{i-1} = \mathsf{D}_{z^{(i-1)}} z^{(L)}(v_{i-1})$$

Proof. Assume that the premise holds. Then, we have

$$\begin{aligned}\Lambda_{i-1} &= \Lambda_i \mathsf{D}_{z^{(i-1)}} z^{(i)}(v_{i-1}) \\ &= \mathsf{D}_{z^{(i)}} z^{(L)}(v_i) \mathsf{D}_{z^{(i-1)}} z^{(i)}(v_{i-1}) \\ &= \mathsf{D}_{z^{(i-1)}} z^{(L)}(v_{i-1})\end{aligned}$$

The last identity is the multivariate chain rule.

□

To aid the intuition, it can be helpful to write the multivariate chain rule informally in the familiar Leibniz notation:

$$\frac{\partial z^{(L)}}{\partial z^{(i)}} \frac{\partial z^{(i)}}{\partial z^{(i-1)}} = \frac{\partial z^{(L)}}{\partial z^{(i-1)}}$$

A worked out example

The backpropagation algorithm works in great generality. It produces partial derivatives for any variable appearing in any layer $z^{(i)}$. So, if we want partial derivatives with respect to some parameters, we only need to make sure they appear on one of the layers. This abstraction is so general that we can easily express all sorts of deep architectures and the associated objective functions with it.

But let's make that more concrete and get a feeling for the mechanics of backpropagation in the two cases that are most common: a non-linear transformation applied coordinate-wise, and a linear transformation.

Suppose at some point in our computation we apply the *rectified linear unit* $\mathrm{ReLU} = \max\{u, 0\}$ coordinate-wise to the vector u. ReLU units remain one of the most common non-linearities in deep neural networks. To implement the backward pass, all we need to be able to do is to compute the partial derivatives of $\mathrm{ReLU}(u)$ with respect to u. It's clear that when $u_i > 0$, the derivative is 1. When $u_i < 0$, the derivative is 0. The derivative is not defined at $u_i = 0$, but we choose to ignore this issue by setting it to be 0. The resulting Jacobian matrix is a diagonal matrix $D(u)$ which has a one in all coordinates corresponding to positive coordinates of the input vector, and is zero elsewhere.

- Forward pass:
  - Input: u
  - Store u and compute the value $v = \mathrm{ReLU}(u)$.
- Backward pass:

  - Input: Jacobian matrix Λ
  - Output $\Lambda D(u)$

If we were to swap out the rectified linear unit for some other coordinate-wise nonlinearity, we'd still get a diagonal matrix. What changes are the coefficients along the diagonal.

Now, let's consider an affine transformation of the input $v = Au + b$. The Jacobian of an affine transformation is simply the matrix A itself. Hence, the backward pass is simple:

- Backward pass:
  - Input: Jacobian matrix Λ
  - Output: ΛA

We can now easily chain these together. Suppose we have a typical ReLU network that strings together L linear transformations with ReLU operations in between:

$$f(x) = A_L \text{ReLU}(A_{L-1} \text{ReLU}(\cdots \text{ReLU}(A_1 x)))$$

The Jacobian of this chained operation with respect to the input x is given by a chain of linear operations:

$$A_L D_{L-1} A_{L-1} \cdots D_1 A_1$$

Each matrix D_i is a diagonal matrix that zeroes out some coordinates and leaves others untouched. Now, in a deep model architecture the weights of the linear transformation are typically trainable. That means we really want to be able to compute the gradients with respect to, say, the coefficients of the matrix A_i.

Fortunately, the same reasoning applies. All we need to do is to make A_i part of the input to the matrix-vector multiplication node and backpropagation will automatically produce derivatives for it. To illustrate this point, suppose we want to compute the partial derivatives with respect to, say, the j-th column of A_i. Let $u = \text{ReLU}(A_{i-1} \cdots (A_1 x))$ be the vector that we multiply A_i with. We know from the argument above that the Jacobian of the output with respect to the vector $A_i u$ is given by the linear map $B = A_L D_{L-1} A_{L-1} \cdots D_i$. To get the Jacobian with respect to the j-th column of A_i we therefore only need to find the partial derivative of $A_i u$ with respect to the j-th column of A_i. We can verify by taking derivatives that this equals $u_j I$. Hence, the partial derivatives of $f(x)$ with respect to the j-th column of A_i are given by Bu_j.

Let's add in the final ingredient, a loss function. Suppose A_L maps into one dimension and consider the squared loss $\frac{1}{2}(f(x) - y)^2$. The only thing

this loss function does is to scale our Jacobian by a factor $f(x) - y$. In other words, the partial derivatives of the loss with respect to the j-th columns of the weight matrix A_i is given by $(f(x) - y)Bu_j$.

As usual with derivatives, we can interpret this quantity as the *sensitivity* of our loss to changes in the model parameters. This interpretation will be helpful next.

Vanishing gradients

The previous discussion revealed that the gradients of deep models are produced by chains of linear operations. Generally speaking, chains of linear operations tend to either blow up the input exponentially with depth, or shrink it, depending on the singular values of the matrices. It is helpful to keep in mind the simple case of powering the same symmetric real matrix A a number of times. For almost all vectors u, the norm $\|A^L u\|$ grows as $\Theta\left(\lambda_1(A)^L\right)$ asymptotically with L. Hence it vanishes exponentially quickly if $\lambda_1(A) < 1$ and it grows exponentially quickly if $\lambda_1(A) > 1$.

When it comes to gradients, these two cases correspond to the *vanishing gradients problem* and the *exploding gradients problem*, respectively. Both result in a failure case for gradient-based optimization methods. Huge gradients are numerically unstable. Tiny gradients preclude progress and stall any gradient-based optimization method. We can always avoid one of the problems by scaling the weights, but avoiding both can be delicate.

Vanishing and exploding gradients are not the fault of an optimization method. They are a property of the model architecture. As a result, to avoid them we need to engineer our model architectures carefully. Many architecture innovations in deep learning aim to address this problem. We will discuss two of them. The first are residual connections, and the second are layer normalizations.

Residual connections

The basic idea behind *residual networks* is to make each layer close to the identity map. Traditional building blocks of neural networks typically looked like two affine transformations A, B, with a nonlinearity in the middle:

$$f(x) = B(\text{ReLU}(A(x)))$$

Residual networks modify these building blocks by adding the input x back to the output:

$$h(x) = x + B(\text{ReLU}(A(x)))$$

In cases where the output of C differs in its dimension from x, practitioners use different padding or truncation schemes to match dimensions. Thinking about the computation graph, we create a connection from the input to the output of the building block that *skips* the transformation. Such connections are therefore called *skip connections*.

This seemingly innocuous change was hugely successful. Residual networks took the computer vision community by storm, after they achieved leading performance in numerous benchmarks upon their release in 2015. These networks seemed to avoid the vanishing gradients better than prior architectures and allowed for model depths not seen before.

Let's begin to get some intuition for why residual layers are reasonable by thinking about what they do to the gradient computation. Let $J = \mathsf{D}_x f(x)$ be the Jacobian of the function f with respect to its input. We can verify that the Jacobian of the residual block looks like

$$J' = \mathsf{D}_x h(x) = \mathsf{D}_x I(x) + \mathsf{D}_x f(x) = I + J\,.$$

In other words, the Jacobian of a residual block is the Jacobian of a regular block plus the identity. This means that if we scale down the weights of the regular block, the Jacobian J' approaches the identity in the sense that all its singular values are between $1 - \epsilon$ and $1 + \epsilon$. We can think of such a transformation as locally well-conditioned. It neither blows up nor shrinks down the gradient much. Since the full Jacobian of a deep residual network will be a product of such matrices, our reasoning suggests that suitably scaled residual networks have well-conditioned Jacobians, and hence, as we discussed above, predictions should converge rapidly.

Our analyses of non-convex ERM thus far have described scenarios where we could prove the *predictions* converge to the training labels. Residual networks are interesting as we can construct cases where *the weights* converge to a unique optimal solution. This perhaps gives even further motivation for their use.

Let's consider the simple case of where the activation function is the identity. The resulting residual blocks are linear transformations of the form $I + A$. We can chain them as $A = (I + A_L) \cdots (I + A_1)$. Such networks are no longer universal approximators, but they are non-convex parameterizations of linear functions. We can turn this parameterization into an optimization problem by solving a least squares regression problem in this residual parameterization:

$$\text{minimize}_{A_1,\ldots,A_L} \quad \mathbb{E}[\tfrac{1}{2}\|A(X) - Y\|^2]$$

Assume X is a random vector with covariance matrix I and $Y = B(X) + G$ where B is a linear transformation and G is centered random Gaussian

noise. A standard trick shows that up to an additive constant the objective function is equal to

$$f(A_1, \ldots, A_L) = \frac{1}{2}\|A - B\|_F^2 .$$

What can we say about the gradients of this function? We can verify that the Jacobian of f with respect to A_i equals

$$\mathsf{D}_{A_i} f(A_1, \ldots, A_L) = P^T E Q^T$$

where $P = (I + A_L) \cdots (I + A_{i+1})$ and $Q = (I + A_{i-1}) \cdots (I + A_1)$.

Note that when P and Q are non-singular, then the gradient can only vanish at $E = 0$. But that means we're at the global minimum of the objective function f. We can ensure that P and Q are non-singular by making the largest singular value of each A_i to be less than $1/L$.

The property we find here is that the gradient vanishes only at the optimum. This is implied by convexity, but it does not imply convexity. Indeed, the objective function above is not convex. However, this weaker property is enough to ensure that gradient-based methods do not get stuck except at the optimum. In particular, the objective has no saddle points. This desirable property does not hold for the standard parameterization $A = A_L \cdots A_1$ and so it speaks to the benefit of the residual parameterization.

Normalization

Consider a feature vector x. We can partition this vector into subsets so that

$$x = \begin{bmatrix} x_1 & \cdots & x_P \end{bmatrix} ,$$

and *normalize* each subset to yield a vector with partitions

$$x_i' = 1/s_i(x_i - \mu_i)$$

where μ_i is the mean of the components of x_i and s_i is the standard deviation.

Such normalization schemes have proven powerful for accelerating the convergence of stochastic gradient descent. In particular, it is clear why such an operation can improve the conditioning of the Jacobian. Consider the simple linear case where $\widehat{y} = Xw$. Then $\mathsf{D}(w) = X$. If each row of X has a large mean, i.e., $X \approx X_0 + c11^T$, then, X may be ill conditioned, as the rank one term will dominate first singular value, and the remaining singular values may be small. Removing the mean improves the condition number. Rescaling by the variance may improve the condition number, but also has the benefit of avoiding numerical scaling issues, forcing each layer in a neural network to be on the same scale.

Such whole vector operations are expensive. Normalization in deep learning chooses parts of the vector to normalize that can be computed quickly. Batch Normalization normalizes along the data-dimension in batches of data used to as stochastic gradient minibatches.[8] Group Normalization generalizes this notion to arbitrary partitioning of the data, encompassing a variety of normalization proposals.[9] The best normalization scheme for a particular task is problem dependent, but there is a great deal of flexibility in how one partitions features for normalization.

Generalization in deep learning

While our understanding of optimization of deep neural networks has made significant progress, our understanding of generalization is considerably less settled. In the previous chapter, we highlighted four paths towards generalization: stability, capacity, margin, optimization. It's plausible deep neural networks have elements of all four of these core components. The evidence is not as cut and dry as it is for linear models, but some mathematical progress has been made to understand how deep learning leverages classic foundations of generalization. In this section we review the partial evidence gathered so far.

In the next chapter we will extend this discussion by taking a closer look at the role that data and community practices play in the study of generalization for deep learning.

Algorithmic stability of deep neural networks

We discussed how stochastic gradient descent trained on convex models was algorithmically stable. The results we saw in Chapter 6 extend to some to the non-convex case. However, results that go through uniform stability ultimately cannot explain the generalization performance of training large deep models. The reason is that uniform stability does not depend on the data generating distribution. In fact, uniform stability is invariant under changes to the labeling of the data. But we saw in Chapter 6 that we can make the generalization gap whatever we want by randomizing the labels in the training set.

Nevertheless, it's possible that a weaker notion of stability that is sensitive to the data would work. In fact, it does appear to be the case in experiment that stochastic gradient descent is relatively stable in concrete instances. To measure stability empirically we can look at the Euclidean distance between the parameters of two identical models trained on the datasets which differ only by a single example. If these are close for many

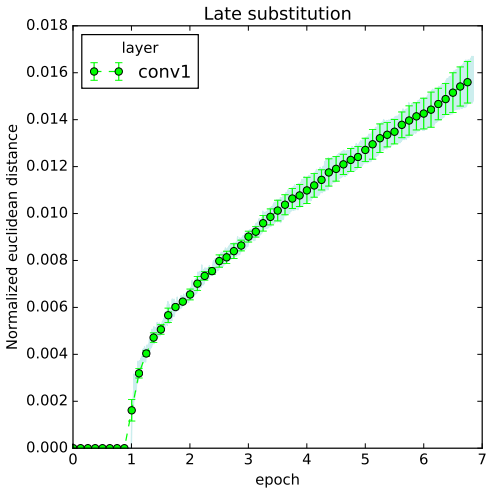

Figure 1: Parameter divergence on AlexNet trained on ImageNet. The two models differ only in a single example.

independent runs, then the algorithm appears to be stable. Note that parameter distance is a *stronger* notion than stability of the loss.

The plot below displays the parameter distance for the venerable AlexNet model trained on the ImageNet benchmark. We observe that the parameter distance grows sub-linearly even though our theoretical analysis is unable to prove this to be true.

Capacity of deep neural networks

Many researchers have attempted to compute notions like VC-dimension or Rademacher complexity of neural networks. The earliest work by Baum and Haussler bounded the VC-dimension of small neural networks and showed that as long these networks could be optimized well in the *underparameterized* regime, then the neural networks would generalize.[10] Later, seminal work by Bartlett showed that the size of the weights was more important than the number of weights in terms of understanding generalization capabilities of neural networks.[11] These bounds were later sharpened using Rademacher complexity arguments.[12]

Margin bounds for deep neural networks

The margin theory for linear models conceptually extends to neural networks. The definition of margin is unchanged. It simply quantifies how close the network is to making an incorrect prediction. What changes is that

for multi-layer neural networks the choice of a suitable norm is substantially more delicate.

To see why, a little bit of notation is necessary. We consider multi-layer neural networks specified by a composition of L layers. Each layer is a linear transformation of the input, followed by a coordinate-wise non-linear map:

$$\text{Input } x \to Ax \to \sigma(Ax)$$

The linear transformation has trainable parameters, while the non-linear map does not. For notational simplicity, we assume we have the same non-linearity σ at each layer scaled so that the map is 1-Lipschitz. For example, the popular coordinate-wise ReLU $\max\{x, 0\}$ operation satisfies this assumption.

Given L weight matrices $\mathcal{A} = (A_1, \ldots, A_L)$ let $f_{\mathcal{A}} \colon \mathbb{R}^d \to \mathbb{R}^k$ denote the function computed by the corresponding network:

$$f_{\mathcal{A}}(x) := A_L \sigma(A_{L-1} \cdots \sigma(A_1 x) \cdots)) \,.$$

The network output $F_{\mathcal{A}}(x) \in \mathbb{R}^k$ is converted to a class label in $\{1, \ldots, k\}$ by taking the arg max over components, with an arbitrary rule for breaking ties. We assume $d \geq k$ only for notational convenience.

Our goal now is to define a complexity measure of the neural network that will allow us to prove a margin bound. Recall that margins are meaningless without a suitable normalization of the network. Convince yourself that the Euclidean norm can no longer work for multi-layer ReLU networks. After all we can scale the linear transformation on one later by a constant c and a subsequent layer by a constant $1/c$. Since the ReLU non-linearity is piecewise linear, this transformation changes the Euclidean norm of the weights arbitrarily without changing the function that the network computes.

There's much ongoing work about what good norms are for deep neural networks. We will introduce one possible choice here.

Let $\|\cdot\|_{\text{op}}$ denote the spectral norm. Also, let $\|A\|_{2,1} = \|(\|A_{:,1}\|_2, \ldots, \|A_{:,m}\|_2)\|_1$ the matrix norm where we apply the ℓ_2-norm to each column of the matrix and then take the ℓ_1-norm of the resulting vector.

The *spectral complexity* $R_{\mathcal{A}}$ of a network $F_{\mathcal{A}}$ with weights $\mathcal{A}$ is the defined as

$$R_{\mathcal{A}} := \left(\prod_{i=1}^{L} \|A_i\|_{\text{op}} \right) \left(\sum_{i=1}^{L} \left(\frac{\|A_i^\top - M_i^\top\|_{2,1}}{\|A_i\|_{\text{op}}} \right)^{2/3} \right)^{3/2} . \tag{1}$$

Here, the matrices $M_1, \ldots, M_L$ are free parameters that we can choose to minimize the bound. Random matrices tend to be good choices.

The following theorem provides a generalization bound for neural networks with fixed nonlinearities and weight matrices $\mathcal{A}$ of bounded spectral complexity $R_{\mathcal{A}}$.

Theorem 1. *Assume data $(x_1, y_1), \ldots, (x_n, y_n)$ are drawn i.i.d. from any probability distribution over $\mathbb{R}^d \times \{1, \ldots, k\}$. With probability at least $1 - \delta$, for every margin $\theta > 0$ and every network $f_{\mathcal{A}} \colon \mathbb{R}^d \to \mathbb{R}^k$,*

$$R[f_{\mathcal{A}}] - R_S^{\theta}[f_{\mathcal{A}}] \leq \widetilde{\mathcal{O}}\left(\frac{R_{\mathcal{A}}\sqrt{\sum_i \|x_i\|_2^2}\ln(d)}{\theta n} + \sqrt{\frac{\ln(1/\delta)}{n}}\right),$$

where $R_S^{\theta}[f] \leq n^{-1}\sum_i \mathbf{1}\left[f(x_i)_{y_i} \leq \theta + \max_{j \neq y_i} f(x_i)_j\right]$.

The proof of the theorem involves Rademacher complexity and so-called data-dependent covering arguments. Although it can be shown empirically that the above complexity measure $R_{\mathcal{A}}$ is somewhat correlated with generalization performance in some cases, there is no reason to believe that it is the "right" complexity measure. The bound has other undesirable properties, such as an exponential dependence on the depth of the network as well as an implicit dependence on the size of the network.

Implicit regularization by stochastic gradient descent in deep learning

There have been recent attempts to understand the dynamics of stochastic gradient descent on deep neural networks. Using an argument similar to the one we used to reason about convergence of the *predictions* of deep neural networks, Jacot et al. used a differential equations argument to understand to which *function* stochastic gradient converged.[13]

Here we can sketch the rough details of their argument. Recall our expression for the dynamics of the predictions in gradient descent:

$$\widehat{y}_{t+1} - y = (I - \alpha \mathsf{D}(w_t)^T \mathsf{D}(w_t))(\widehat{y}_t - y) + \alpha\epsilon_t\,.$$

If α is very small, we can further approximate this as

$$\widehat{y}_{t+1} - y = (I - \alpha \mathsf{D}(w_0)^T \mathsf{D}(w_0))(\widehat{y}_t - y) + \alpha\epsilon'_t\,.$$

Now ϵ'_t captures both the curvature of the function representation and the deviation of the weights from their initial condition. Note that in this case, $\mathsf{D}(w_0)$ is a constant for all time. The matrix $\mathsf{D}(w_0)^T\mathsf{D}(w_0)$ is $n \times n$, positive semidefinite, and only depends on the data and the initial setting of the weights. When the weights are random, this is a kernel induced by

random features. The expected value of this random feature embedding is called a *Neural Tangent Kernel*.

$$k(x_1, x_2) = \mathop{\mathbb{E}}_{w_0} \left[\langle \nabla_w f(x_1; w_0), \nabla_w f(x_2; w_0) \rangle \right] .$$

Using a limit where α tends to zero, Jacot et. al argue that a deep neural net will find a minimum norm solution in the RKHS of the Neural Tangent Kernel. This argument was made non-asymptotic by Heckel and Soltanolkotabi.[14] In the generalization chapter, we showed that the minimum norm solution of in RKHS generalized with a rate of $O(1/n)$. Hence, this argument suggests a similar rate of generalization may occur in deep learning, provided the norm of the minimum norm solution does not grow to rapidly with the number of data points and that the true function optimized by stochastic gradient descent isn't too dissimilar from the Neural Tangent Kernel limit. We note that this argument is qualitative, and there remains work to be done to make these arguments fully rigorous.

Perhaps the most glaring issue with NTK arguments is that they do not reflect practice. Models trained with Neural Tangent Kernels do not match the predictive performance of the corresponding neural network. Moreover, simpler kernels inspired by these networks can out perform Neural Tangent Kernels.[15] There is a significant gap between the theory and practice here, but the research is new and remains active and this gap may be narrowed in the coming years.

Chapter notes

Deep learning is at this point a vast field with tens of thousands of papers. We don't attempt to survey or systematize this vast literature. It's worth emphasizing though the importance of learning about the area by experimenting with available code on publicly available datasets. The next chapter will cover datasets and benchmarks in detail.

Apart from the development of benchmark datasets, one of the most important advances in deep learning over the last decade has been the development of high quality, open source software. This software makes it easier than ever to prototype deep neural network models. One such open source project is PyTorch (`pytorch.org`), which we recommend for the researcher interested in experimenting with deep neural networks. The best way to begin to understand some of the nuances of architecture selection is to find pre-existing code and understand how it is composed. We recommend the tutorial by David Page which demonstrates how the many different pieces fit together in a deep learning pipeline.[16]

For a more in depth understanding of automatic differentiation, we recommend Griewank and Walther's *Evaluating Derivatives*.[17] This text works through a variety of unexpected scenarios—such as implicitly defined functions—where gradients can be computed algorithmically. Jax is open source automatic differentiation package that incorporates many of these techniques, and is useful for any application that could be assisted by automatic differentiation.[18]

The AlexNet architecture was introduced by Krizhevsky, Sutskever, and Hinton in 2012.[19] Achieving the best known performance on the ImageNet benchmark at the time, it was pivotal in kicking of the most recent wave of research on deep learning.

Residual networks were introduced by He, Zhang, Ren, and Sun in 2016.[20] The observation about linear residual networks is due to Hardt and Ma.[21]

Theorem 1 is due to Bartlett, Foster, and Telgarsky.[22] The dimension dependence of the theorem can be removed.[23]

Bibliography

[1] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[2] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition*, 2005.

[3] Pedro F. Felzenszwalb, Ross B. Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2009.

[4] Amit Daniely, Roy Frostig, and Yoram Singer. Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity. In *Advances in Neural Information Processing Systems*, 2016.

[5] Peter Auer, Mark Herbster, and Manfred K. Warmuth. Exponentially many local minima for single neurons. In *Advances in Neural Information Processing Systems*, 1996.

[6] Van H. Vu. On the infeasibility of training neural networks with small mean-squared error. *Transactions on Information Theory*, 44(7):2892–2900, 1998.

[7] Surbhi Goel, Adam Klivans, Pasin Manurangsi, and Daniel Reichman. Tight hardness results for training depth-2 ReLU networks. In *Innovations in Theoretical Computer Science*, 2021.

[8] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456. PMLR, 2015.

[9] Yuxin Wu and Kaiming He. Group normalization. In *European Conference on Computer Vision*, pages 3–19, 2018.

[10] Eric B. Baum and David Haussler. What size net gives valid generalization? In *Advances in Neural Information Processing Systems*, 1988.

[11] Peter L. Bartlett. The sample complexity of pattern classification with neural networks: The size of the weights is more important than the size of the network. *IEEE Transactions on Information Theory*, 44(2):525–536, 1998.

[12] Peter L. Bartlett and Shahar Mendelson. Rademacher and Gaussian complexities: Risk bounds and structural results. *Journal of Machine Learning Research*, 3(Nov):463–482, 2002.

[13] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in Neural Information Processing Systems*, pages 8580–8589, 2018.

[14] Reinhard Heckel and Mahdi Soltanolkotabi. Compressive sensing with untrained neural networks: Gradient descent finds a smooth approximation. In *International Conference on Machine Learning*, pages 4149–4158. PMLR, 2020.

[15] Vaishaal Shankar, Alex Fang, Wenshuo Guo, Sara Fridovich-Keil, Jonathan Ragan-Kelley, Ludwig Schmidt, and Benjamin Recht. Neural kernels without tangents. In *International Conference on Machine Learning*, 2020.

[16] David Page. `https://myrtle.ai/learn/how-to-train-your-resnet/`, 2020.

[17] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2nd edition, 2008.

[18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Composable transformations of Python+NumPy programs, 2018.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Computer Vision and Pattern Recognition*, 2016.

[21] Moritz Hardt and Tengyu Ma. Identity matters in deep learning. In *International Conference on Learning Representations*, 2017.

[22] Peter L. Bartlett, Dylan J. Foster, and Matus J. Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pages 6240–6249, 2017.

[23] Noah Golowich, Alexander Rakhlin, and Ohad Shamir. Size-independent sample complexity of neural networks. In *Conference on Learning Theory*, pages 297–299, 2018.