18 Kernel Methods

If you want the kernel, you must break the shell.
—Meister Eckhart

18.1 Kernels

So far in all of our methods we have been given data $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in a fixed-dimensional space $\mathbb{R}^d$. But not all data is easily representable in such a space. Consider, for example, linguistic data—sentences and words come in varying lengths and contain vaying amounts of information. Similarly, genomic data may come in pieces of varying length. If the data consist of molecular structures, these may have important geometric properties that vary widely from one molecule to another. In all these cases it is not so clear how to represent these in a finite-dimensional space $\mathbb{R}^d$. Moreover, as we have already seen in the sections on dimension reduction, even if the data arrive as points in $\mathbb{R}^d$, they may be better represented in a different space of another dimension. Surprisingly, sometimes increasing the dimension of the representation can also be very useful.

But not all machine learning techniques require us to know the correct embedding of the data in $\mathbb{R}^d$. Instead some methods only require that we have a notion of an inner product or maybe only a notion of distance between the data points.

For example, k-nearest neighbor classification only requires knowing the distance between the various data points—not the actual location of those points. Similarly, the dimension reduction methods t-SNE and UMAP do not require us to know the location of the points $\mathbf{x}_i$ in space—just the distances between them. That means that if we have a good definition of distance, we can still use t-SNE or UMAP to define an embedding of the points $\mathbf{x}_1, \ldots \mathbf{x}_N$ into $\mathbb{R}^s$ without using anything about an original space where the $\mathbf{x}_i$ might live. Similarly, the spectral clustering methods of the previous section only require that we know the distance between points in order to construct a weighted graph.

Linear regression involves estimating $\boldsymbol{\beta}$ by minimizing the loss function

$$\frac{1}{N}\sum_{i=1}^{N}\mathscr{L}(\hat{f},\mathbf{x}_i,y_i)=\frac{1}{N}\sum_{i=1}^{N}(\hat{f}(\mathbf{x}_i)-y_i)^2=\frac{1}{N}\sum_{i=1}^{N}(\langle\widehat{\boldsymbol{\beta}},\mathbf{x}_i\rangle-y_i)^2.$$

So, if I know how to compute the inner products $\langle\widehat{\boldsymbol{\beta}},\mathbf{x}_i\rangle$, I don't need to know anything about the embedding of $\mathbf{x}_1,\dots,\mathbf{x}_N$ in $\mathbb{R}^d$. The L^2-regularization term $\|\widehat{\boldsymbol{\beta}}\|_2^2=\left\langle\widehat{\boldsymbol{\beta}},\widehat{\boldsymbol{\beta}}\right\rangle$ is also an inner product, so regularization can also be done without knowing an embedding in $\mathbb{R}^d$.

Even using PCA for dimension-reduction boils down to computing eigenvectors of the matrix $\mathbb{X}^\mathsf{T}\mathbb{X}$, but this matrix can be expressed solely in terms of inner products of pairs $\mathbf{x}_i,\mathbf{x}_j$ of data points (as we show later). So without knowing the embedding of the $\mathbf{x}_i$ in $\mathbb{R}^d$, if we just know all the inner products, then we can still find the necessary decomposition and do PCA.

18.1.1 Kernels

The general strategy we want to explore in this section is the idea of mapping the space $\mathscr{X}$ of data points (which might not be an inner product space or normed space) to some (possibly infinite-dimensional) inner product space $\mathscr{H}$ with a map $\varphi:\mathscr{X}\to\mathscr{H}$ and then computing inner products $\langle\varphi(\mathbf{x}),\varphi(\mathbf{x}'))\rangle$ and distances $\|\varphi(\mathbf{x})-\varphi(\mathbf{x}')\|$ between the resulting points.

A *kernel function*[50] is a way to make use of this idea without necessarily explicitly identifying the inner product space $\mathscr{H}$ or the map $\varphi:\mathscr{X}\to\mathscr{H}$.

Definition 18.1.1. *For a set $\mathscr{X}$, a* kernel function *is a function $\kappa:\mathscr{X}\times\mathscr{X}\to\mathbb{R}$. A kernel function κ is* symmetric *if $\kappa(\mathbf{x},\mathbf{x}')=\kappa(\mathbf{x}',\mathbf{x})$ for any $\mathbf{x},\mathbf{x}'\in\mathscr{X}$.*

Example 18.1.2. A very important type of kernel is constructed by taking a map $\varphi:\mathscr{X}\to\mathscr{H}$ to a real inner product space φH and defining a kernel of the form

$$\kappa(\mathbf{x},\mathbf{x}')=\langle\varphi(\mathbf{x}),\varphi(\mathbf{x}')\rangle\,.$$

This is the prototypical example of a symmetric kernel. Note that, unlike an inner product on $\mathscr{X}$, this kernel is not bilinear unless the map φ is a linear transformation.

[50] The use of the name *kernel* in mathematics is a hot mess. Despite their name, the kernels here have nothing to do with the kernels of linear transformations or the kernels in Definition 6.1.1. There are several other mathematical objects also called *kernels*. So whenever anyone uses the word *kernel*, make them define exactly what they mean.

Although this kernel is a very important example, it may be computationally expensive to compute the values $\varphi(\mathbf{x})$ and $\varphi(\mathbf{x}')$ and then compute their inner product, especially if $\mathscr{H}$ is high dimensional. One advantage of many other kernels is that they can be much easier or cheaper to compute than this one, and even for kernels of this special form, there are often much more efficient ways to compute the value of $\kappa(\mathbf{x}, \mathbf{x}')$ without explicitly evaluating the map φ.

Example 18.1.3. If $\mathscr{X}$ is the set of all possible strings (of characters), then one possible kernel to use to compare two strings $\mathbf{x}$ and $\mathbf{x}'$ is the *edit distance*, which is the minimum number of operations needed to change $\mathbf{x}$ into $\mathbf{x}'$.

For example if the allowable operations are deletion, insertion, or substitution, then the edit distance between *kitten* and *sitting* is 3:

(i) kitten $\mapsto$ sitten (substitute s for k).

(ii) sitten $\mapsto$ sittin (substitute i for e).

(iii) sittin $\mapsto$ sitting (append g at the end).

The kernel defined by edit distance is symmetric because doing the inverse of each operation in reverse order gives the same number of edits to go from $\mathbf{x}'$ to $\mathbf{x}$ as it takes to go from $\mathbf{x}$ to $\mathbf{x}'$.

Example 18.1.4 (Kernels for comparing documents). The *bag-of-words* model is useful for studying documents. Given a vocabulary of d words (sorted in some order), we map a document to the vector $\mathbf{x} = (x_1, \dots, x_d) \in \mathbb{R}^d$, where x_ℓ is the number of times that word ℓ occurs in the document. A common choice of kernel for this model is *cosine similarity*

$$\kappa(\mathbf{x}, \mathbf{x}') = \frac{\mathbf{x}^\mathsf{T}\mathbf{x}'}{\|\mathbf{x}\|_2 \|\mathbf{x}'\|_2}$$

where $\mathbf{x}$ and $\mathbf{x}'$ are the bag-of-words representations of the two documents. This kernel is just the cosine of the angle between the two embedded documents in $\mathbb{R}^d$. Since $\mathbf{x}$ and $\mathbf{x}'$ are always nonnegative, the angle between them lies in the interval $[0, \pi/2]$ and the kernel always lies in $[0, 1]$.

This kernel doesn't work very well as is, because it is skewed by common words that are repeated many times in both documents. To fix this problem, remove all the stop words (see Section 16.5.6) and apply the *term frequency, inverse document frequency (tf-idf)* transformation (also from Section 16.5.6) to the data before computing the cosine similarity. Thus, for any document $\mathbf{x} = (x_1, \dots, x_d)$ we let

$$\varphi(\mathbf{x}) = (\mathrm{tf}(x_1)\,\mathrm{idf}(1), \mathrm{tf}(x_2)\,\mathrm{idf}(2), \dots, \mathrm{tf}(x_d)\,\mathrm{idf}(d))$$

The value of $\mathrm{tf}(x_j)$ grows more slowly than x_j does, which reduces the effect of words that are repeated many times in one document. And $\mathrm{idf}(j)$ is smaller for words that appear in more documents, so multiplying each term by the corresponding $\mathrm{idf}(j)$ reduces the impact of words that appear in many documents.

All this comes together to give the following improved kernel for identifying similar documents:

$$\kappa(\mathbf{x}, \mathbf{x}') = \frac{\varphi(\mathbf{x})^\mathsf{T} \varphi(\mathbf{x}')}{\|\varphi(\mathbf{x})\| \|\varphi(\mathbf{x}')\|}.$$

18.1.2 Mercer Kernels

Definition 18.1.5. *Given any finite subset $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathscr{X}$ and a kernel $\kappa : \mathscr{X} \times \mathscr{X} \to \mathbb{R}$, the matrix K whose (i,j)-entry is $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ is called the* Gram matrix *of the kernel κ for the subset. A symmetric kernel κ is called a* Mercer kernel *if for any finite subset $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathscr{X}$ the corresponding Gram matrix is positive semidefinite.*

Remark 18.1.6. Mercer kernels are also called *positive definite* kernels. This is a confusing name because the condition for being a Mercer kernel is that all the Gram matrices must be positive semidefinite, not positive definite. To avoid confusion, we prefer to use the name *Mercer* for these kernels.

Example 18.1.7. The kernels defined in Example 18.1.2 are always Mercer kernels. The proof of this follows from the properties of inner products.

Example 18.1.8. For $\mathscr{X} = \mathbb{R}^d$ there are many choices of kernel. Here are two common ones.

(i) Gaussian kernel:

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^\mathsf{T} \Sigma^{-1} (\mathbf{x} - \mathbf{x}')\right) \tag{18.1}$$

for some choice $\Sigma > 0$ of $d \times d$ matrix.

(ii) Polynomial kernel:

$$\kappa(\mathbf{x}, \mathbf{x}') = (\gamma \mathbf{x}^\mathsf{T} \mathbf{x}' + r)^\delta \tag{18.2}$$

for some choice of $r, \gamma \in \mathbb{R}$, with $\gamma > 0$, $r > 0$, and some integer $\delta > 0$.

Both of these kernels are clearly symmetric. They are also both Mercer, although that is harder to check.

Proposition 18.1.9. *If $\kappa : \mathscr{X} \times \mathscr{X} \to \mathbb{R}$ is a Mercer kernel, then the following hold for all $\mathbf{x}, \mathbf{x}' \in \mathscr{X}$:*

(*i*) *$\kappa(\mathbf{x}, \mathbf{x}) \geq 0$ for all $\mathbf{x} \in \mathscr{X}$.*

(*ii*) *(Cauchy-Schwarz)*
$$\kappa(\mathbf{x}, \mathbf{x}')^2 \leq \kappa(\mathbf{x}, \mathbf{x})\kappa(\mathbf{x}', \mathbf{x}').$$

Proof. The proof is Exercise 18.2. □

Proposition 18.1.10. *Let $\kappa_1, \ldots, \kappa_m$ be Mercer kernels on a set $\mathscr{X}$, and let $\alpha_1, \ldots, \alpha_m$ be nonnegative constants. The following are also Mercer kernels:*

(*i*) *Nonnegative combinations: $\sum_{i=1}^m \alpha_i \kappa_i$.*

(*ii*) *Products: $\prod_{i=1}^m \kappa_i$*

(*iii*) *Pullback: $\xi(f(\mathbf{x}), f(\mathbf{x}'))$ for any Mercer kernel $\xi : \mathscr{Y} \to \mathbb{R}$ and any function $f : \mathscr{X} \to \mathscr{X}$.*

Mercer kernels are important because they can all be rewritten in the form of Example 18.1.2, thanks to the following theorem.

Theorem 18.1.11 (Mercer's Theorem). *If $\mathscr{X}$ is a compact set and $\kappa : \mathscr{X} \times \mathscr{X} \to \mathbb{R}$ is a continuous Mercer kernel, then there exists an inner product space $(V, \langle , \rangle)$ and a continuous map $\varphi : \mathscr{X} \to V$ such that $\kappa(\mathbf{x}, \mathbf{x}') = \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle$ for all $\mathbf{x}, \mathbf{x}' \in \mathscr{X}$.*

Proof. We prove the special case when $\mathscr{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ is finite. For a proof of Mercer's theorem in the infinite case see [MRT18, Section 6.2].

Let $K = [k_{ij}]$ be the Gram matrix with $k_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$. Since K is positive semidefinite, it can be orthonormally diagonalized to
$$K = U^\mathsf{T} \Lambda U,$$
with Λ diagonal and U orthonormal. Since Λ is diagonal with nonnegative eigenvalues, it has a unique nonnegative square root $\Lambda^{\frac{1}{2}}$ and this gives
$$K = (\Lambda^{\frac{1}{2}} U)^\mathsf{T} (\Lambda^{\frac{1}{2}} U).$$
Denote the ith column of the matrix U by $U_{:,i}$. The (i,j)-entry k_{ij} of K is, therefore, equal to
$$k_{ij} = (\Lambda^{\frac{1}{2}} U_{:,i})^\mathsf{T} (\Lambda^{\frac{1}{2}} U_{:,j}) \tag{18.3}$$
Define a map $\varphi : \mathscr{X} \to \mathbb{R}^n$ by
$$\varphi(\mathbf{x}_i) = \Lambda^{\frac{1}{2}} U_{:,i}$$
Combining this with (18.3) shows that for every i, j we have
$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = k_{ij} = (\Lambda^{\frac{1}{2}} U_{:,i})^\mathsf{T} (\Lambda^{\frac{1}{2}} U_{:,j}) = \varphi(\mathbf{x}_i)^\mathsf{T} \varphi(\mathbf{x}_j),$$

as required. □

Remark 18.1.12. If $\mathscr{X}$ is infinite (for example, if $\mathscr{X} \subset \mathbb{R}^d$), the inner product space V guaranteed by the theorem is not necessarily finite dimensional.

Example 18.1.13. The kernel $\kappa : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ given by $\kappa(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\mathsf{T}\mathbf{x}')^2$ is a special case of the polynomial kernel, so it is Mercer. Mercer's theorem guarantees it is of the form $\langle \varphi(\mathbf{x}_i), \varphi(\mathbf{x}') \rangle$ for some map φ. In many cases it is not easy to write down the map φ, but for this kernel we can do it as follows:

$$\begin{aligned}\kappa(\mathbf{x}, \mathbf{x}') &= (x_1 x_1' + x_2 x_2')^2 \\ &= x_1^2 (x_1')^2 + x_2^2 (x_2')^2 + 2 x_1 x_1' x_2 x_2'\end{aligned}$$

Setting $\varphi : \mathbb{R}^2 \to \mathbb{R}^3$ to be $\varphi(\mathbf{x}) = (x_1^2, x_2^2, \sqrt{2} x_1 x_2)$ gives

$$\begin{aligned}\langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle &= (x_1^2, x_2^2, \sqrt{2} x_1 x_2)^\mathsf{T} ((x_1')^2, (x_2')^2, \sqrt{2} x_1' x_2') \\ &= x_1^2 (x_1')^2 + x_2^2 (x_2')^2 + 2 x_1 x_1' x_2 x_2',\end{aligned}$$

as required.

Remark 18.1.14. The embedding of $\mathbb{R}^d$ that corresponds to the polynomial kernel (18.2) of degree δ requires all the monomials $(x_1^\delta, x_1^{\delta-1} x_2, \ldots, x_d^\delta)$ of degree δ. Since there are $\binom{\delta+d}{d}$ monomials of degree δ in d variables, this means that φ maps to $\mathbb{R}^{\binom{\delta+d}{d}}$, which can be very large, but the kernel (18.2) is not computationally expensive to evaluate.

18.1.3 Kernels for High-Dimensional Computation

As mentioned in above, in any computation that involves inner products, we can replace every inner product with a kernel. This allows us to compute things like K-nearest neighbors or a t-SNE or UMAP embedding on data that does not have a natural embedding into $\mathbb{R}^d$. But kernels are also useful for data that already is embedded in $\mathbb{R}^d$ because they allow effiient computation of inner products in higher-dimensional space.

In general, the target V of the map $\varphi : \mathbb{R}^d \to V$ guaranteed by Mercer's theorem can have a very large dimension. For example, in the case of the polynomial kernel $(\gamma \mathbf{x}^\mathsf{T}\mathbf{x}' + r)^\delta$, the dimension of V is the number of monomials in $x_1, \ldots, x_n$ of degree at most δ, which is $\binom{n+\delta-1}{n-1}$. If $n = \delta = 7$, this is $\binom{13}{6} = 1716$. If we just need the kernel and not the map φ, then it is much more efficient to compute $(\gamma \mathbf{x}^\mathsf{T}\mathbf{x}' + r)^7$ than to apply the map φ and then compute the inner product in 1716-dimensional space. Moreover, it can be shown that the space V for the Gaussian kernel must be infinite dimensional, so it is not really practical to use the map φ, and yet the value of the kernel κ is easy to compute.

It may seem strange to want to move data to a higher-dimensional space, especially after we have gone to such work in previous chapters to reduce dimensions. However, if done well, increasing the dimension of an embedding can significantly improve the effectiveness of machine learning techniques in many circumstances. A substantial benefit of using kernels is that they allow us to reap the benefits of working in high-dimensional space without all of the computational costs. We discuss these ideas in more depth in subsequent sections.

18.2 The Kernel Trick and Support Vector Machines

Logistic regression and many other methods try to separate binary data using a single hyperplane. In low-dimensional space this can be difficult or impossible to do well, but it often becomes much easier in higher-dimensional space, as shown in Figure 18.1

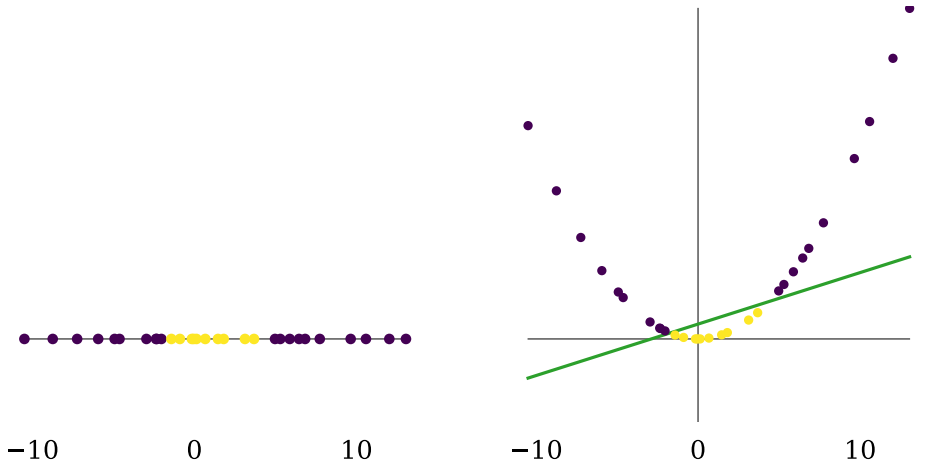


Figure 18.1: Embedding a data set into a higher-dimensional space with a nonlinear map can make it easier to separate points with a hyperplane. The yellow and purple points in the one-dimensional data set in the left panel cannot be separated by a hyperplane, but mapping it to two-dimensional space via the function $\varphi(x) = (x, x^2)$ (right panel) makes the set linearly separable by the green hyperplane.

Unfortunately, working with a logistic regression model on data in high-dimensional space can be computationally expensive. To get the benefits of the high-dimensional space without the computational cost of working in that space, we use a classifier for which all the computations in high-dimensional space can be expressed in terms of a cheaply computable kernel. This is called the *kernel trick*.

This is an important idea worth restating: If everything we want to evaluate can be written as an inner product, then we need not bother with φ at all and can just use the kernel. For a data set $\mathbf{D}$ consisting of points $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^d$, if all computations of the objective function and constraints can be expressed in terms of inner products $\mathbf{x}_i^\mathsf{T}\mathbf{x}_j$, and is not dependent on other functions of $\mathbf{x}_i$ or $\mathbf{x}_j$, then instead of mapping the data to a much higher-dimensional space via a map φ and computing the inner products there, simply replace inner products $\mathbf{x}_i^\mathsf{T}\mathbf{x}_j$ with kernel evaluations $\kappa(\mathbf{x}_i, \mathbf{x}_j)$, which are usually inexpensive to evaluate.

18.2.1 Hard-margin Classifiers

Two linear classifiers that led themselves well to the kernel trick are the *hard-margin classifier* and *soft-margin classifier*. Unfortunately these classifiers don't have a natural interpretation in terms of probability, but because they are well suited to the kernel trick, they are very useful.

Consider a binary classifier of the form $y = \text{sign}(\mathbf{w}^\mathsf{T}\mathbf{x} + b)$, giving values of either 1 or -1, where $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$. Assume that the data set $\mathbf{D} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, y_N)\}$ is linearly separable, meaning that there exists a hyperplane H of the form $H = \{\mathbf{x}' \mid \mathbf{w}^\mathsf{T}\mathbf{x}' + b = 0\}$ in $\mathbb{R}^d$ such that $y_i = -1$ if and only if $\mathbf{w}^\mathsf{T}\mathbf{x}_i + b < 0$. There may be many choices of $\mathbf{w}$ that linearly separate the classes in the data set. The *hard-margin classifier* chooses the values of b and $\mathbf{w}$ that maximize the distance from the closest point $\mathbf{x}_i$ to the decision boundary H. The points $\mathbf{x}_i$ that are closest to H are called the *support vectors* of the classifier.

Proposition 18.2.1. *For a given $\mathbf{x} \in \mathbb{R}^d$ the signed distance (negative if $\mathbf{w}^\mathsf{T}\mathbf{x}+b < 0$ and positive otherwise) to the decision boundary $H = \{\mathbf{x}' \mid \mathbf{w}^\mathsf{T}\mathbf{x}' + b = 0\}$ is*

$$\frac{\mathbf{w}^\mathsf{T}\mathbf{x} + b}{\|\mathbf{w}\|}. \tag{18.4}$$

Proof. Let $\mathbf{u} = \frac{\mathbf{w}}{\|\mathbf{w}\|}$ be the unit vector in the direction of $\mathbf{w}$, and let $\mathbf{z} = r\mathbf{u}$ be the unique vector in the span of $\mathbf{w}$ that lies on the decision boundary H; see Figure 18.2. Because $\mathbf{z} \in H$, we have $\mathbf{w}^\mathsf{T}\mathbf{z} + b = 0$, so $\mathbf{z} = -\frac{b}{\|\mathbf{w}\|}\mathbf{u}$.

The distance from $\mathbf{x}$ to H is the same as the distance from $\text{proj}_\mathbf{u}(\mathbf{x})$ to $\mathbf{z}$, where $\text{proj}_\mathbf{u}(\mathbf{x}) = (\mathbf{u}^\mathsf{T}\mathbf{x})\mathbf{u}$ is the orthogonal projection of $\mathbf{x}$ onto the subspace spanned by $\mathbf{u}$. Hence the distance from $\mathbf{x}$ to H is

$$\mathbf{u}^\mathsf{T}\mathbf{x} + \frac{b}{\|\mathbf{w}\|} = \frac{\mathbf{w}^\mathsf{T}\mathbf{x} + b}{\|\mathbf{w}\|},$$

as desired. □

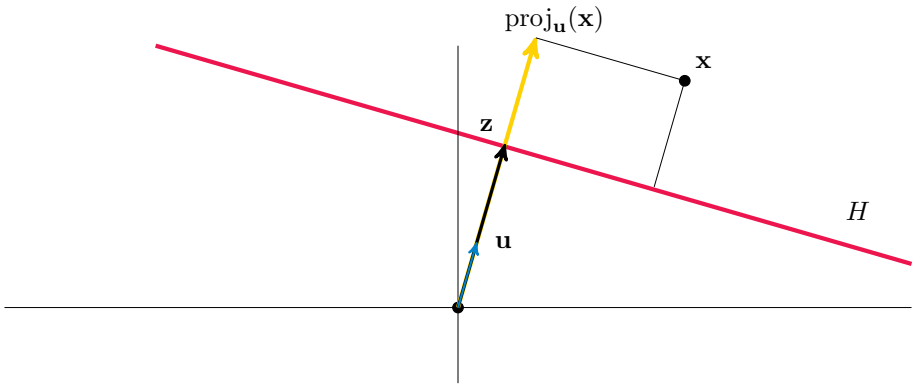


Figure 18.2: Diagram to support the proof of Proposition 18.2.1. Here $\mathbf{u} = \frac{\mathbf{w}}{\|\mathbf{w}\|}$ is the unit-length vector in the direction of $\mathbf{w}$ and $\mathbf{z} = r\mathbf{u}$ is the unique multiple of $\mathbf{u}$ lying in $H = \{\mathbf{x}' \mid \mathbf{w}^\mathsf{T}\mathbf{x}' + b = 0\}$. The distance from $\mathbf{x}$ to the hyperplane H is the same as the distance from the orthonormal projection $\text{proj}_{\mathbf{u}}\,\mathbf{x}$ to $\mathbf{z}$.

Given training data $\mathbf{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ with $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{\pm 1\}$, training the hard-margin classifier amounts to finding $b, \mathbf{w}$ defining a hyperplane $H = \{\mathbf{x}' \mid \mathbf{w}^\mathsf{T}\mathbf{x}' + b = 0\}$ that separates all the points with $y_i = +1$ from those with $y_i = -1$, and such that the distance from the support vectors (the nearest $\mathbf{x}_i$s to the decision boundary H) is maximized. Since the distance (18.4) is signed (negative if closer to the origin than H, and positive otherwise), multiplying that signed distance by the classification y_i makes it nonnegative, provided H correctly separates the two classes. That means for training the classifier we seek $b, \mathbf{w}$ to maximize the following expression:

$$\max_{b,\mathbf{w}} \frac{1}{\|\mathbf{w}\|} \min_i y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b).$$

Rescaling $b, \mathbf{w}$ by a constant $c > 0$ does not change the objective, so we may choose c such that $y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b) = 1$ for the support vectors (the points $\mathbf{x}_i$ nearest H). This allows us to reformulate the problem as that of choosing b and $\mathbf{w}$ to

$$\begin{array}{ll} \text{maximize} & \frac{1}{\|\mathbf{w}\|} \\ \text{subject to} & y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b) \geq 1 \end{array}$$

or, alternatively, to

$$\begin{array}{lll} \text{minimize} & \frac{1}{2}\|\mathbf{w}\|^2 & \\ \text{subject to} & 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b) \leq 0 & \forall i \in \{1, \ldots, N\} \end{array} \tag{18.5}$$

The objective in this problem is convex, and all the constraints are affine, making this a convex optimization problem. If d is small and the data are linearly separable, then solving the problem in this form is a straightforward numerical optimization problem.

18.2.2 Hard-Margin Dual Form

If the data are not linearly separable, then the intuition gained from Figure 18.1 suggests that a natural solution would be to embed the points $\mathbf{x}_i$ in a higher-dimensional space in which they are linearly separable. The problem is that working in a high-dimensional space is computationally expensive, so we'd like to use the kernel trick. To do this we need to express the hard-margin optimization problem solely in terms of inner products of the form $\langle \mathbf{x}_i, \mathbf{x} \rangle$ or $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$, since replacing $\mathbf{x}$ or $\mathbf{x}_i$ with $\phi(\mathbf{x})$ and $\phi(\mathbf{x}_i)$ would transform these inner products into kernel evaluations $\kappa(\mathbf{x}_i, \mathbf{x})$ or $\kappa(\mathbf{x}_i, \mathbf{x}_j)$.

To get (18.5) in the desired form, we use the dual formulation of the problem (see Volume 2 Example 15.4.13). The Lagrangian for this problem is

$$\mathscr{L}(\mathbf{w}, b, \boldsymbol{\mu}) = \frac{1}{2}\|\mathbf{w}\|_2^2 + \sum_{i=1}^{N} \mu_i (1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b)),$$

where $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_N) \succeq \mathbf{0}$.

The dual function $\widetilde{f}(\boldsymbol{\mu})$ is defined to be

$$\widetilde{f}(\boldsymbol{\mu}) = \inf_{\mathbf{w},b} \mathscr{L}(\mathbf{w}, b, \boldsymbol{\mu}),$$

and the dual formulation of the optimization problem is the constrained problem of choosing $\boldsymbol{\mu}$ to

$$\begin{aligned} &\text{maximize} && \widetilde{f}(\boldsymbol{\mu}) \\ &\text{subject to} && \boldsymbol{\mu} \succeq \mathbf{0}. \end{aligned}$$

The constraints to the primal problem are all affine, which means that it satisfies the weak Slater conditions (Volume 2, Definition 15.5.2). This implies that strong duality holds (Volume 2 Theorem 15.5.6) and the KKT first-order conditions hold (Volume 2 Theorem 15.5.9). In particular, there exists a minimizer $\mathbf{w}_*, b_*$ of the primal problem (18.5) and a maximizer $\boldsymbol{\mu}^*$ of the dual problem (18.9) such that the optimal values of the objectives are equal

$$\frac{1}{2}\|\mathbf{w}_*\|_2^2 = \sup_{\boldsymbol{\mu} \succeq \mathbf{0}} \widetilde{f}(\boldsymbol{\mu}),$$

and the KKT conditions hold, so

$$D_{\mathbf{w},b}\mathscr{L}(\mathbf{w}_*, b_*, \boldsymbol{\mu}^*) = \mathbf{0} \tag{18.6}$$

$$\mu_i^*(1 - y_i(\mathbf{w}_*^\mathsf{T}\mathbf{x}_i + b_*)) = 0 \qquad \forall i \in \{1, \ldots, N\} \tag{18.7}$$

For any given $\boldsymbol{\mu} \succeq \mathbf{0}$ the Lagrangian is a convex function of $\mathbf{w}$ and b, and so the unique minimizer $\operatorname{argmin}_{\mathbf{w},b} \mathscr{L}(\mathbf{w}, b, \boldsymbol{\mu})$ satisfies

$$\mathbf{0} = D_{\mathbf{w}}\mathscr{L}(\mathbf{w}, b, \boldsymbol{\mu}) = \mathbf{w}^\mathsf{T} - \sum_{i=1}^{N} \mu_i y_i \mathbf{x}_i^\mathsf{T}.$$

This implies that the minimizing $\mathbf{w}$ satisfies

$$\mathbf{w} = \sum_{i=1}^{N} \mu_i y_i \mathbf{x}_i, \tag{18.8}$$

and the Lagrange dual function can be rewritten as

$$\widetilde{f}(\boldsymbol{\mu}) = \inf_{\mathbf{w},b} \mathscr{L}(\mathbf{w}, b, \boldsymbol{\mu}) = \inf_{\mathbf{w},b} \left(\frac{1}{2} \left\| \sum_{i=1}^{N} \mu_i y_i \mathbf{x}_i \right\|_2^2 - \sum_{i=1}^{N} \mu_i (1 - y_i (\sum_{j=1}^{N} \mu_j y_j \mathbf{x}_j^\mathsf{T} \mathbf{x}_i + b)) \right).$$

A little algebra shows that

$$\widetilde{f}(\boldsymbol{\mu}) = \inf_{b} \left(-\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \mu_i \mu_j y_i y_j \mathbf{x}_i^\mathsf{T} \mathbf{x}_j + \sum_{i=1}^{n} \mu_i - b \sum_{i=1}^{N} \mu_i y_i \right).$$

But if $\sum_{i=1}^{N} \mu_i y_i \neq 0$, then the infimum is $-\infty$, so the feasible set for the dual problem satisies $\sum_{i=1}^{N} \mu_i y_i = 0$, and the dual problem is to choose $\boldsymbol{\mu}$ to

$$\begin{array}{ll} \text{maximize} & -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \mu_i \mu_j y_i y_j \mathbf{x}_i^\mathsf{T} \mathbf{x}_j + \sum_{i=1}^{n} \mu_i \\ \text{subject to} & \sum_{i=1}^{N} \mu_i y_i = 0 \\ & \boldsymbol{\mu} \succeq \mathbf{0}. \end{array} \tag{18.9}$$

That was a lot of work to change a fairly simple convex optimization problem (the primal problem) into something that, if anything, feels messier than the original problem. The advantage of all this work is that the only place the original data points $\mathbf{x}_i$ show up in the dual computation is as inner products $\mathbf{x}_i^\mathsf{T} \mathbf{x}_j$. If κ is a Mercer kernel, then $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \langle \varphi(\mathbf{x}_i), \varphi(\mathbf{x}_j) \rangle$ for some function φ, so replacing the inner products $\mathbf{x}_i^\mathsf{T} \mathbf{x}_j$ in (18.9) by the kernel $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ is equivalent to, but much less computationally expensive than, first mapping the data $\mathbf{x}_i$ to $\varphi(\mathbf{x}_i)$ and then solving the hard-margin classification problem for the new data $\varphi(\mathbf{x}_i)$, where the data are very likely to be linearly separable, even if the original $\mathbf{x}_i$s were not.

Substituting $\varphi(\mathbf{x}_i)$ for $\mathbf{x}_i$, we have

$$\begin{array}{ll} \text{maximize} & -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \mu_i \mu_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^{n} \mu_i \\ \text{subject to} & \sum_{i=1}^{N} \mu_i y_i = 0 \\ & \boldsymbol{\mu} \succeq \mathbf{0}. \end{array} \tag{18.10}$$

The kernelized dual problem (18.10) is a convex quadratic function in $\boldsymbol{\mu}$ with affine constraints; hence the optimization problem is a convex problem and is readily solvable with standard numerical optimization techniques. Solving the dual problem gives an optimal $\boldsymbol{\mu}^*$. The optimal $\mathbf{w}_*$ is determined by (18.8) with $\mathbf{x}_i$ replaced by $\varphi(\mathbf{x}_i)$.

$$\mathbf{w}_* = \sum_{i=1}^{N} \mu_i^* y_i \varphi(\mathbf{x}_i). \tag{18.11}$$

But, as shown below, we never need $\mathbf{w}_*$ in this form; instead we can always use use kernels.

The optimal b_* is determined by the complementary slackness equation (18.7) with $\mathbf{x}_i$ replaced by $\varphi(\mathbf{x}_i)$. Specifically, for each k with $\mu_k^* \neq 0$, we must have

$$1 - y_k(\mathbf{w}_*^\mathsf{T} \varphi(\mathbf{x}_k) + b_*) = 0$$

which gives

$$b_* = \frac{1}{y_k} - \mathbf{w}_*^\mathsf{T}\varphi(\mathbf{x}_k) = \frac{1}{y_k} - \sum_{j=1}^{N} \mu_j^* y_j \kappa(\mathbf{x}_j, \mathbf{x}_k). \tag{18.12}$$

Thus b_* can be written in terms of the kernel without using the map φ.

The learned (optimal) hard-margin classifier that results from all this is given by mapping $\mathbf{x} \in \mathbb{R}^d$ to $\varphi(\mathbf{x})$ and then computing $\text{sign}(\mathbf{w}_*^\mathsf{T}\varphi(\mathbf{x}) + b_*)$. But this can also be rewritten in terms of the kernel without φ, as follows:

$$\begin{aligned} \text{sign}(\mathbf{w}_*^\mathsf{T}\varphi(\mathbf{x}) + b_*) &= \text{sign}(\sum_{i=1}^{N} \mu_i^* y_i \varphi(\mathbf{x}_i)^\mathsf{T}\varphi(\mathbf{x}) + b_*) \\ &= \text{sign}\left(\sum_{i=1}^{N} \mu_i^* y_i \kappa(\mathbf{x}_i, \mathbf{x}) + b_*\right). \end{aligned} \tag{18.13}$$

In summary, we can train the classifier by solving the relatively easy (convex quadratic with affine constraints) kernelized dual problem (18.10) to find $\boldsymbol{\mu}^*$ without ever needing the explicit form of the function φ, and from that we can find b_* without φ using (18.12). Finally the trained classifier itself is given completely without reference to φ in (18.13).

18.2.3 Soft-margin classifier

It may be that the data $\mathbf{D}$ are not linearly separable even after mapping the $\mathbf{x}_i$ to some high-dimensional space via φ. In this case the hard-margin classifier has no solution. We can adjust it slightly as follows: Instead of requiring the inequality constraint of (18.5) to hold for all i, we use the *hinge loss* function $\max(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b))$. This loss function is zero if the inequality constraint of (18.5) holds, but is positive when it does not hold and gets larger when the point $\mathbf{x}_i$ is farther on the wrong side of the decision boundary. This lets us replace the the optimization problem (18.5) with the unconstrained problem

$$\text{minimize} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \max(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b)), \tag{18.14}$$

where C needs to be chosen. If C is small, then the objective is dominated by $\|\mathbf{w}\|^2$, and if C is large, then the loss function is dominated by the hinge loss, forcing the decision boundary to put more weight on the goal of separating the two classes.

The optimization problem (18.14) is a convex optimization problem because the loss function is a nonnegative linear combination of convex functions. If the dimension d of the data is small, then computing the optimizer is straightforward as it stands. But, if d is very large, or if we want to use the kernel trick, mapping the data into a high-dimensional space, then the computation of the optimizer could be expensive in this formulation.

Just as for the hard-margin classifier, we want to use the kernel trick, and for this we need to look at a dual problem. But since the problem is unconstrained, its dual is uninteresting. To fix this, rewrite the primal problem as

$$\begin{array}{ll} \text{minimize} & C\displaystyle\sum_{i=1}^{N}\xi_i + \frac{1}{2}\|\mathbf{w}\|_2^2 \\ \text{subject to} & 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b) \le \xi_i \quad \forall i \in \{1,\dots,N\} \\ & 0 \le \xi_i \qquad\qquad \forall i \in \{1,\dots,N\}. \end{array} \tag{18.15}$$

Here ξ_i is playing the role of $\max(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b))$.

Writing $\boldsymbol{\mu} = (\boldsymbol{\alpha}, \boldsymbol{\beta})$ in the Lagrangian gives

$$\mathscr{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = C\sum_{i=1}^{N}\xi_i + \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_{i=1}^{N}\alpha_i(\xi_i - 1 + y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b)) - \sum_{i=1}^{N}\beta_i\xi_i.$$

A little work (see Volume 2, Exercise 15.35) shows that the dual (with $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ substituted for $\varphi(\mathbf{x}_i)^\mathsf{T}\varphi(\mathbf{x}_j)$ is almost identical to that for the hard margin:

$$\begin{array}{ll} \underset{\boldsymbol{\alpha}}{\text{maximize}} & -\frac{1}{2}\displaystyle\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^{n}\alpha_i \\ \text{subject to} & \displaystyle\sum_{i=1}^{N}\alpha_i y_i = 0 \\ & 0 \le \alpha_i \le C \qquad \forall i \in \{1,\dots,N\}. \end{array} \tag{18.16}$$

Slater's condition holds, so strong duality and the KKT conditions also hold for the optimizer.

A similar argument to the hard-margin case gives b_* and the classifier in terms of the kernel $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ as

$$b_* = y_k - \sum_{i=1}^{n}\alpha_i y_i \kappa(\mathbf{x}_i, \mathbf{x}_k) \qquad \text{for all } k \text{ with } 0 < \alpha_k < C \tag{18.17}$$

$$\operatorname{sign}(\mathbf{w}_*^\mathsf{T}\varphi(\mathbf{x}) + b_*) = \operatorname{sign}\left(\sum_{i=1}^{n}\alpha_i y_i \kappa(\mathbf{x}_i, \mathbf{x})\right). \tag{18.18}$$

Computing the classification of any new point $\mathbf{x}$ now corresponds just to plugging $\mathbf{x}$ into (18.18), which can be done without ever computing $\varphi(x)$ or $\varphi(\mathbf{x}_i)$. The classifier (18.18) is called a *support vector machine (SVM)*. The choice of which kernel to use and the the value of C are usually made using a combination of cross validation and domain expertise (for example, the geometry of the problem may motivate the choice of a certain type of kernel, or knowledge about the applications may motivate the choice of how much error to allow via C).

18.2.4 Other Appications of the Kernel Trick

The kernel trick can be used in many settings other than just binary classification.

(i) There is a form of regression similar to the support vector classifier called *support vector regression*, where the loss function is similar to the soft-margin loss function, treating errors less than a certain threshold as no error at all. Of course this has the advantage of working inexpensively in higher-dimensional space, via the kernel trick, but it also is more robust than ordinary least squares because it gives no penalty for errors that are less than a certain amount.

(ii) There is a kernelized form of logistic regression that replaces the hinge loss $\max(0, y(\varphi(\mathbf{w})^\mathsf{T}, \varphi(\mathbf{x}))+b)$ of the soft-margin classifier with $\log(1+e^{y(\varphi(\mathbf{w})^\mathsf{T}, \varphi(\mathbf{x}))+b})$. The advantage of this formulation over the hard- or soft-margin of SVMs is that the resulting classifier gives an estimate of the probability that $y = 1$ instead of just a prediction of the value of y.

(iii) k-nearest neighbors is all about distances, which can be formulated in terms of inner products, hence can be kernelized. Specifically, for any input $\mathbf{x}$ one must measure the distance

$$d(\mathbf{x}, \mathbf{x}_i) = \|\mathbf{x} - \mathbf{x}_i\|_2^2 = \langle \mathbf{x}, \mathbf{x} \rangle + \langle \mathbf{x}_i, \mathbf{x}_i \rangle - 2 \langle \mathbf{x}, \mathbf{x}_i \rangle$$

for every i. Applying the kernel trick means simply replacing the inner products with the appropriate kernel κ to get a kernel distance

$$d_\kappa(\mathbf{x}, \mathbf{x}_i) = \kappa(\mathbf{x}, \mathbf{x}) + \kappa(\mathbf{x}_i, \mathbf{x}_i) - 2\kappa(\mathbf{x}, \mathbf{x}_i). \tag{18.19}$$

(iv) Clustering with *k-medoids*. Applying k-means to high-dimensional data $\varphi(\mathbf{x}_i)$ may be problematic and it is not clear how to do this with kernels. But we can replace k-means with *k-medoids*, which uses a data point (the *mediod* of the cluster) to represent each cluster's center. The medoid of a cluster is the data point $\mathbf{x}_j$, where

$$j = \operatorname*{argmin}_j \sum_{i \in \text{cluster}} d_\kappa(\mathbf{x}_j, \mathbf{x}_i),$$

and d_κ is the kernel distance defined in (18.19).

(v) *Kernel PCA* Principle component analysis with a kernel works by finding the eigenvectors of the $N \times N$ Gram matrix $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$.

Exercises

Note to the student: Each section of this chapter has several corresponding exercises, all collected here at the end of the chapter. The exercises between the first and second line are for Section 1, the exercises between the second and third lines are for Section 2, and so forth.

You should **work every exercise** (your instructor may choose to let you skip some of the advanced exercises marked with *). We have carefully selected them, and each is important for your ability to understand subsequent material. Many of the examples and results proved in the exercises are used again later in the text. Exercises marked with ⚠ are especially important and are likely to be used later in this book and beyond. Those marked with † are harder than average, but should still be done.

Although they are gathered together at the end of the chapter, we strongly recommend you do the exercises for each section as soon as you have completed the section, rather than saving them until you have finished the entire chapter.

18.1. Prove that the kernel defined in Example 18.1.2 is always Mercer but is not bilinear unless φ is a linear transformation.

18.2. Prove Proposition 18.1.9. Hint: For the second part, consider the Gram matrix K for the subset $\{\mathbf{x}, \mathbf{x}'\}$. First prove that if $\kappa(\mathbf{x}, \mathbf{x}) = 0$ and $\kappa(\mathbf{x}', \mathbf{x}') = 0$, then $\kappa(\mathbf{x}, \mathbf{x}') = 0$. Now assuming that $\kappa(\mathbf{x}', \mathbf{x}') \neq 0$ let $\mathbf{a} = (\kappa(\mathbf{x}', \mathbf{x}'), -\kappa(\mathbf{x}, \mathbf{x}'))$. Since K is positive semidefinite, we have $\mathbf{a}^\mathsf{T} K \mathbf{a} \geq 0$.

18.3. If κ_1 and κ_2 are Mercer kernels, show that the following need not be Mercer kernels:

(i) $\kappa_1 - \kappa_2$

(ii) $\kappa(\mathbf{x}, \mathbf{x}') = \exp(\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$ for any $\gamma > 0$.

18.4. Prove that the polynomial kernel (18.2) is Mercer. Hint: Consider using Proposition 18.1.10

18.5. The soft-margin SVM classifier accepts input $\mathbf{x} \in \mathbb{R}^d$ and returns a value $\hat{y}(\mathbf{x}) \in \{-1, 1\}$. Given a data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ with each $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$, the SVM returns

$$\hat{y}(\mathbf{x}) = \operatorname{sign}(\mathbf{x}^\mathsf{T} \mathbf{w}^* + b^*)$$

for $\mathbf{w}^* \in \mathbb{R}^d$ and $b^* \in \mathbb{R}$ minimizing the loss function

$$L(\mathbf{w}, b) = C \sum_{i=1}^N \max(0, 1 - y_i(\mathbf{x}_i^\mathsf{T} \mathbf{w}^* + b^*)) + \frac{1}{2} \|\mathbf{w}\|_2^2,$$

where C is a regularization parameter.

(i) Explain how the unconstrained, nonsmooth optimization problem of finding $\operatorname{argmin}_{\mathbf{w}, b} L(\mathbf{w}, b)$ can be rewritten, using slack variables ξ_i (for $i \in \{1, \ldots, n\}$), as the quadratic constrained optimization problem

$$\begin{array}{ll} \underset{\xi_i, \mathbf{w}, b}{\text{minimize}} & C \sum_{i=1}^n \xi_i + \frac{1}{2} \|\mathbf{w}\|_2^2 \\ \text{subject to} & \xi_i \geq 0 \\ & \xi_i \geq 1 - y_i(\mathbf{x}_i^\mathsf{T} \mathbf{w} + b). \end{array}$$

This is a convex optimization problem that is relatively fast to compute numerically.

(ii) Using the KKT conditions, show that there exist real values β_i and α_i for each $i \in \{1, \ldots, n\}$ such that the following hold for all i:

$$\begin{aligned} \beta_i \xi_i &= 0 \\ \alpha_i(\xi_i - (1 - y_i(\mathbf{x}_i^\mathsf{T} \mathbf{w}_* + b_*)) &= 0 \\ \mathbf{w}_* &= \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j \end{aligned}$$

(iii) Prove that for the soft-margin classifier the optimal b_* satisfies

$$b_* = y_k - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^\mathsf{T} \mathbf{x}_k$$

for all k with $0 < \alpha_k < C$.

(iv) For the soft-margin classifier with optimal $\mathbf{w}_*$, b_*, and $\boldsymbol{\alpha}^*$ prove that

$$\operatorname{sign}(\mathbf{w}_*^\mathsf{T} \mathbf{x} + b_*) = \operatorname{sign}\left(\sum_{i=1}^{n} \alpha_i^* y_i \mathbf{x}_i^\mathsf{T} \mathbf{x} \right).$$

This shows that the classifier can be kernelized, meaning that if $\kappa : \mathbb{R}^E \times \mathbb{R}^E \to \mathbb{R}$ is any Mercer kernel corresponding to a map $\varphi : \mathbb{R}^d \to \mathbb{R}^E$, then we can compute an SVM classifier

$$\hat{y}(\varphi(\mathbf{x})) = \operatorname{sign}\left(\sum_{i=1}^{n} \alpha_i^* (y_i \varphi(\mathbf{x}_i)^\mathsf{T} \varphi(\mathbf{x})) + b_* \right) = \operatorname{sign}\left(\sum_{i=1}^{n} \alpha_i^* (y_i \kappa(\mathbf{x}_i, \mathbf{x}) + b_* \right)$$

without computing (or even knowing) φ, provided we know all the α_i^*.

(v) Show that $\alpha_i^* = 0$ unless $y_i(\mathbf{x}_i^\mathsf{T} \mathbf{w}^* + b^*) \leq 1$. The $\mathbf{x}_i$ with the property that $y_i(\mathbf{x}_i^\mathsf{T} \mathbf{w}^* + b^*) \leq 1$ are called *support vectors*. If we are using the kernel trick, then the support vectors are those $\varphi(\mathbf{x}_i)$ such that $y_i(\varphi(\mathbf{x}_i)^\mathsf{T} \mathbf{w}^* + b^*) \leq 1$.

This shows that the computation of $\hat{y}(\mathbf{x})$ depends only on those few $\mathbf{x}_i$ (or $\varphi(\mathbf{x}_i)$) that are support vectors. Even without knowing φ, we can deduce which vectors are not support vectors by looking at the α_i^* (assuming we know α_i^*).

18.6. Train a support vector machine classifier on the FashionMNIST data set (if it takes too long to train, you may use 6,000 randomly selected images in the training set), using cross validation to select optimal hyperparameter values for

(i) The trivial kernel $\kappa(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle$ (corresponding to φ equal to the identity map).

(ii) A *polynomial kernel* $\kappa(\mathbf{x}, \mathbf{x}') = (\gamma \langle \mathbf{x}, \mathbf{x}' \rangle + r)^d$ for various values of γ, r, and $d \in \{2, 3, 4, 5\}$.

(iii) A *radial basis kernel* $\kappa(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$ for values of $\gamma \in \{10^-3, \ldots, 10^3\}$.

Test the best classifiers from each of the three different kernels on the FashionMNIST test set to see which performs best.

Notes

Useful notes:
Michael I Jordan on linear classifiers and SVM and Kernel trick.

(i) https://people.eecs.berkeley.edu/~jordan/courses/281B-spring04/lectures/lec1.pdf

(ii) https://people.eecs.berkeley.edu/~jordan/courses/281B-spring04/lectures/lec2.pdf

(iii) https://people.eecs.berkeley.edu/~jordan/courses/281B-spring04/lectures/lec3.pdf

(iv) https://people.eecs.berkeley.edu/~jordan/courses/281B-spring04/lectures/lec4.pdf

Also Andrew Ng on SVM: http://cs229.stanford.edu/notes/cs229-notes3.pdf

Part V

Deep Learning

19 Neural Networks

There [will] unquestionably be an explosive development in science, and it [will] be possible to let the machines tackle all the most difficult problems of science.... For what it is worth, my guess of when all this will come to pass is 1978
—I.J. Good, 1962

In from three to eight years we will have a machine with the general intelligence of an average human being.
—Marvin Minsky, 1970

By 2015...some computers will already match brains in terms of raw computing power.
—Jürgen Schmidhuber, 2006

Within a decade, AIs will be replacing scientists and other thinking professions.
—John Storrs Hall, 2011

19.1 Feedforward Neural Networks

A feedforward neural network is a parametrized function constructed as a repeated composition of some simple parametrized functions. Composing many simple functions, each depending on some parameters, gives a more interesting function depending on all the parameters. Ideally, we hope that the system can learn the correct choices of parameters to give good approximations of various functions of interest.

19.1.1 Definitions

A very natural choice for a simple function is an affine function: for a given input $\mathbf{x} \in \mathbb{R}^d$, a linear transformation $W_1 : \mathbb{R}^d \to \mathbb{R}^{d_1}$, and a constant $\mathbf{b}_1 \in \mathbb{R}^{d_1}$, the function $F_1(\mathbf{x}) = W_1\mathbf{x}+\mathbf{b}_1$ is an affine function of $\mathbf{x}$. Given two affine functions $F_1 : \mathbb{R}^d \to \mathbb{R}^{d_1}$ and $F_2 : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$, it is straightforward to check that the composition $F_2 \circ F_1$ is also affine, and thus can be written as $W\mathbf{x}+\mathbf{b}$ for some $W \in M_{d_2\times d}$ and some $\mathbf{b} \in \mathbb{R}^{d_2}$. Thus, composing multiple affine functions just gives another affine function, and there is little benefit to composing these—we might as well just use one affine function.

Most neural networks consist of compositions of affine functions with a nonlinear function $a : \mathbb{R} \to \mathbb{R}$, usually called an *activation function*. Currently a very popular choice of activation function is the *rectified linear unit (ReLU)*

$$\mathrm{ReLU}(x) = \max(x, 0) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

The original choice of activation function was just the Heaviside function

$$h(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Another activation function that was very popular until about 2013 is the sigmoid function sigm used in logistic regression (see Section 7.5.2).

Regardless of the choice of activation function, we write

$$\mathbf{a}(\mathbf{x}) = \begin{bmatrix} a(x_1) \\ \vdots \\ a(x_d) \end{bmatrix}$$

for any $\mathbf{x} \in \mathbb{R}^d$.

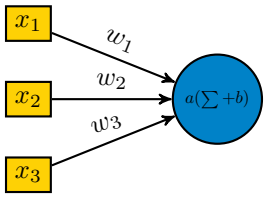
Composing affine functions with a nonlinear activation function $\mathbf{a}(\mathbf{x})$ gives a result that is not affine. Doing this repeatedly gives a feedforward neural network.

Definition 19.1.1. *A* feedforward neural network $\mathbf{n}$ *is an alternating composition of an affine function with a nonlinear activation function: activation, affine, activation, affine, and so forth*

$$f_{\mathbf{n}}(\mathbf{x}) = \mathbf{a}(W_k(\dots\mathbf{a}(W_3(\mathbf{a}(W_2(\mathbf{a}(W_1\mathbf{x}+\mathbf{b}_1)+\mathbf{b}_2))+\mathbf{b}_3))\dots)+\mathbf{b}_k).$$

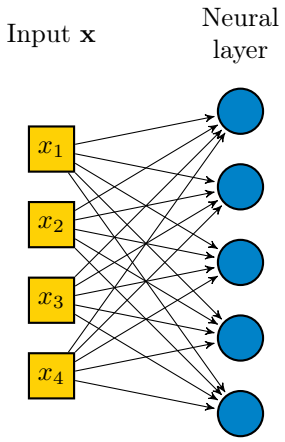
The network is often called a deep *neural network if k is larger than 3 or so, although modern deep neural networks can be thousands of layers deep. Often the final activation function differs from the others (typically a logistic function or a softmax function), and sometimes the intermediate activation functions differ from one another. The linear transformations W_i are called* weight matrices *or just* weights*. The terms $\mathbf{b}_i$ are called the* biases*. The weight matrices are often (but not always) fixed to have a special form, with many of their entries forced to be zero. If the weights W_i are not forced to have any special form, then the network is called* fully connected.

Such a composition is called a neural network because it was originally conceived as a mathematical approximation of how people thought networks of neurons in the brain work. Namely, for each neuron there are several stimuli $x_1, \dots, x_d$, and each of these is scaled by some weight w_i to get $w_i x_i$. If the sum of those inputs $\sum_i w_i x_i = \mathbf{w}^\mathsf{T}\mathbf{x}$ is below some threshold $-b$, then the neuron does nothing (output is 0), but if the weighted sum of inputs exceeds the threshold, then the neuron outputs 1, if the activation is the Heaviside function (or $\mathbf{w}^\mathsf{T}\mathbf{x} + b$ if the activation function is ReLU). Thiscould be depicted as follows:



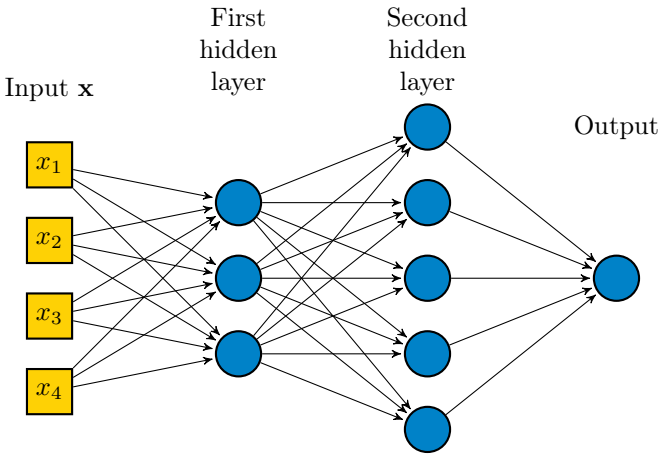
The neuron (circle) recieves weighted inputs, sums them, shifts by b, and evaluates the activation function $a(\mathbf{w}^\mathsf{T}\mathbf{x} + b)$. Assuming a is the Heaviside function, then the activation is 1 when $\mathbf{w}^\mathsf{T}\mathbf{x} \geq -b$, and it is 0 otherwise. If the activation a is the sigmoid sigm, then this single neuron is just a logistic regression, which can be trained with standard convex optimization techniques.

If many neurons are all receiving the same input, but each one applying a different affine transformation before its activation function, then the output of that group (layer) of neurons is $\mathbf{a}(W\mathbf{x} + \mathbf{b})$ for some choice of W and $\mathbf{b}$. The ith row of W is the weight vector $\mathbf{w}_i^\mathsf{T}$ for the ith neuron. If the matrix W has shape $d_{in} \times d_{out}$, then we call d_{out} the *width* of the layer. This structure is often depicted as follows:



In this example, the neural layer has width 5.

A more general neural network corresponds to multiple layers. For example, a function of the form $\mathbf{a}(W_3\mathbf{a}(W_2\mathbf{a}(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) + b_3)$ with $\mathbf{x} \in \mathbb{R}^4$, $W_1 \in M_{3\times 4}$, $W_2 \in M_{5\times 3}$, and $W_3 \in M_{1\times 5}$ is often depicted as follows:



The final layer with activation function is often called the *output layer* and the other layers are called the *hidden layers*. Often the weights and biases in a given layer are forced to have a special form (many of them zero or many of them the same as others). If the weights in a given layer are not forced to have a special form, then that layer is said to be *fully connected*.

Nota Bene 19.1.2. Unfortunately fully connected layers are often called *linear* layers because some thoughtless developer for a very popular deep learning tool happened to use that word in the code for them in 2016. *Linear* is a bad name for these because essentially all neural network layers—even those that the clumsy developer didn't call "linear"—have a linear part (a composition of an affine function) and a nonlinear part (the activation function). The name *linear layer* is wrong, because the full layer is not linear, and misleading, because forcing the weights to have a specific form doesn't make anything less linear—every layer is just as "linear" and just as "nonlinear" as any fully connected layer.

Remark 19.1.3. The function $f_{\mathbf{n}}$ defined by a neural network $\mathbf{n}$ doesn't carry all the information of $\mathbf{n}$ because it does not identify the functions in the composition that defined it. There are usually many different neural networks with different structures that all define the same function.

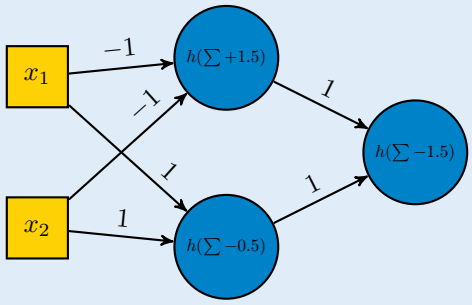
Example 19.1.4. The *exclusive or (XOR)* function is a basic logical function that can be thought of as a map from the set of pairs $\{(0,0),(0,1),(1,0),(1,1)\}$ to the set $\{0,1\}$, given as

$$\mathrm{XOR}(a,b) = \begin{cases} 0 & \text{if } a = b = 0 \text{ or } a = b = 1 \\ 1 & \text{if } a \neq b \end{cases}$$

This is often written as a *truth table*

	0	1
0	0	1
1	1	0

This function can be constructed as a neural network with a Heaviside activation as follows.



The reader should verify that each of the four different input pairs is mapped to the appropriate output by this network.

19.1.2 Training

The number of layers (the *depth*), the widths of each layer, and any constraints on the weights is often called the *topology*[51] or the *architecture* of the network. The architecture defines a family $\mathscr{F}$ of all functions that can be constructed by a neural network with the prescribed architecture.

When used for supervised learning with a given loss function $\mathscr{L}$, the neural network is *trained* by finding, or rather approximating, the following function

$$f^* = \operatorname*{argmin}_{f \in \mathscr{F}} \frac{1}{N} \sum_{i=1}^{N} \mathscr{L}(f(\mathbf{x}_i), y_i),$$

where $\mathscr{L}$ is a loss function. The functions in the set $\mathscr{F}$ are completely determined by the choices of weights W_i and biases $\mathbf{b}_i$. The training of the network amounts to finding the right choices of weights and biases. This is normally done by one of the standard numerical optimization methods (we'll say much more about this later).

[51] This topology is actually related to the idea of a topology described in Volume 1. When we speak of neural networks and graphs, they way they are connected defines a notion of which points in the network or graph are close to each other and which are not, and this is essentially what a topology is. To be more precise, defining a topology in the sense of Volume 1 means defining which subsets of the network or graph are open subsets. If you think of the graph as being locally embedded in $\mathbb{R}^2$ (you usually cannot embed the entire graph in $\mathbb{R}^2$ without edges crossing), then locally it inherits a topology from $\mathbb{R}^2$ (intersect the open sets in $\mathbb{R}^2$ with the neighborhood of the graph to define open sets in the graph). With some work one can show that the topology inherited from the local embeddings is well defined and everything works as expected. The resulting topology is completely determined by the choice of which vertices connect to which nodes. This is the topology we (and almost everyone else) mean when we talk about the topology of a graph or a network.

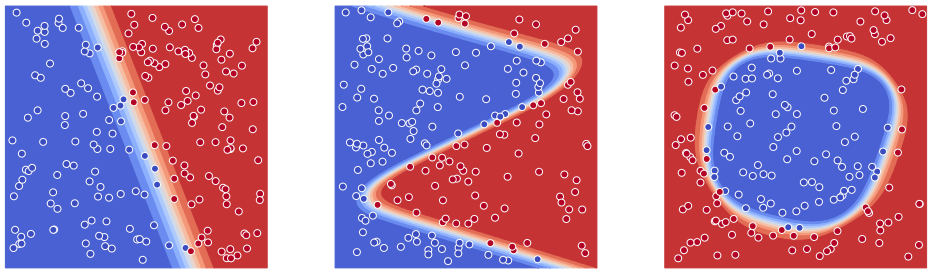


Figure 19.1: A network with just four hidden neurons is easily trained to fit these simple nonlinear patterns without any manual feature engineering. In this example all neurons used logistic activation functions.

19.1.3 Universal Approximation

A function defined as an alternating composition of activation functions and affine functions is a very special function. If the activation function is ReLU, then the resulting function is piecewise affine. Since most functions are not piecewise affine, it would be natural to think that many functions shouldn't be easy to approximate with a neural network, but it turns out that every continuous function can be approximated arbitrarily well by a neural network with ReLU activation functions and one single hidden layer (often with very large width).

Theorem 19.1.5 (Hornik [Hor93]). *Assume $a : \mathbb{R} \to \mathbb{R}$ is a continuous function that is not a polynomial. Let $X \subset \mathbb{R}^d$ be compact, and let $C(X)$ be the set of continuous functions from X to $\mathbb{R}$. If $\mathscr{F}$ is the set of functions of the form*

$$\sum_{i=1}^{n} c_i a\left(\mathbf{w}_i^{\mathsf{T}}\mathbf{x} + b_i\right),$$

where $\mathbf{w}_i \in \mathbb{R}^d$ and $b_i, c_i \in \mathbb{R}$, then the closure of $\mathscr{F}$ with respect to the uniform norm contains $C(X)$.

Corollary 19.1.6. *Let $X \subset \mathbb{R}^{d_{in}}$ be compact. Let f be a continuous function from X to $\mathbb{R}^{d_{out}}$ and let $\varepsilon > 0$. For any continuous activiation function a that is not a polynomial (including ReLU or sigmoid) there exists a fully-connected network $\mathbf{n}$ with a single hidden layer and activation function a such that*

$$\sup_{\mathbf{x}\in X} \|f(\mathbf{x}) - f_{\mathbf{n}}(\mathbf{x})\| \leq \varepsilon, \tag{19.1}$$

where $f_{\mathbf{n}}$ is the function defined by the network $\mathbf{n}$.

Proof. The function ReLU (and sigmoid) is continuous and nonpolynomial, so setting $a = \text{ReLU}$ (or sigmoid) meets the conditions of the theorem for the activation function. The function f can be broken into d_{out} component functions $f_k : X \to \mathbb{R}$ for $k \in \{1, \ldots, d_{out}\}$. The theorem shows there exists a network $\mathbf{n}_k$ for each k such that

$$\sup_{\mathbf{x} \in X} \|f_k(\mathbf{x}) - f_{\mathbf{n}_k}(\mathbf{x})\| \leq \frac{\varepsilon}{d_{out}}.$$

These can be combined into a single network $\mathbf{n}$ by concatenating the weight matrices and bias vectors of the first layers vertically, and by assembling the weight matrices of the output layer into a diagonal block matrix. Subadditivity of suprema shows that the resulting network $\mathbf{n}$ satisfies (19.1). □

This theorem and its corollary show that essentially any continuous function can be approximated arbitrarily well with a single-layer neural network, provided that single hidden layer is sufficiently wide. Unfortunately, just because there exists a choice of weights and biases that allow the neural network to approximate a given function arbitrarily well, that doesn't mean that those weights and biases can be found efficiently (or at all) by a numerical optimizer. Indeed, shallow, wide neural networks are not generally used much in practice, because such networks are hard to train.

Deep networks, on the other hand, have been the key to many breakthroughs and have helped make neural networks very popular tools in machine learning [LTR17]. Many attempts have been made to explain the success of deep networks, and this continues to be an area of active research [MPCB14, Tel15, MLP16, LS16, MP16, ABMM16, RPK$^+$17, RT17, ES16, CSS16, Tel16, PGEB18].

Several authors have proven that neural networks of fixed width and increasing depth can also be used to approximate functions. [SM15, LPW$^+$17, HLM17, Yar17, Han17, HS17]. In particular, each of these results establishes a minimum width. So long as each hidden layer has at least some minimum number of neurons, the networks can be universal approximators if they have sufficient depth (rather than by increasing the width).

Theorem 19.1.7 (Hanin et al [HS17]). *Define $w_{min}(d_{in}, d_{out})$ to be the minimal value of w such that for every continuous function $f : [0,1]^{d_{in}} \to \mathbb{R}^{d_{out}}$ and every $\varepsilon > 0$ there is a network $\mathbf{n}$ with* ReLU *activations, input dimension d_{in}, hidden layer widths at most w, and output dimension d_{out} that ε-approximates f in the sense that*

$$\sup_{\mathbf{x} \in [0,1]^{d_{in}}} \|f(\mathbf{x}) - f_{\mathbf{n}}(\mathbf{x})\| \leq \varepsilon.$$

For all d_{in}, d_{out}, the value $w_{min}(d_{in}, d_{out})$ is bounded by

$$d_{in} + 1 \leq w_{min}(d_{in}, d_{out}) \leq d_{in} + d_{out}.$$

Corollary 19.1.8. *Let $X \subset \mathbb{R}^{d_{in}}$ be compact. Let f be a continuous function from X to $\mathbb{R}^{d_{out}}$ and let $\varepsilon > 0$. There exists a fully-connected ReLU network $\mathbf{n}$ with hidden layers of width not greater than $d_{in} + d_{out}$ such that*

$$\sup_{\mathbf{x} \in X} \|f(\mathbf{x}) - f_{\mathbf{n}}(\mathbf{x})\| \leq \varepsilon.$$

Proof. Because X is compact, there exist a $c > 0$ and $\mathbf{s} \in \mathbb{R}^{d_{in}}$ such that $X' = cX + \mathbf{s} \subseteq [0,1]^{d_{in}}$. Let $\tilde{f}(\mathbf{x}) = f((\mathbf{x} - \mathbf{s})/c)$ on X'. By the theorem, there exists a satisfactory network $\mathbf{n}'$. Let $\mathbf{n}$ be a copy of $\mathbf{n}'$ in which the weight matrix W' and bias vector $\mathbf{b}'$ of the first layer have been replaced with $W = W'/c$ and $\mathbf{b} = \mathbf{b}' - W\mathbf{s}$. The new network $\mathbf{n}$ ε-approximates f, as desired. □

Figure 19.2: It's linear algebra all the way down. Source: XKCD `https://xkcd.com/1838/`

19.1.4 Eliminating Biases

So far we have formulated neural networks using two types of parameters—weights and biases. With a little rearranging, we can arrive at an equivalent formulation that uses only weights. It's a small simplification, but it makes the math significantly cleaner.

Theorem 19.1.9. *Define* $p : \mathbb{R}^d \to \mathbb{R}^{d+1}$ *by*

$$p(\mathbf{x}) = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}.$$

For every network $\mathbf{n}$ *using activation functions satisfying* $a(1) = 1$*, there is another network* $\mathbf{n}'$ *with the same type of activation functions that has no bias vectors but satisfies the relation* $f_{\mathbf{n}'}(p(\mathbf{x})) = p(f_{\mathbf{n}}(\mathbf{x}))$*.*

Proof. Define $q : \mathbb{R}^{m \times n} \times \mathbb{R}^m$ by

$$q(W, \mathbf{b}) = \begin{bmatrix} W & \mathbf{b} \\ 0 & 1 \end{bmatrix}.$$

It is straightforward to show that

$$\mathbf{a}(q(W, \mathbf{b})p(\mathbf{x})) = p(\mathbf{a}(W\mathbf{x} + \mathbf{b})). \tag{19.2}$$

This shows that once the input space has been transformed by p, all of the layers satisfying $a(1) = 1$ in a network can be replaced by layers without bias vectors. These new layers are one unit wider than the layers with bias they replace. □

By far the most common activation function in current use is ReLU, which satisfies the condition of the theorem, $\text{ReLU}(1) = 1$. Therefore, we often write all our neural networks using only weights and no biases. All the results we describe still hold for more general neural networks, but tracking the biases throughout the proofs and computations would be messier.

19.2 Training Neural Networks

Neural networks are trained with a numerical optimizer, usually a variant of gradient descent called *stochastic gradient descent (SGD)*, which we discuss in more detail in Section 19.2.2. One might think that a second-order optimizer using the Hessian of the loss function (like Newton's method) or an approximation of the Hessian, like BFGS, would be more efficient than a first-order optimizer (using only the gradient of the objective), because second-order optimizers usually converge in fewer steps than first-order optimizers. But most useful neural networks involve a large number of parameters (often millions), and second-order methods on a p-dimensional space involve matrix-vector multiplications, which have a complexity of $O(p^2)$, and sometimes also matrix inversions, which have a temporal complexity in the range of roughly $O(p^{2.37})$ to $O(p^3)$, depending on the implementation. Although the second-order methods converge in fewer steps, they take so long to complete a single step for a large neural network that it usually is much faster overall to use first-order methods.

Limited-memory BFGS (L-BFGS) is less expensive, with a complexity of $O(mp)$ where m is the number of previous updates it stores (usually a small fixed number, like 10). This means that L-BFGS can be used in training neural networks. But SGD usually outperforms L-BFGS for several reasons. First, SGD uses less memory, and it also saves a lot of time by not using all the data at each step. More important, quadratic methods are as likely to find saddle points as they are to find minima, whereas gradient descent methods rarely find saddle points (they must be almost perfectly aligned in a certain way to converge to a saddle point). The loss functions for neural networks often have saddle points, so methods that can converge to a saddle point are undesirable. Finally, SGD has many stability benefits (see Section 19.2.3). It acts as a sort of regularization, helping to prevent overfitting.

There has been some work on a stochastic variant of L-BFGS that might apply some of the time and model stability benefits of SGD to L-BFGS, but it does not change the fact that L-BFGS is more likely to find a saddle point. Moreover stochastic L-BFGS still uses more memory than SGD, which is important because training neural networks on GPUs and TPUs is usually constrained more by memory than by computational power.

In this section we discuss stochastic gradient descent, which is the main method used for training a neural network, and *backpropagation*, which is the main tool for calculating the gradient efficiently.

19.2.1 Formulation of the Optimization Problem

For a given choice of loss function $\mathscr{L}(f(\mathbf{x}), \mathbf{y})$ and a set of training data $\mathbf{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$ we want to find the minimizer

$$f^* = \operatorname*{argmin}_{f \in \mathscr{F}} \frac{1}{N} \sum_{i=1}^{N} \mathscr{L}(f(\mathbf{x}_i), y_i),$$

where $\mathscr{F}$ is a family of functions that can be defined by a neural network $\mathbf{n}$ of a given architecture:

$$f_{\mathbf{n}}(\mathbf{x}) = \mathbf{a}(W_k(\dots \mathbf{a}(W_3(\mathbf{a}(W_2(\mathbf{a}(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)) + \mathbf{b}_3)) \dots) + \mathbf{b}_k).$$

Finding f^* amounts to finding $(W_1, \dots, W_k, b_1, \dots, b_k)$ to minimize the objective function

$$J = \frac{1}{N} \sum_{i=1}^{N} \mathscr{L}(f_{\mathbf{n}}(\mathbf{x}_i), y_i)$$

Finding the parameters $(W_1, \dots, W_k, b_1, \dots, b_k)$ is called *training* the network. This is usually done with a variant of gradient descent called *stochastic gradient descent.*

19.2.2 Stochastic Gradient Descent

For optimization problems with a large number N of training instances or when computing the derivative DJ (or the gradient DJ^{T}) is expensive, we can substantially improve the performance of gradient descent by replacing the true derivative

$$DJ = \frac{1}{N} \sum_{i=1}^{N} D\mathscr{L}(f, \mathbf{x}_i, y_i) \tag{19.3}$$

with an approximation

$$DJ \approx \frac{1}{|B|} \sum_{i \in B} D\mathscr{L}(f(\mathbf{x}_i), y_i), \tag{19.4}$$

where $B \subset \{1, \ldots, N\}$ is a randomly selected subset (uniform, without replacement). This is called *stochastic gradient descent (SGD)*. If the sample B is representative of the entire training set $\mathbf{D}$, then the approximation (19.4) is close to the true gradient. If $|B| \ll N$, then this can be much cheaper to evaluate than the true gradient (19.3).

The set B is often called a *batch* or *minibatch*. One extreme variant of SGD takes the batch size to be $|B| = 1$. In this case the evaluation of (19.4) is not expensive, but the approximation is not likely to be very close to the true value of DJ. Sometimes people reserve the name *stochastic gradient descent* for this special case where $|B| = 1$, and they call the more general case with $|B| > 1$ *minibatch gradient descent*. At the other extreme, SGD is the same as usual gradient descent when the batch is the entire training set ($|B| = N$). It is common to take the batch size $|B|$ to be in the range from roughly 8 to 512, and $|B| = 128$ is a popular choice. The optimal batch size depends on many factors, including the size of the training set, the architecture of the network, and the nature of the data.

In order to ensure that the entire training set is used in the training process, it is common to partition the entire training set $\mathbf{D}$ into K disjoint minibatches $\mathbf{D} = B_1 \cup \cdots \cup B_K$, and then use $B = B_1$ for the first step of SGD, use $B = B_2$ for the next step, and so forth until all the partitions have been used. One pass through all K of the batches is called an *epoch*.[52] Typically one trains a model for several epochs—say 50 to 1000, terminating when the total loss function J seems to have converged, or when J is not improving enough at each step to justify continuing further.

Of course the partitions $B_1, \ldots, B_K$ should be chosen in a way that makes each of them a representative sample of the entire set $\mathbf{D}$. That is most easily done by randomly permuting the data (that is, changing the order of the data points) and then taking B_1 to be the first N/K instances, B_2 to be the next N/K instances, and so forth. Since many data sets come presorted in some way (oldest to newest, for example), it is always a good idea to make an initial random permutation.

19.2.3 Stability of SGD

In addition to reducing the time it takes to approximate the gradient, stochastic gradient descent has an additional benefit of helping to reduce overfitting. Specifically, at each step SGD does not go directly toward the optimizer for the full training set, but rather only in the optimal direction for the batch. But then the next step moves in the optimal direction for a different batch. This prevents the algorithm from precisely reaching the local minimizer for the entire training set, which could be over fit.

[52] In America the word *epoch* in English is usually pronounced EPP-uck, the same as the word *epic*. But in the UK it is is more commonly pronounced EE-pock. Some Americans working in deep learning have lately adopted the British pronunciation, which to our ears sounds silly when used by people who don't also say al-you-MIN-ium and SHED-jule, but you are welcome to pronounce it however you like.

A randomized algorithm is called *uniformly stable* if for all data sets differing in only one element, the learned models all give nearly the same predictions. It can be shown that SGD is uniformly stable. For convex loss functions, the stability of gradient descent decreases as a function of the sum of the step sizes. For strongly convex loss functions, stochastic gradient descent is stable even if we train for an arbitrarily long time. This may seem counterintuitive, since using standard gradient descent for an arbitrarily long time should reach the local minimizer for the loss function J (on the full training set) and since that minimizer fits the training set very well, it should not generalize well—it should be over fit. Yet SGD does not do this.

More surprisingly, these results carry over to the case where the loss function is not convex. SGD still generalizes well, provided the steps are sufficiently small and the number of iterations is not too large. More specifically, SGD generalizes well if the number of steps of SGD is bounded by N^c for some small $c > 1$ (N is the number of training data points).

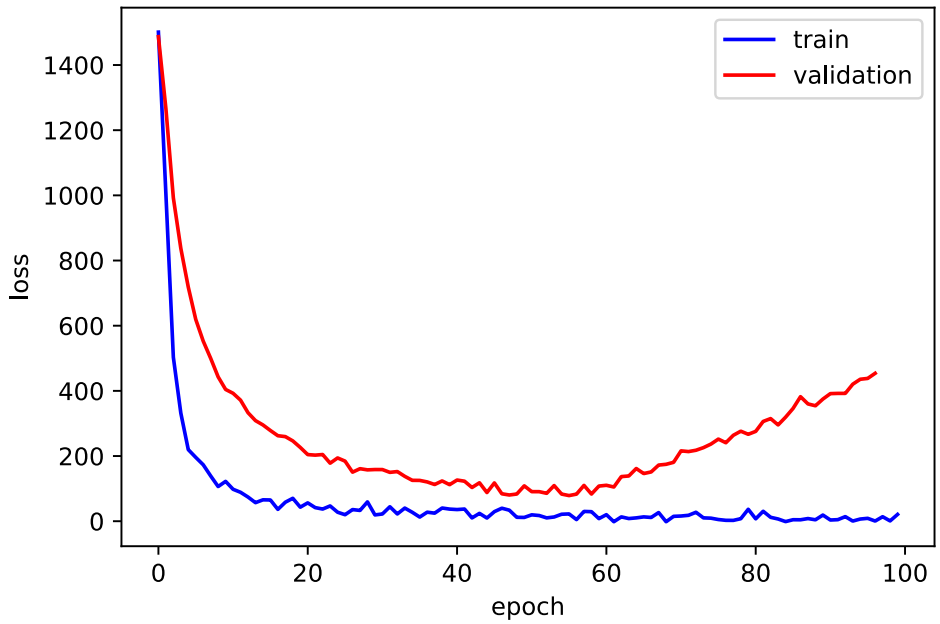


Figure 19.3: Typical neural network loss curves. The training loss decreases steadily with more training, and the validation loss decreases with it for a while, but usually the validation loss eventually starts to increase again. The usual interpratation of this is that the model becomes overfit if training continues past the minimizer of the validation loss (in this plot around epoch number 60).

19.2.4 Learning Rates

Choosing a good learning rate is difficult. If it's too large, gradient descent/SGD can become unstable and even diverge. If it's too small, not only can training take forever, but the network is more likely to overfit. To complicate matters further, the standard practice is to change the learning rate during training. Many people decrease the learning rate periodically according to a fixed schedule, resulting in loss graphs that look like that in Figure 19.4. Alternatively, one can set up a learning rate that gradually gets smaller and smaller (decays).

Decreasing the learning rate allows gradient descent to get closer to a minimizer of the loss function by taking smaller steps. However, smaller steps tend to result in improvements that don't translate as well to test data. Usually, there comes a point after which the training loss will continue to decrease while the validation loss stops or even increases a bit.

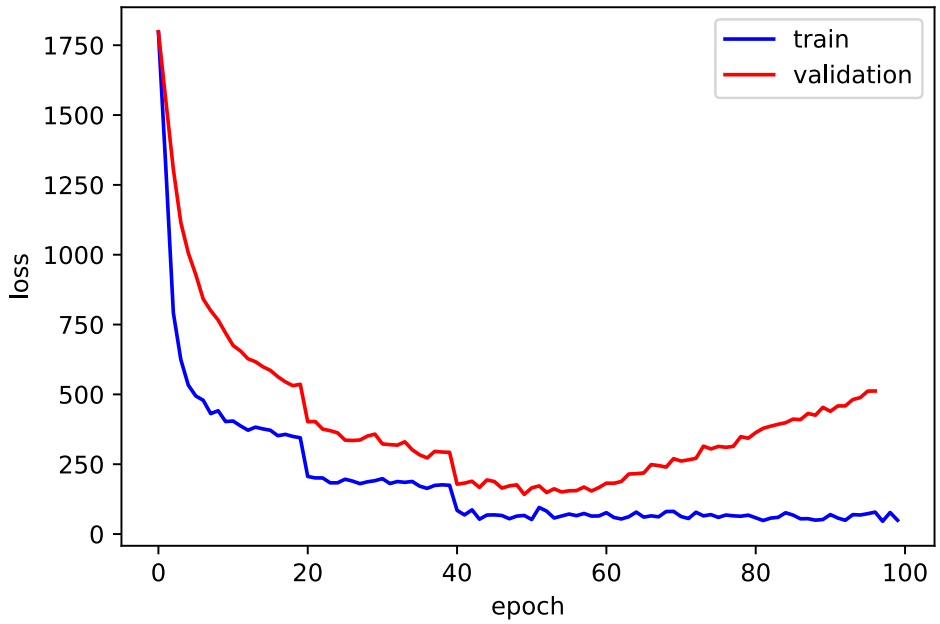


Figure 19.4: Plot of typical neural network loss curves with a decreasing learning rate schedule. Once the training loss levels off, shrinking the learning rate can often result in a significant new improvement in the training loss, and sometimes in the validation loss. Here the learning rate has been reduced at epoch 20 and again at epoch 40. By epoch 60, the validation loss has reached a minimum and training beyond that point will probably hurt the ability of the model to generalize.

19.2.5 Momentum

Gradient descent is often described with the metaphor of person descending a mountain by repeatedly taking steps downhill—in the direction of steepest descent. Momentum is explained by replacing the person with a boulder rolling downhill that also tends to continue in the same direction it was already traveling in. Momentum tends to help an optimizer escape from shallow local minima.

In pseudocode, standard gradient descent looks like the following.

```
while training:
    for p in parameters:
        dp = get_gradient(batch,p)
        p -= r * dp
```

Here we use `p` for a network parameter, `dp` for $D_p J$ (or its mini-batch approximation), and `r` for the learning rate. To add momentum, we also need a decay rate `m` and a velocity `vp`.

```
for p in parameters:
    vp = 0
while training:
    for p in parameters:
        dp = get_gradient(batch,p)
        vp = m * vp + dp
        p -= r * vp
```

The velocity term `vp` is a decaying sum of the past gradients with respect to the parameter `p`. The decay rate of this sum, $0 \leq m < 1$, determines how quickly momentum will diminish. If $m = 0$, the result is the same as regular gradient descent.

Choosing the right value of m is usually an empirical process. Momentum tends to speed up learning significantly, but too much can slow it down again by introducing oscillations. However, it's not uncommon to see decay rates as high as $m = 0.99$ in practice.

19.2.6 Adaptive Methods for Training

A number of variants of SGD called *adaptive methods* have been developed to try to train large neural networks more efficiently. Some of these, like *Adam*, are very popular because they often do, in fact, train more rapidly than SGD or SGD with momentum. Unfortunately, these methods also often have worse generalization error than SGD [WRS+17, ZFM+20]. Moreover, there is not much theory to justify why adaptive methods should be expected to find solutions with good generalization error.

This does not mean that adaptive methods like Adam and AdaGrad cannot be useful, but one should always remember that the generalization error (which we usually try to approximate with the test-set error) is what we actually want to minimize—not the training error or the validation error.

19.3 Regularization and Other Practical Considerations

The appeal of neural networks lies largely in their ability to learn complex patterns in data. However, this flexibility naturally comes with the capacity to overfit the training data. Indeed, deep neural networks often have millions of parameters and are trained on relatively small (tens of thousands) data sets, so we'd expect these models to severely overfit the training data—badly enough to make these models unusable. One of the great mysteries of modern deep learning is that deep learning models usually don't overfit nearly as badly as one might expect.

Although overfitting usually isn't as bad a problem as it could be, it is still a problem. In this section we discuss a number of methods have been developed to help combat overfitting, as well as a few other practical considerations for developing and training neural networks.

19.3.1 Regularization

Regularization methods are simple and popular ways to try to reduce overfitting. *Parameter regularization* adds a penalty to the loss function to keep the weights of a network from growing too large.

$$J = \frac{1}{N}\sum_{i=1}^{N} \mathscr{L}\left(f(\mathbf{x}_i), y_i\right) + c\frac{1}{2}\sum_{\ell} \|W_\ell\|_F^2$$

Here the second sum runs over all the weights in the network. Biases are typically not penalized because they do not usually contribute to overfitting. This specific example uses the Frobenius norm of the weight matrices W_ℓ, which amounts to the sum of the squares of the individual entries w_{ij}^2 (the L^2-norm of the flattened matrix). Assume that the learning rate is α. Applying one step of SGD with this regularization corresponds to updating the weights as

$$\begin{aligned} W^{\text{new}} &= W - \alpha D_W J^{\mathsf{T}}(W, b) \\ &= W - \alpha\frac{1}{|B|}\sum_{B} D\mathscr{L}^{\mathsf{T}}\left(f(\mathbf{x}_i), y_i\right) - \alpha c W \\ &= (1 - \alpha c)W - \alpha\frac{1}{|B|}\sum_{B} D\mathscr{L}^{\mathsf{T}}\left(f(\mathbf{x}_i), y_i\right) \end{aligned}$$

This shows that at each step of regularized SGD, the weights W shrink by a factor of $(1 - \alpha c)$. For this reason, this type of regularization is also called *weight decay*.

Another common choice is L^1 regularization, which uses the absolute values $|w_{ij}|$ instead of w_{ij}^2. As always, L^1-regularization tends to encourage sparsity (that is, it rewards setting more of the w_{ij} to zero).

With all types of regularization, it is difficult to choose an appropriate regularization constant c. If c is too small, it won't impact training. If it is too large, the network may force weights to be too small (or too many to be zero) because it prioritizes regularization over the original learning task. Appropriate regularization constants are generally found empirically.

19.3.2 Dropout

Dropout is another anti-overfitting tool. With dropout, some neurons are randomly dropped from the network at each training iteration. Specifically, dropout requires the user to choose a probability p, and each hidden neuron is ignored with probability p, in the sense that their outputs are replaced with zero and the associated parameters are not updated. Thus each training iteration looks at a random subnetwork, asking it to perform the network's task despite the missing units.

This method compels the network to learn redundancy. When neurons can be dropped, the network must ensure that critical information is learned by sets of neurons. At test time, the whole network is used. Dropout in training means that the complete network behaves somewhat like an ensemble of smaller models. This tends to result in better generalization.

Here are two examples of submodels that dropout might produce at different training iterations. Output neurons are not dropped because in general it doesn't make sense for the output $\hat{y}$ to have fewer dimensions than y.

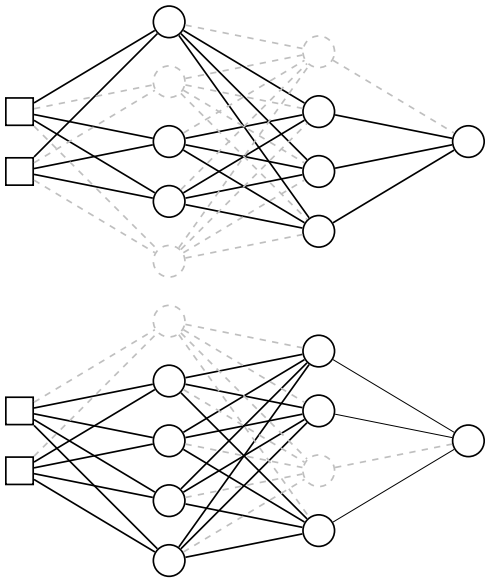


Figure 19.5: Example subnetworks active during a given training step with dropout.

19.3.3 Initialization

Most networks are initialized with random values. It is unwise to initialize with weights equal to zero because that is a special point for which the gradient also often vanishes, and so gradient-based methods will not easily move away from this point.

Most commonly, biases are initially set to zero while weights are randomly sampled from distributions such as normal, uniform, or truncated normal. All of these distributions are consistently chosen to have mean zero, while the variance of W_i is usually chosen to be ℓ/d_i where ℓ is a constant (popular values for ℓ include 1, 2, and 6). The scaling is done to keep the derivatives of the loss function from having wildly differing magnitudes, as this can cause problems for gradient descent.

19.3.4 Preprocessing

Like other machine learning models, neural networks perform appreciably better when data is processed so as to have certain properties. In particular, it is often helpful to normalize inputs—shifting and scaling the inputs $\mathbf{x}$ so as to have zero mean and unit variance. Like the previously described initialization scheme, this makes SGD more effective by ensuring that different parameters have derivatives of similar magnitudes. Categorical data is usually reshaped so as to consist of one-hot vectors. Additional, specialized preprocessing techniques exist for structured data types, such as images and text.

As discussed in Section 16.2.3, different situations require different choices about whether to normalize each feature separately (subtract the mean of each feature and divide by the standard deviation of that feature), or to normalize them all together (subtract the mean across all features and divide by the stadard deviation across all features), or to avoid normalizing altogether.

19.3.5 Batch Normalization

As mentioned above, normalizing the inputs tends to give better results in many cases. In deeper networks this idea also helps when applied to the inputs to intermediate layers. When training with SGD, however, we only use a few training points (a batch) at each step, and so we only use the corresponding intermediate-layer inputs for computing the normalization. This is called *batch normalization*.

The original authors of the idea of batch normalization claimed that it "reduces internal covariate shift." We won't unpack that phrase here because subsequent work has shown that batch normalization does *not* reduce internal covariate shift [STIM18]. Nevertheless, batch normalization does seem to have significant positive impact on training for deeper networks. Specifically, it seems to allow for larger learning rates, which leads to faster training.

Batch normalization also seems to help avoid overfitting and improve generalization error.

Nota Bene 19.3.1. Generally is it not recommended to use both batch norm and dropout in the same layers. The extra randomness that arises from dropping some neurons tends to throw off the normalization. Moreover, batch normalization generally has a better overall effect than dropout.

19.3.6 Hyperparameter Validation

In addition to the parameters that are optimized by gradient descent—the weights and biases—a neural network has many *hyperparameters* that are found experimentally. These include depth (number of layers), layer width (number of neurons in a given layer), regularization constants, and dropout probability. There are also many additional choices that are more about optimization than about architecture, but still affect the finaly performance of the model. These include learning rates, momentum decay rates, batch sizes, when to stop training, and possibly other choices.

For this reason, it is prudent to have (at least) three data sets—train, validate, and test. The training of a specific model (using SGD or a variant) should only be done using the training set. The person or machine choosing the hyperparameters can see how the hyperparameters affect performance on the validation set and adjust them accordingly. But this means that the validation data has been used for optimization (training) and so can't be used for final testing.

The test set should only be used at the very end to evaluate how well the model is likely to generalize—not to further adjust hyperparameters or select models. The three-way split is necessary to avoid bias and give meaningful conclusions about generalizability.

19.3.7 Brief History of Neural Networks

The original neural network (due to Rosenblatt in 1958) consisted of a single neuron with the Heaviside function as the activation. This was later upgraded to a single-hidden-layer neural network, again with heaviside activations. Rosenblatt's big contribution was an algorithm to train the network on supervised binary classification problems. This is significant because the function defined by the network was not continuous. This simple model, together with Rosenblatt's algorithm, was called the *perceptron*. The perceptron got a lot of attention, and it was widely and heavily hyped.

Closely related to the perceptron was the *adaptive linear element (ADALINE)*, which in modern terms was just a single neuron with the identity activation function (thus, completely affine). ADALINE was trained with a version of SGD, but without needing back propagation, because the derivative of an affine function is an easily computed linear function.

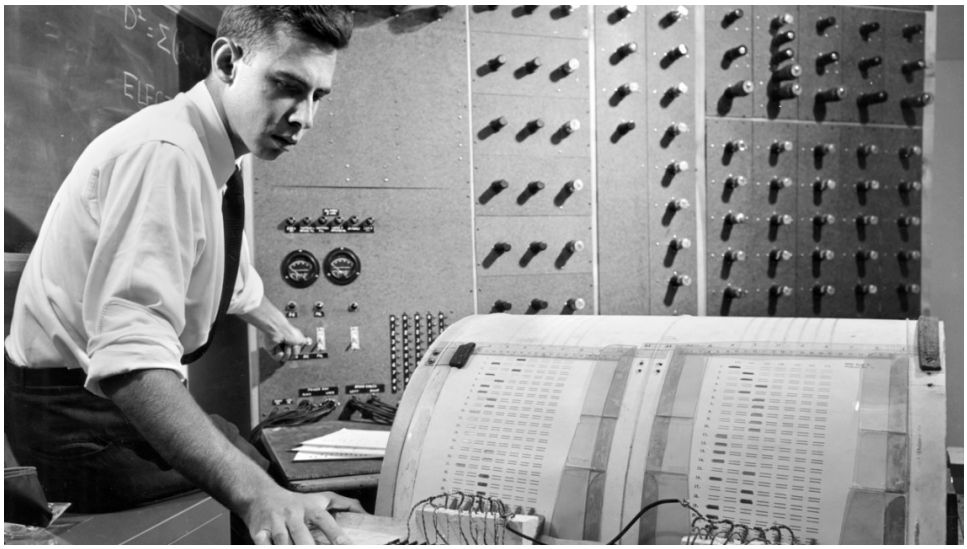


Figure 19.6: Frank Rosenblatt with his perceptron.

Although it could be trained, the perceptron had the drawback of only being able to draw a linear separation—it was not, by any means, a universal function approximator. This was not recognized by most until it was pointed out by Marvin Minsky and Seymor Pappert in 1969 that the perceptron (and the linear models) could not model the XOR function. The perceptron training algorithm also did not really work with multi-layered networks, so this severely limited its ability to approximate functions. Minsky and Pappert's work played a big role in starting the first *AI winter*, when funding for artificial intelligence dried up because it became clear that artificial intelligence was not living up to all the hype.

In the 1980s there was another AI boom. *Expert systems*, which in many cases are essentially just decision trees, became very popular. Around this time people started using stochastic gradient descent to train multilayer neural networks with the sigmoid activation function.[53] The use of the sigmoid activation was needed to do gradient descent in the multilayer networks, since the perceptron algorithm does not work for multilayer networks and because the derivative of the Heaviside function is always either zero or undefined. The use of stochastic gradient descent rather than just gradient descent may have been initially just a convenience to try to reduce computing time and save memory, but it had the wonderful effect of making the resulting networks much more generalizable (more likely to do well on data that was not in the original training set). In 1986 Geoff Hinton popularized the (previously known, but unappreciated by AI researchers) *back propagation* algorithm for computing gradients more efficiently than had been done before. Back propagation is just the reverse mode of automatic differentiation, which had already been described in 1970 by Seppo Linnainmaa.

[53]Some people call any fully connected (feedforward) neural network a *multilayer perceptron*, regardless of the activation function used. Others use the term only to describe neural networks that use the Heaviside activation function. Using the term *perceptron* is a great way to sound more grown up—everyone will think you were born in 1930.

But despite these improvements, computing power was not sufficient to train networks of any real depth. Moreover, the sigmoid function tended to suffer from a *vanishing gradient* problem—for inputs very far from 0, the derivative of the sigmoid function is very close to zero, which means that if the weights in a neural network are not very close to correct, the gradient for those weights will be close to zero and thus will not move much, making it hard to train the network. Once again, neural networks and other forms of AI failed to live up to the hype of their promoters, and funding dried up in the mid 1990s—marking the beginning of the second AI winter.

Because of all these issues, neural networks were not the most popular nor the most useful models until about 2011. Many considered them to be out of date and not very practical. According to Geoff Hinton, the International Conference on Machine Learning (ICML) accepted no papers with the words *neural network* in the title in 2009. Other models like support vector machines and random forests were seen by most as superior to neural networks in essentially every way. But deep (meaning more than two layers!) neural networks started to really take off about 2011 because of two things:

- Increasing computing power. This came in part just by Moore's law, but it was also affected by the realization (by Andrew Ng and his collaborators in 2009) that graphics processing units (GPUs), which had been specially developed to do linear algebra quickly for video rendering, were also very well suited for doing the linear algebra needed to train deep neural networks.

- Switching from a sigmoid to ReLU as the activation function. This first happened in neural networks in 2011. This just seems like a lucky break. ReLU works much better as an activation function than we have any right to expect. Yes, it is clear that ReLU won't suffer as much from the vanishing gradient problem as sigm does, and ReLU is also fast to compute and easy to compute the derivative of; but lots of functions could be used to solve the vanishing gradient problem. And it's not at all obvious that ReLU won't have serious problems with vanishing gradients for negative inputs. In fact, ReLU does still have some problem with vanishing gradients, but it works surprisingly well in many settings.

19.4 Backpropagation and Automatic Differentiation

Neural networks are usually trained using SGD or some variant of SGD. The main ingredient these algorithms need is the derivative with respect to all the parameters $\Theta = W_1, \ldots, W_k, \mathbf{b}_1, \ldots, \mathbf{b}_k$ of the loss function $\mathscr{L}(f_{\mathbf{n}}(\mathbf{x}_i), y_i)$ at a data point $(\mathbf{x}_i, y_i)$.

$$D_\Theta \mathscr{L}(f_{\mathbf{n}}(\Theta, \mathbf{x}_i), y_i) = D_f \mathscr{L}(f, y_i) D_\Theta f_{\mathbf{n}}(\Theta, \mathbf{x}_i)$$

The first term $D_f \mathscr{L}(f, y_i)$ is usually simple to compute, for example, if $\mathscr{L}(f(\mathbf{x}), y) = \frac{1}{2}(f(\mathbf{x}) - y)^2$ then

$$D_f \mathscr{L}(f, \mathbf{y}) = (f - y).$$

So a significant part of training a neural network boils down to computing the derivative of $f_{\mathbf{n}}$ with respect to all the parameters Θ:

$$D_\Theta f_{\mathbf{n}}.$$

Symbolic computation of the derivative of a complicated function like

$$f_{\mathbf{n}}(\mathbf{x}, \Theta) = \mathbf{a}_k(W_k \mathbf{a}_{k-1}(W_{k-1}(\cdots \mathbf{a}_1(W_1 \mathbf{x}) \cdots))) \tag{19.5}$$

for nonlinear functions $\mathbf{a}_1, \ldots, \mathbf{a}_k$ is generally not feasible, especially if k is very large. Symbolic differentiation often suffers from expansion swell, where the number of terms in the symbolic expression grows very rapidly; see Volume 2, Section 11.4.1.

Computing derivatives numerically by taking a difference quotient like the *forward difference*

$$f'(w) \approx \frac{f(w+h) - f(w)}{h}$$

or the *centered difference*

$$f'(w) \approx \frac{f(w+h) - f(w-h)}{2h}$$

for small values of $h > 0$ is also problematic, mostly because a good choice of of h is tricky to determine, and a poor choice gives bad results; for more on this see Volume 2, Example 11.1.9 and Section 11.4.2.

Instead of symbolic differentiation or difference quotient approximation, neural networks are trained using *back propagation*, which we prefer to call the *reverse mode* of *automatic differentiation*, or *algorithmic differentiation*. The idea and initial implementation of reverse mode automatic differentiation is due independently to Seppo Linnainmaa in his masters thesis in 1970 and Gerardi Ostrowski in 1971.

19.4.1 The Chain Rule

The main idea of automatic differentiation is to replace each function f with a pair of functions, (f, Df), where Df represents the total derivative of $f(\mathbf{x})$ as a function of the variable $\mathbf{x}$. That is, for some large library of basic function whose derivatives are known and compuatable (we'll call these *primitive* functions), an evaluation of a primitive function $f(\mathbf{x})$ at some point $\boldsymbol{\alpha}$ should compute not only $f(\boldsymbol{\alpha})$ but also $Df(\boldsymbol{\alpha})$.

Example 19.4.1. If $g(\mathbf{x}) = g(x_1, x_2) = x_1 x_2$ is treated as a primitive function in an automatic differentiation scheme, then a call to evaluate g at the point $\boldsymbol{\alpha} = (1.7, 2.3)$ should return both the function evalution $g(\boldsymbol{\alpha})$ but also the total derivative $\begin{bmatrix} \frac{\partial g}{\partial x_1} & \frac{\partial g}{\partial x_2} \end{bmatrix}$ evaluated at the point $\boldsymbol{\alpha}$. That is, it returns the pair

$$(g(\boldsymbol{\alpha}), D_{\mathbf{x}} g(\boldsymbol{\alpha})) = (\alpha_1 \alpha_2, \begin{bmatrix} \alpha_2 & \alpha_1 \end{bmatrix}) = (3.91, \begin{bmatrix} 2.3 & 1.7 \end{bmatrix}).$$

Example 19.4.2. If matrix multiplication is treated as a primitive function in an automatic differentiation scheme, and if $f : \mathbb{R}^2 \to \mathbb{R}^3$ is matrix multiplication by

$$B = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix},$$

then in the framework of automatic differentiation, a call to evaluate f at the point $\boldsymbol{\alpha} = (1.7, 2.3)$ should return the pair

$$(f(\boldsymbol{\alpha}), D_{\mathbf{x}} f(\boldsymbol{\alpha})) = (B\boldsymbol{\alpha}, B) = \left(\begin{bmatrix} 2.3 \\ 10.3 \\ 18.3 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} \right).$$

If a function F is constructed as the composition of several primitive functions, the derivative of F can be computed using the chain rule. Assume that

$$F(\mathbf{x}) = (f_k \circ f_{k-1} \circ \cdots \circ f_1)(\mathbf{x}). \tag{19.6}$$

To compute the derivative $D_{\mathbf{x}} F(\boldsymbol{\alpha}_0)$, let $\boldsymbol{\alpha}_i = f_i(\boldsymbol{\alpha}_{i-1})$ for each $i \in \{1, \ldots, k\}$ and use the chain rule

$$D_{\mathbf{x}} F(\boldsymbol{\alpha}_0) = Df_k(\boldsymbol{\alpha}_{k-1}) Df_{k-1}(\boldsymbol{\alpha}_{k-2}) \cdots Df_2(\boldsymbol{\alpha}_1) Df_1(\boldsymbol{\alpha}_0). \tag{19.7}$$

In an automatic differentiation scheme, each intermediate derivative evaluation $Df_i(\boldsymbol{\alpha}_{i-1})$ is evaluated when $f(\boldsymbol{\alpha}_{i-1})$ is evaluated. The derivative of the composition is computed by multiplying these individual derivatives together as in (19.10). This multiplication can be implemented in many ways, but there are two standard choices: *forward mode*, which multiplies them from right to left (in the same order that the functions are evaluated), and *reverse mode*, which multiplies them in from left to right (in reverse of the order that the functions are evaluated). Each of these modes has advantages and disadvantages.

19.4.2 Forward Mode Automatic Differentiation

The forward mode of automatic differentiation is equivalent to multiplying the derivatives in (19.10) from right to left—in the same order that the functions are evaluated:

$$D_{\mathbf{x}} F(\boldsymbol{\alpha}_0) = Df_k(\boldsymbol{\alpha}_k) \Big(\cdots \big(Df_2(\boldsymbol{\alpha}_2) Df_1(\boldsymbol{\alpha}_1) \big) \Big). \tag{19.8}$$

This is done by keeping a running product, denoted $\dot{\boldsymbol{\alpha}}_i$:

$$\dot{\boldsymbol{\alpha}}_i = \Big(Df_i(\boldsymbol{\alpha}_{i-1}) \Big(\cdots \big(Df_2(\boldsymbol{\alpha}_1) Df_1(\boldsymbol{\alpha}_0) \big) \Big) \Big) = Df_i(\boldsymbol{\alpha}_{i-1}) \dot{\boldsymbol{\alpha}}_{i-1}. \tag{19.9}$$

The evaluation of $F(\boldsymbol{\alpha}_0)$ and $DF(\boldsymbol{\alpha}_0)$ is done by consecutive evaluations of the primitive functions and their derivatives. At step i the partial composition $\boldsymbol{\alpha}_i = f_i(\boldsymbol{\alpha}_{i-1})$ is computed, as well as the derivative $Df_i(\boldsymbol{\alpha}_{i-1})$, and then the partial product $\dot{\boldsymbol{\alpha}}_i = Df_i(\boldsymbol{\alpha}_{i-1})\dot{\boldsymbol{\alpha}}_{i-1}$. At this point the system need no longer retain the derivative $Df_i(\boldsymbol{\alpha}_{i-1})$—all that is needed for the next step is the pair $(\boldsymbol{\alpha}_i, \dot{\boldsymbol{\alpha}}_i)$.

In summary, a forward mode automatic differentiation scheme for compositions like (19.6) can be implemented by replacing each function f with a function-derivative pair $(f, \delta f)$ consisting of a function f and the corresponding partial product $\delta f = Df(\boldsymbol{\alpha})\dot{\boldsymbol{\alpha}}$. Each pair $(f, \delta f)$ accepts as input a pair of values $(\boldsymbol{\alpha}, \dot{\boldsymbol{\alpha}})$ and returns the pair $(f(\boldsymbol{\alpha}), Df(\boldsymbol{\alpha})\dot{\boldsymbol{\alpha}})$. The derivative $D_{\mathbf{x}}F(\boldsymbol{\alpha}_0)$ of the composition (19.6) is computed simply by evaluating these pairs consecutively and returning the final $\dot{\boldsymbol{\alpha}}_k$:

$$(F(\boldsymbol{\alpha}_0), DF(\boldsymbol{\alpha}_0) = (\boldsymbol{\alpha}_k, \dot{\boldsymbol{\alpha}}_k).$$

Example 19.4.3. To compute the derivative of $\sin(\log(x))$ as a function of x, the forward mode of automatic differentiation uses the function-derivative pair $(\log(a), \delta\log)$ which accepts as input $(a, \dot{a})$ and returns $\big(\log(a), \frac{\dot{a}}{a}\big)$. It also uses the pair $(\sin, \delta\sin)$ which accepts as input $(a, \dot{a})$ and returns $(\sin(a), \cos(a)\dot{a})$.

For example, to compute the derivative of $\frac{d}{dx}\sin(\log(x))$ at $\alpha_0 = 2$, the initial input to the composite function is the pair $(\alpha, \dot{\alpha}) = (2, 1)$. The term $\dot{\alpha} = 1$ corresponds to the empty product, or it could be thought of as the total derivative of the identity function $\dot{\alpha} = \frac{d}{dx}x|_{x=2} = 1$. The steps of the computation then are

$$\begin{aligned}
(a_0, \dot{a}_0) &= (x, \dot{x}) = (2, 1) \\
(a_1, \dot{a}_1) &= (\log, \delta\log)(a_0, \dot{a}_0) = (\log(2), 0.5) \\
(a_2, \dot{a}_2) &= (\sin, \delta\sin)(a_1, \dot{a}_1) = (\sin(\log(2)), \cos(\log(2))0.5)
\end{aligned}$$

By the chain rule (19.11), the desired derivative is the final term $\dot{a}_2 = 0.5\cos(\log(2))$.

Example 19.4.4. For a fixed $B \in M_{2\times 2}(\mathbb{R})$, consider $F(\mathbf{x}) = \mathbf{x}^\mathsf{T}B^\mathsf{T}B\mathbf{x}$, which we can write as $F(\mathbf{x}) = (f \circ g)(\mathbf{x})$, where $f(\mathbf{y}) = \mathbf{y}^\mathsf{T}\mathbf{y}$ and $g(\mathbf{x}) = B\mathbf{x}$. To compute the derivative of F as a function of $\mathbf{x}$, the forward mode of automatic differentiation uses the pair $(g, \delta g)$, which accepts as input $(\boldsymbol{\alpha}, \dot{\boldsymbol{\alpha}})$ and returns

$$(g(\boldsymbol{\alpha}), Dg(\boldsymbol{\alpha})\dot{\boldsymbol{\alpha}}) = (B\boldsymbol{\alpha}, B\dot{\boldsymbol{\alpha}}).$$

It also uses the pair $(f, \delta f)$, which accepts $(\boldsymbol{\alpha}, \dot{\boldsymbol{\alpha}})$ as input and returns

$$(f(\boldsymbol{\alpha}), Df(\boldsymbol{\alpha})\dot{\boldsymbol{\alpha}}) = (\boldsymbol{\alpha}^\mathsf{T}\boldsymbol{\alpha}, 2\boldsymbol{\alpha}^\mathsf{T}\dot{\boldsymbol{\alpha}}).$$

The initial input to the composite function is $(\mathbf{x}, \dot{\mathbf{x}}) = (\mathbf{x}, I)$. The term $\dot{\boldsymbol{\alpha}}_0 = I$ occurs as the empty matrix product, or, alternatively, it could be thought of as the derivative (Jacobian) $D_\mathbf{x}\mathbf{x} = I$ of the identity function $\mathbf{x}$ with respect to $\mathbf{x}$.

For example, if $B = \left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$, and $\mathbf{x} = (-7, 5)$, to compute the derivative $DF(\mathbf{x})$, the algorithm proceeds as

$$\begin{aligned}
(\boldsymbol{\alpha}_0, \dot{\boldsymbol{\alpha}}_0) &= \left(\begin{bmatrix} -7 \\ 5 \end{bmatrix}, I\right) \\
(\boldsymbol{\alpha}_1, \dot{\boldsymbol{\alpha}}_1) &= (g, \delta g)(\boldsymbol{\alpha}_0, \dot{\boldsymbol{\alpha}}_0) = (B\boldsymbol{\alpha}_0, B\dot{\boldsymbol{\alpha}}_0) = \left(\begin{bmatrix} 3 \\ -1 \end{bmatrix}, B\right) \\
(\boldsymbol{\alpha}_2, \dot{\boldsymbol{\alpha}}_2) &= (f, \delta f)(\boldsymbol{\alpha}_1, \dot{\boldsymbol{\alpha}}_1) = (\boldsymbol{\alpha}_1^\mathsf{T}\boldsymbol{\alpha}_1, 2\boldsymbol{\alpha}_1^\mathsf{T}B) = (10, \begin{bmatrix} 0 & 4 \end{bmatrix})
\end{aligned}$$

Thus $F(\mathbf{x}) = 10$ and $DF(\mathbf{x}) = \begin{bmatrix} 0 & 4 \end{bmatrix}$.

If, instead of the total derivative $DF(\mathbf{x})$, we want to compute a directional derivative $D_\mathbf{v}F$, one naïve way to do this is to compute the total derivative $DF(\mathbf{x})$ and then compute the matrix-vector product $DF(\mathbf{x})\mathbf{v}$. But this can be computed more efficiently by replacing the initial empty product $D_\mathbf{x}\mathbf{x} = I$ with the directional derivative $D_\mathbf{v}\mathbf{x} = \mathbf{v}$ and running through the forward mode, which immediately gives the desired directional derivative $D_\mathbf{v}F(\mathbf{x})$. It is generally much cheaper, computationally, to make a forward pass with an initial $\delta\boldsymbol{\alpha}_0 = \mathbf{v}$ than it is to compute the total derivative $DF(\mathbf{x})$ followed by the matrix-vector product $DF(\mathbf{x})\mathbf{v}$.

Example 19.4.5. In the previous example (Example 19.4.4), to compute the directional derivative $D_\mathbf{v}F(\mathbf{x})$ in the direction $\mathbf{v} = (\sqrt{2}, 1)$ at $\mathbf{x} = (-7, 5)$, simply set $\dot{\boldsymbol{\alpha}}_0 = \mathbf{v}$ and proceed with the usual forward mode:

$$\begin{aligned}
(\boldsymbol{\alpha}_0, \dot{\boldsymbol{\alpha}}_0) &= (\mathbf{x}, \dot{\mathbf{x}}) = \left(\begin{bmatrix} -7 \\ 5 \end{bmatrix}, \mathbf{v}\right) = \left(\begin{bmatrix} -7 \\ 5 \end{bmatrix}, \begin{bmatrix} \sqrt{2} \\ 1 \end{bmatrix}\right) \\
(\boldsymbol{\alpha}_1, \dot{\boldsymbol{\alpha}}_1) &= (g, \delta g)(\boldsymbol{\alpha}_0, \dot{\boldsymbol{\alpha}}_0) = (B\boldsymbol{\alpha}_0, B\delta\boldsymbol{\alpha}_0) = \left(\begin{bmatrix} 3 \\ -1 \end{bmatrix}, \begin{bmatrix} 3.4142 \\ 8.2426 \end{bmatrix}\right) \\
(\boldsymbol{\alpha}_2, \dot{\boldsymbol{\alpha}}_2) &= (f, \delta f)(\boldsymbol{\alpha}_1, \dot{\boldsymbol{\alpha}}_1) = \left(\boldsymbol{\alpha}_1^\mathsf{T}\boldsymbol{\alpha}_1, 2\boldsymbol{\alpha}_1^\mathsf{T}\delta\boldsymbol{\alpha}_1\right) = (10, 4)
\end{aligned}$$

Thus, the directional derivative in the direction $\mathbf{v}$ is 4.

19.4.3 Forward Mode with Computation Graphs

Writing a general multivariable function as a composition of several multivariable primitive functions in the style of (19.6) often results in large sparse derivatives and inefficient computations.

Example 19.4.6. Let $F(x_1, x_2, x_3) = x_1x_2 + \log(x_3)$ We can write this as a composition like (19.6) by setting $F = f_3 \circ f_2 \circ f_1$ with

$$\begin{aligned} f_1(x_1, x_2, x_3) &= (x_1x_2,\ x_3) \\ f_2(a_1, a_2) &= (a_1, \log(a_2)) \\ f_3(b_1, b_2) &= b_1 + b_2 \end{aligned}$$

Evaluating $f_3 \circ f_2 \circ f_1(x_1, x_2, x_3)$ and the derivatives in the forward mode gives

$$\begin{aligned} Df_1 &= \begin{bmatrix} x_2 & x_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ Df_2 &= \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{x_3} \end{bmatrix} \\ Df_3 &= \begin{bmatrix} 1 & 1 \end{bmatrix}, \end{aligned}$$

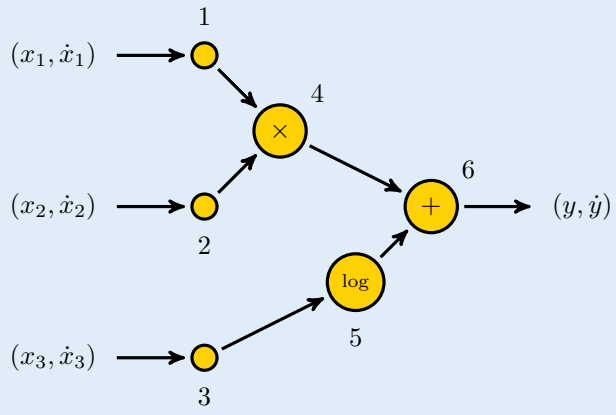
Multiplying these matrices involves many computations that contribute zero to the final result.

$$\begin{aligned} Df_3Df_2Df_1 &= \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{x_3} \end{bmatrix} \begin{bmatrix} x_2 & x_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_2 + 0 & x_1 + 0 & 0 + 0 \\ 0 + 0 & 0 + 0 & 0 + \frac{1}{x_3} \end{bmatrix} = \begin{bmatrix} x_2 & x_1 & \frac{1}{x_3} \end{bmatrix}. \end{aligned}$$

Writing compositions of functions like that in Example 19.4.6 leads to derivatives with many zeros, which makes the matrix multiplications in (19.9) inefficient. It would be better not to have to include those in the computations. Moreover, functions like those in these two examples are not really natural choices for primitives. It would be better to express these in terms of fewer, more fundamental functions.

We can solve both of these problems by reformulating the function F in terms of a *computation graph*, which is a DAG whose nodes are primitive function-derivative pairs and whose directed edges indicate that the output of the source node should be input into the target node. Nodes of the DAG that have no incoming edges correspond to the inputs to the function F and nodes that have no outgoing edges correspond to outputs of F. The matrix multiplications of the forward mode can be rewritten as a sum over incoming edges.

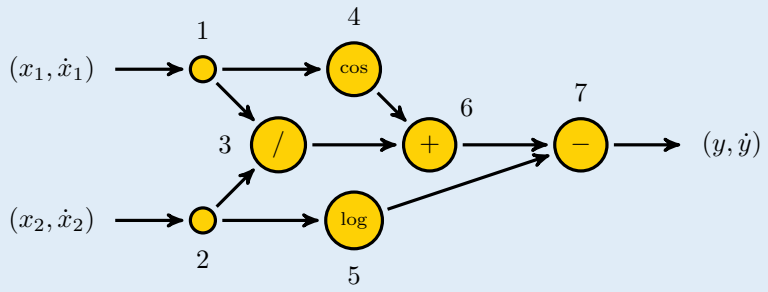
Example 19.4.7. Let $F(x_1, x_2, x_3) = x_1x_2 + \log(x_3)$ be the function from Example 19.4.6. A computation graph for this function (assuming the primitive pairs include log and standard arithmetic operations) is



The matrix multiplications of the forward mode can be rewritten as a sum over incoming edges. For each node ν of the computation graph with corresponding function g_ν, write the output of the corresponding function evaluation as α_ν and the result of the corresponding (forward-mode partial matrix product) derivative as $\dot{\alpha}_\nu$. Also, let I_ν denote the set of nodes i with a directed edge from i into ν. For this particular computation graph, we have

$$
\begin{aligned}
(\alpha_1, \dot{\alpha}_1) &= (x_1, \dot{x}_1) \\
(\alpha_2, \dot{\alpha}_2) &= (x_2, \dot{x}_2) \\
(\alpha_3, \dot{\alpha}_3) &= (x_3, \dot{x}_3) \\
\alpha_4 &= \alpha_1 \alpha_2 \\
\dot{\alpha}_4 &= Dg_4(\alpha_1, \alpha_2) \begin{bmatrix} \dot{\alpha}_1 \\ \dot{\alpha}_2 \end{bmatrix} = \sum_{i \in I_4} \frac{\partial g_4}{\partial \alpha_i} \dot{\alpha}_i \\
&= \alpha_2 \dot{\alpha}_1 + \alpha_1 \dot{\alpha}_2 \\
\alpha_5 &= \log(\alpha_3) \\
\dot{\alpha}_5 &= Dg_5(\alpha_3)\dot{\alpha}_3 = \sum_{i \in I_5} \frac{\partial g_5}{\partial \alpha_i} \dot{\alpha}_i = \frac{1}{\alpha_3} \dot{\alpha}_3 \\
\alpha_6 &= \alpha_4 + \alpha_5 \\
\dot{\alpha}_6 &= Dg_6(\alpha_4, \alpha_5) \begin{bmatrix} \dot{\alpha}_4 \\ \dot{\alpha}_5 \end{bmatrix} = \sum_{i \in I_6} \frac{\partial g_6}{\partial \alpha_i} \dot{\alpha}_i = \dot{\alpha_4} + \dot{\alpha_5}
\end{aligned}
$$

Example 19.4.8. Let $F(x_1, x_2) = \cos(x_1) + \frac{x_1}{x_2} - \log(x_2)$. A computation graph for this function (assuming the primitive pairs include cos, log, and standard arithmetic operations) is



The forward-mode computation is

$$
\begin{aligned}
(a_1, \dot{a}_1) &= (x_1, \dot{x}_1) \\
(a_2, \dot{a}_2) &= (x_2, \dot{x}_2) \\
(a_3, \dot{a}_3) &= \left(g_3(a_1, a_2), \sum_{i \in I_3} \frac{\partial g_3}{\partial a_i} \dot{a}_i \right) = \left(\frac{a_1}{a_2}, \frac{\dot{a}_1 a_2 - a_1 \dot{a}_2}{a_2^2} \right) \\
(a_4, \dot{a}_4) &= \left(g_4(a_1)), \sum_{i \in I_4} \frac{\partial g_4}{\partial a_i} \dot{a}_i \right) = (\cos(a_1), -\sin(a_1) \dot{a}_1) \\
(a_5, \dot{a}_5) &= \left(g_5(a_2), \sum_{i \in I_5} \frac{\partial g_5}{\partial a_i} \dot{a}_i \right) = (\log(a_2), \frac{\dot{a}_2}{a_2}) \\
(a_6, \dot{a}_6) &= \left(g_6(a_3, a_4), \sum_{i \in I_6} \frac{\partial g_6}{\partial a_i} \dot{a}_i \right) = (a_3 + a_4, \dot{a}_3 + \dot{a}_4) \\
(a_7, \dot{a}_7) &= \left(g_7(a_5, a_6), \sum_{i \in I_7} \frac{\partial g_7}{\partial a_i} \dot{a}_i) \right) = (-a_5 + a_6, -\dot{a}_5 + \dot{a}_6) \\
(y, \dot{y}) &= (a_7, \dot{a}_7)
\end{aligned}
$$

For example, at the point $(x_1, x_2) = (3, 4)$ the forward mode of automatic differentiation computes $\frac{\partial F}{\partial x_1}$ as follows:

$$
\begin{aligned}
(a_1, \dot{a}_1) &= (a_1, \dot{a}_1) = (x_1, \dot{x}_1) = (3, 1) \qquad \text{because } \frac{\partial}{\partial x_1} x_1 = 1 \\
(a_2, \dot{a}_2) &= (a_2, \dot{a}_2) = (x_2, \dot{x}_2) = (4, 0) \qquad \text{because } \frac{\partial}{\partial x_1} x_2 = 0 \\
(a_3, \dot{a}_3) &= (a_3, \dot{a}_3) = (g_3, \delta g_3)((3,1)(4,0)) = \left(\frac{3}{4}, \frac{1 \cdot 4 - 3 \cdot 0}{4^2}\right) = (0.75, 0.25) \\
(a_4, \dot{a}_4) &= (g_4, \delta g_4)(3,1) = (\cos(3), -\sin(3)) \\
(a_5, \dot{a}_5) &= (g_5, \delta g_5)(4,0) = (\log(4), \frac{0}{4}) = (\log(4), 0) \\
(a_6, \dot{a}_6) &= (g_6, \delta g_6)((0.75, 0.25), (\sin(3), \cos(3))) = (0.75 + \cos(3), 0.25 - \sin(3)) \\
(a_7, \dot{a}_7) &= (g_7, \delta g_7)((\log(4), 0), (0.75 + \cos(3), 0.25 - \sin(3))) \\
&= (-\log(4) + 0.75 + \cos(3), 0.25 - \sin(3)) \\
(y, \dot{y}) &= (a_7, \dot{a}_7)
\end{aligned}
$$

This algorithm gives both the function evaluation $y = F(3,4) = -\log(4) + 0.75 + \cos(3)$ and the derivative evaluation $\dot{y} = \frac{\partial}{\partial x_1} F|_{(3,4)} = 0.25 - \sin(3)$.

19.4.4 Complexity Depends on Order of Computation

Assume that $F(\mathbf{x}) = (f_k \circ f_{k-1} \circ \cdots \circ f_1)(\mathbf{x})$, and $\mathbf{a}_{i+1} = f_i(\mathbf{a}_i)$, with $\mathbf{a}_0 = \mathbf{x}$. The chain rule gives

$$DF(\mathbf{x}) = Df_k(\mathbf{a}_{k-1}) Df_{k-1}(\mathbf{a}_{k-2}) \cdots Df_2(\mathbf{a}_1) Df_1(\mathbf{a}_0). \tag{19.10}$$

Assume also that $f_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}$, and thus the derivative $Df_i(\mathbf{a}_{i-1})$ is represented by an $n_i \times n_{i-1}$ matrix.

The complexity of carrying out this matrix multiplication depends on the order in which the multiplications are carried out. Recall that the complexity of multiplying an $m \times n$ matrix A times an $n \times r$ matrix B is $O(mnr)$ and produces a matrix of size $m \times r$. If C is $r \times q$, then computing $(AB)C$ has complexity $O(mnr + mrq)$, while computing $A(BC)$ has complexity $O(mnq + nrq)$. If m is small and the other dimensions, n, r, q are large, then $mnr + mrq \ll mnq + nrq$, so it is advantageous to compute $(AB)C$ instead of $A(BC)$. If, on the other hand, m, n, r are large and q is small, then it is preferable to evaluate the matrix multiplication as $A(BC)$.

Computing the derivative using the forward mode amounts to doing the matrix multiplication in (19.10) as

$$DF(\mathbf{x}) = \Big(Df_k(\mathbf{a}_{k-1}) \Big(Df_{k-1}(\mathbf{a}_{k-2}) \Big(\cdots \big(Df_3(\mathbf{a}_2) (Df_2(\mathbf{a}_1) Df_1(\mathbf{a}_0)) \big) \Big) \Big) \Big). \tag{19.11}$$

Usually in a machine learning the function that needs to be evaluated is a loss function, which takes values in $\mathbb{R}^1$, but the inputs and intermediate stages may be very high dimensional. That means the final dimension in the loss function is $n_k = 1$ and the others are $n_i \gg 1$ for all $i < k$. Thus the complexity of computing the matrix multiplications in (19.11) is

$$n_0 n_1 n_2 + n_0 n_2 n_3 + n_1 n_3 n_4 + \cdots + n_0 n_{k-1} \cdot 1.$$

It would be much cheaper to multiply in the opposite order, from last to first, which has complexity

$$n_{k-2} n_{k-1} \cdot 1 + n_{k-3} n_{k-2} \cdot 1 + \cdots + n_1 n_2 \cdot 1 + n_0 n_1 \cdot 1.$$

For example, consider a composition $F(\mathbf{x}) = f_3 \circ f_2 \circ f_1(\mathbf{x})$, where $\mathbf{x}$ has dimension 10^6 (as might occur with a photograph) and the outputs of f_1 and f_2 both have dimension $1,000$. The complexity of the matrix multiplications in the product (19.11) is on the order of $10^6 \cdot 10^3 \cdot 10^3 + 10^6 \cdot 10^3 \cdot 1 \approx 10^{12}$, while multiplying from last to first has complexity on the order of $1 \cdot 10^3 \cdot 10^3 + 1 \cdot 10^3 \cdot 10^6 \approx 10^9$. Just changing the order of multiplication reduces the complexity by a factor of $1,000$. This motivates doing automatic differentiation in a way that allows these matrix multiplications to be computed from last to first, that is, in *reverse mode*.

19.4.5 Reverse-Mode Automatic Differentiation

Computing derivatives in reverse mode still uses the chain rule and still computes the intermediate steps of the composite function one at a time in order from first to last, but it waits until the last (outer) function has been evaluated before computing the matrix products. To accomplish this, instead of computing the derivatives $D_{\mathbf{x}} f_i$, it computes the *adjoint*, as we now explain.

Assume, as in the previous section, that $F(\mathbf{x}) = (f_k \circ f_{k-1} \circ f_{k-2} \circ \cdots \circ f_0)(\mathbf{x})$, and $\mathbf{a}_i = f_i(\mathbf{a}_{i-1})$, with $\mathbf{a}_0 = \mathbf{x}$. Moreover, denote

$$F_i(\mathbf{a}_i) = f_k \circ f_{k-1} \circ \cdots \circ f_{i+1}(\mathbf{a}_i)$$

For each i the *adjoint* is the value $\bar{\mathbf{a}}_i = D_{\mathbf{a}_i} F_i(\mathbf{a}_i)$ is the derivative of F_i as a function of $\mathbf{a}_i$. Reverse-mode automatic differentiation is done first by a forward pass that computes the values $\mathbf{a}_i = f_i(\mathbf{a}_{i-1})$ and $D_{\mathbf{a}_{i-1}} f_i(\mathbf{a}_{i-1})$, stores those values, and then, once $\mathbf{y} = \mathbf{a}_k$ has been computed, it makes a backward pass, that starts at $\bar{y} = \bar{\mathbf{a}}_k = 1$ and works backwards, computing each adjoint $\bar{\mathbf{a}}_i$ by

$$\bar{\mathbf{a}}_i = D_{\mathbf{a}_i} F_i(\mathbf{a}_i) = \bar{\mathbf{a}}_{i+1} D_{\mathbf{a}_i} f_{i+1}(\mathbf{a}_i) \tag{19.12}$$

The computation in (19.12) requires only a single matrix multiplication of the previous step $\bar{\mathbf{a}}_{i+1}$ and the stored value of $D_{\mathbf{a}_i} f_{i+1}(\mathbf{a}_i)$. Once that matrix product has been computed, both of these intermediate matrices can be forgotten.

Example 19.4.9. To compute the derivative $\frac{d}{dx}\sin(\log(x))$ at $x = \alpha_0$, the reverse mode of automatic differentiation computes the pair $(\alpha_1, D_{\alpha_0}\log(\alpha_0)) = \left(\log(\alpha_0), \frac{1}{\alpha_0}\right)$. It then computes the pair $(\alpha_2, D_{\alpha_1}\sin(\alpha_1)) = (\sin(\alpha_1), \cos(\alpha_1))$. This completes the forward pass. The backward pass begins with $\bar{\alpha}_2 = 1$, then computes the product $\bar{\alpha}_1 = \bar{\alpha}_2\cos(\alpha_1)$, and finally $\bar{\alpha}_0 = \bar{\alpha}_1\frac{1}{\alpha_0}$. This gives $\frac{d}{dx}\sin(\log(x))|_{x=\alpha_0} = \bar{\alpha}_0$.

In the case that $\alpha_0 = 2$, the steps of the reverse-mode differentiation are

$$\begin{aligned}
\alpha_0 &= 2 \\
(\alpha_1, D_{\alpha_0}\log(\alpha_0)) &= (\log(2), 0.5) \\
(\alpha_2, D_{\alpha_1}\sin(\alpha_1)) &= (\sin(\log(2)), \cos(\log(2))) \\
\bar{\alpha}_2 &= 1 \\
\bar{\alpha}_1 &= 1 \cdot \cos(\log(2)) \\
\bar{\alpha}_0 &= \cos(\log(2)) \cdot (0.5)
\end{aligned}$$

This agrees with the result of the forward computation, Example 19.4.3. For functions of one variable, there is usually no special benefit to using the reverse mode over the forward mode.

Example 19.4.10. Recall the function $F(\mathbf{x}) = \mathbf{x}^\mathsf{T}B^\mathsf{T}B\mathbf{x} = (f \circ g)(\mathbf{x})$, where $f(\mathbf{y}) = \mathbf{y}^\mathsf{T}\mathbf{y}$ and $g(\mathbf{x}) = B\mathbf{x}$ of Example 19.4.4 ($B \in M_{2\times 2}(\mathbb{R})$ is fixed). To compute the derivative of F as a function of $\mathbf{x}$ at $\boldsymbol{\alpha}_0$, the reverse mode of automatic differentiation computes the pair

$$(\boldsymbol{\alpha}_1, D_{\boldsymbol{\alpha}_0}g(\boldsymbol{\alpha}_0)) = (g(\boldsymbol{\alpha}_0), D_{\boldsymbol{\alpha}_0}g(\boldsymbol{\alpha}_0)) = (B\boldsymbol{\alpha}_0, B).$$

It then computes the pair

$$(\boldsymbol{\alpha}_2, Df(\boldsymbol{\alpha}_1)) = (f(\boldsymbol{\alpha}_1), D_{\boldsymbol{\alpha}_1}f(\boldsymbol{\alpha}_1)) = (\boldsymbol{\alpha}_1^\mathsf{T}\boldsymbol{\alpha}_1, 2\boldsymbol{\alpha}_1^\mathsf{T}).$$

This completes the forward pass. The backward pass begins with $\bar{\boldsymbol{\alpha}}_2 = 1$ and proceeds with

$$\begin{aligned}
\bar{\boldsymbol{\alpha}}_1 &= \bar{\boldsymbol{\alpha}}_2 2\boldsymbol{\alpha}_1^\mathsf{T} = 2\boldsymbol{\alpha}_1^\mathsf{T} \\
\bar{\boldsymbol{\alpha}}_0 &= \bar{\boldsymbol{\alpha}}_1 B = 2\boldsymbol{\alpha}_1^\mathsf{T}B
\end{aligned}$$

For example, if $B = \left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$, and $\boldsymbol{\alpha}_0 = (-7, 5)$, then to compute the derivative $DF(\boldsymbol{\alpha}_0)$, the algorithm proceeds as

$$\begin{aligned}
(\boldsymbol{\alpha}_1, D_{\boldsymbol{\alpha}_0} g(\boldsymbol{\alpha}_0)) &= (B\boldsymbol{\alpha}_0, B) = \left(\begin{bmatrix} 3 \\ -1 \end{bmatrix}, \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix} \right) \\
(\boldsymbol{\alpha}_2, Df(\boldsymbol{\alpha}_1)) &= (\boldsymbol{\alpha}_1^{\mathsf{T}} \boldsymbol{\alpha}_1, 2\boldsymbol{\alpha}_1^{\mathsf{T}}) = (10, \begin{bmatrix} 6 & -2 \end{bmatrix}) \\
\bar{\boldsymbol{\alpha}}_2 &= 1 \\
\bar{\boldsymbol{\alpha}}_1 &= 12\boldsymbol{\alpha}_1^{\mathsf{T}} = \begin{bmatrix} 6 & -2 \end{bmatrix} \\
\bar{\boldsymbol{\alpha}}_0 &= \begin{bmatrix} 6 & -2 \end{bmatrix} B = \begin{bmatrix} 0 & 4 \end{bmatrix}
\end{aligned}$$

This agrees with the result of the forward mode computation in Example 19.4.4

19.4.6 Reverse Mode on Computation Graphs

The efficiency of the reverse mode can usually be improved by using the same computation graph as the forward mode, but storing different information along the way. Multiplying the matrices in backward mode amounts to summing over outgoing edges instead of incoming edges. Specifically, for each node ν of the computation graph with corresponding function g_ν, write the output of the corresponding function evaluation as α_ν. To each incoming edge from node i to ν compute and store also the derivative $\frac{\partial g_\nu}{\partial \alpha_i}$. Computing the α_ν and all the partial derivatives $\frac{\partial g_\nu}{\partial \alpha_i}$ is the forward pass. The backward pass consists of starting with $\bar{\alpha_y} = 1$, where y is the final (output) node, and then working backward, for each node ν let $\mathscr{O}_\nu$ denote the set of outgoing nodes, that is, nodes o with an edge from ν to o. Now compute

$$\bar{\alpha}_\nu = \sum_{o \in \mathscr{O}_\nu} \bar{\alpha}_o \frac{\partial g_o}{\partial \alpha_\nu}. \tag{19.13}$$

The sum over the outgoing nodes is equivalent to the corresponding matrix multiplication in the backward pass. For any input node ν (that is, one of the nodes with no incoming edges), the partial derivative $\frac{\partial F}{\partial \nu}$ is the computed quantity $\bar{\alpha}_n u$.

Example 19.4.11. Let $F(x_1, x_2, x_3) = x_1 x_2 + \log(x_3)$ be the function from Examples 19.4.6 and 19.4.7. The forward pass of the reverse mode uses the same computation graph as in Example 19.4.7, but instead of computing each incoming $\dot{\alpha}_i$, compute and store the derivative Dg_ν with respect to its inputs; that is, compute the partial derivatives $\frac{\partial g_\nu}{\partial \alpha_i}$ for each incoming edge from i to ν. Given inputs α_1, α_2, and α_3, the forward pass of the reverse mode proceeds as

$$\begin{aligned}
(\alpha_4, Dg_4) &= \left(\alpha_1\alpha_2, \begin{bmatrix} \frac{\partial g_4}{\partial \alpha_1} & \frac{\partial g_4}{\partial \alpha_2} \end{bmatrix} \right) = (\alpha_1\alpha_2, \begin{bmatrix} \alpha_2 & \alpha_1 \end{bmatrix}) \\
(\alpha_5, Dg_5) &= (\log(\alpha_3), \begin{bmatrix} \frac{\partial g_5}{\partial \alpha_3} \end{bmatrix}) = (\log(\alpha_3), \begin{bmatrix} \frac{1}{\alpha_3} \end{bmatrix}) \\
(\alpha_6, Dg_6) &= (\alpha_4 + \alpha_5, \begin{bmatrix} \frac{\partial g_6}{\partial \alpha_4} & \frac{\partial g_6}{\partial \alpha_5} \end{bmatrix}) = \left(\alpha_4 + \alpha_5, \begin{bmatrix} 1 & 1 \end{bmatrix}\right).
\end{aligned}$$

This gives the function evaluation $F(\alpha_1, \alpha_2, \alpha_3) = \alpha_6$. The backward pass begins with $\bar{\alpha}_6 = 1$ and proceeds using (19.13).

$$\begin{aligned}
\bar{\alpha}_5 &= 1 \cdot 1 = 1 \\
\bar{\alpha}_4 &= 1 \cdot 1 = 1 \\
\bar{\alpha}_3 &= \bar{\alpha}_5 \frac{\partial g_5}{\partial \alpha_3} = \frac{1}{\alpha_3} \\
\bar{\alpha}_2 &= \bar{\alpha}_4 \frac{\partial g_4}{\partial \alpha_2} = \alpha_1 \\
\bar{\alpha}_1 &= \bar{\alpha}_4 \frac{\partial g_4}{\partial \alpha_1} = \alpha_2.
\end{aligned}$$

The total derivative $DF(\alpha_1, \alpha_2, \alpha_3)$ is given by $\begin{bmatrix} \bar{\alpha}_1 & \bar{\alpha}_2 & \bar{\alpha}_3 \end{bmatrix} = \begin{bmatrix} \alpha_2 & \alpha_1 & \frac{1}{\alpha_3} \end{bmatrix}$.

Example 19.4.12. Let $F(x_1, x_2) = \cos(x_1) + \frac{x_1}{x_2} - \log(x_2)$ be as in Example 19.4.8 with the same computation graph, but, as before, storing different data along the way. At the point $(x_1, x_2) = (3, 4)$ the reverse mode of automatic differentiation first makes the forward pass

$$\begin{aligned}
(\alpha_1, Dg_1) &= (3, 1) \\
(\alpha_2, Dg_2) &= (4, 1) \\
(\alpha_3, Dg_3(\alpha_1, \alpha_2)) &= \left(0.75, \begin{bmatrix} -0.25 & -0.1875 \end{bmatrix}\right) \\
(\alpha_4, Dg_4) &= \left(\cos(3)), \begin{bmatrix} -\sin(3) \end{bmatrix}\right) \\
(\alpha_5, Dg_5) &= \left(\log(4), \begin{bmatrix} 0.25 \end{bmatrix}\right) \\
(\alpha_6, Dg_6) &= \left(0.75 + \cos(3), \begin{bmatrix} 1 & 1 \end{bmatrix}\right) \\
(\alpha_7, Dg_7) &= \left(0.75 + \cos(3) - \log(4)), \begin{bmatrix} -1 & 1 \end{bmatrix}\right) \\
y &= \alpha_7
\end{aligned}$$

This gives the function evaluation $y = F(3,4) = -\log(4) + 0.75 + \cos(3)$. The backward pass of the reverse mode computation begins with $\bar{y} = 1$ and proceeds using (19.13):

$$\begin{aligned}
\bar{\alpha_7} &= \bar{y} = 1 \\
\bar{\alpha_6} &= \bar{y} \cdot \frac{\partial g_7}{\partial \alpha_6} = 1 \cdot 1 = 1 \\
\bar{\alpha_5} &= \bar{\alpha}_7 \cdot \frac{\partial g_7}{\partial \alpha_5} = 1 \cdot -1 = -1 \\
\bar{\alpha_4} &= \bar{\alpha}_6 \cdot \frac{\partial g_6}{\partial \alpha_4} = 1 \cdot 1 = 1 \\
\bar{\alpha_3} &= \bar{\alpha}_6 \cdot \frac{\partial g_6}{\partial \alpha_3} = 1 \cdot 1 = 1 \\
\bar{\alpha_2} &= \bar{\alpha}_3 \cdot \frac{\partial g_3}{\partial \alpha_2} + \bar{\alpha}_5 \cdot \frac{\partial g_5}{\partial \alpha_2} = -0.1875 - 0.25 = -0.4375 \\
\bar{\alpha_1} &= \bar{\alpha}_3 \cdot \frac{\partial g_3}{\partial \alpha_1} + \bar{\alpha}_4 \cdot \frac{\partial g_4}{\partial \alpha_1} = 0.25 - \sin(3).
\end{aligned}$$

The backward pass gives the total derivative evaluation

$$DF(\alpha_1, \alpha_2) = \begin{bmatrix} 0.25 - \sin(3) & -0.4375 \end{bmatrix}.$$

Note that the forward mode computation of Example 19.4.8 gave only the partial derivative $\frac{\partial F}{\partial \alpha_1} = 0.25 - \sin(3)$, while the reverse mode computation gave the total derivative $DF(\alpha_1, \alpha_2) = \begin{bmatrix} \frac{\partial F}{\partial \alpha_1} & \frac{\partial F}{\partial \alpha_2} \end{bmatrix} = \begin{bmatrix} 0.25 - \sin(3) & -0.4375 \end{bmatrix}$. This illustrates the general principle that the forward mode returns only one of the partial derivatives for each pass, so for a function of n variables, the forward mode must be run n times to compute the total derivative DF, but the reverse mode gives the total derivative DF in just one forward and one backward pass.

19.4.7 Sometimes Automatic Differentiation Fails

Although automatic differentiation is a very powerful tool, there are times when it completely fails. As a simple example, consider the functions $f(x) = x^2 - 9$, $g(x) = x - 3$ and $h(x) = f(x)/g(x)$. Of course $h(3)$ is not defined, and most computers will return NaN for $h(3)$, but for all $x \neq 3$ the function h is equal to the simpler expression $h(x) = x + 3$ and so has derivative $h'(x) = 1$. Unfortunately for values of x sufficiently close to 3, the denominator underflows and automatic differentiation returns NaN for all values in that neighborhood. Moreover, in a larger neighborhood the numerator of h underflows before the denominator does, making region where the result of automatic differentiation is zero when it should, instead be 1. The "bad" region in this example is not terribly large (about 10^{-7} in length), but in other examples it can be larger.

Despite this problem automatic differentiation is a very useful tool. This is partly because these failure modes are not especially common in most applications (for example, neural networks built from compositions of ReLU activations and affine transformations). But if you find yourself in a setting where tools that use automatic differentiation[54] are not performing as expected, it may be useful to consider whether the functions being differentiated might be simplified or rewritten in a way that removes some of these problems. For example, rewriting the function h in the preceding example as $h(x) = x + 3$ removes all the problems with that example.

[54]This includes most optimizers, including stochastic gradient descent, Adam, and Adagrad.

19.5 Convolutional Neural Networks

19.5.1 Convolving with Filters

Fully connected neural networks do poorly on common types of real-world data because they don't account for relationships between features. For example, it's easy to represent an image as a vector by allocating one entry for each pixel value, but the resulting representation fails to capture the relationships between pixels that result from their arrangement in the image plane. Convolving with a filter is one way to address this.

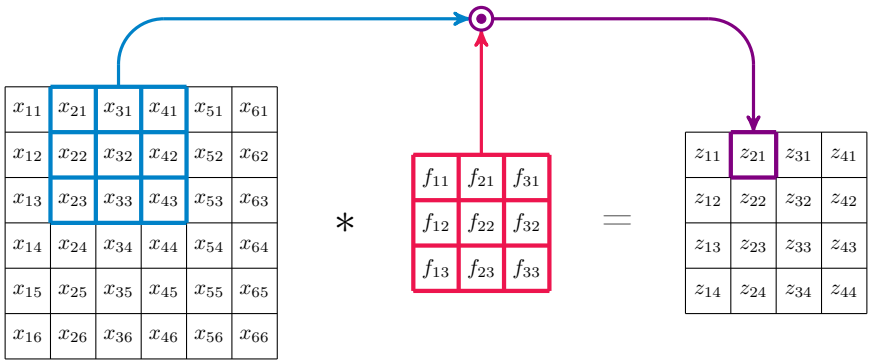


Figure 19.7: A convolution $Z = X * F$ on a single-channel image. Each value $z_{\ell m}$ of the convolution is the sum of the elementwise products of the corresponding submatrix of X with F. Thus z_{12} (purple) is $z_{12} = x_{21}f_{11} + x_{31}f_{21} + x_{41}f_{31} + x_{22}f_{12} + \cdots + x_{43}f_{33}$, the sum of the elementwise (Hadamard) products of the blue submatrix with the red filter F.

For simplicity, imagine an $h \times w$ black and white image X, with each pixel represented by a single real value x_{ij}. Let F be a $k \times k$ matrix of real values $f_{\ell m}$ (often called a *filter*). The *convolution* $X * F = Z$ is an $(h - k + 1) \times (w - k + 1)$ matrix with

$$z_{ab} = \sum_{\ell=1}^{k} \sum_{m=1}^{k} x_{a+k-\ell,b+k-m} f_{\ell m};$$

See also Volume 2, section 8.6. Each point z_{ab} of the convolution is produced by convolving a $k \times k$ submatrix of X with the $k \times k$ matrix F.

The convolution Z is a *filtered image* constructed by matching up the filter with each possible corresponding submatrix S of the image, taking the dot product of the filter F and the patch S (by multiplying the matched-up pairs of pixel values and then adding up all of the products), then placing the resulting value in the corresponding location in the filtered image Z.

In Figure 19.7 a 6×6 image is convolved with a 3×3 filter. The z_{12} entry of the convolution (purple) is constructed by taking the submatrix S (blue) of X corresponding to rows 1, 2, and 3 $(1 + 3 - \ell)$ and columns 2, 3, and 4 $(2 + 3 - m)$, multiplying each entry by its counterpart in F (red), and summing all the products.

For example, if a $k \times k$ filter has large values on its diagonal and small values elsewhere, then whenever it is convolved with a $k \times k$ submatrix that also has large values on its diagonal, then the resulting value will be large. But if the filter is convolved with a submatrix that does not have large values on the diagonal, then the result will be small. This partially explains the name *filter*, since convolving the filter with a submatrix returns a large value when the submatrix matches the filter and a smaller value when it does not match.

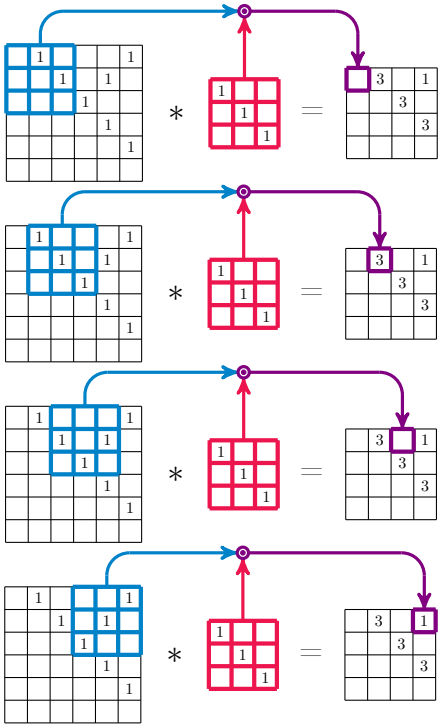


Figure 19.8: If the filter (red) has ones on the diagonal and zeros elsewhere, then each point of the convolution (purple) indicates how well the corresponding submatrix in the image (blue) matches the filter. Throughout this figure a blank square corresponds to 0. In the top panel the z_{11} term (purple) corresponds to the upper left corner of the image, which does not match the filter at all (hence gives 0). In the second panel the submatrix (blue) of the image matches the filter in all three places, so $z_{12} = \sum_{i=1}^{3} 1 \cdot 1 = 3$. In the third panel, the submatrix does not match at all, so $z_{13} = 0$. And in the bottom panel the submatrix matches only in the middle, so $z_{14} = 1 \cdot 1 = 1$.

The result of convolving an image with a filter is a new matrix of values that is no longer really what a human would think of as an image, but rather just a map identifying where and how well the filter matched the image.

19.5.2 Padding

Typically (and in Figure 19.7) the resulting filtered matrix is smaller than the original image. In general, an $h \times w$ image and a $k \times k$ filter will produce an $h - k + 1 \times w - k + 1$ filtered grid. (Non-square filters are possible, but rarely used in practice.) This size-reducing effect is often undesirable.

To address the problem that images shrink when convolved, it is common just to pad the input image, putting a border around it so that the filter has as many places to go as there are pixels.

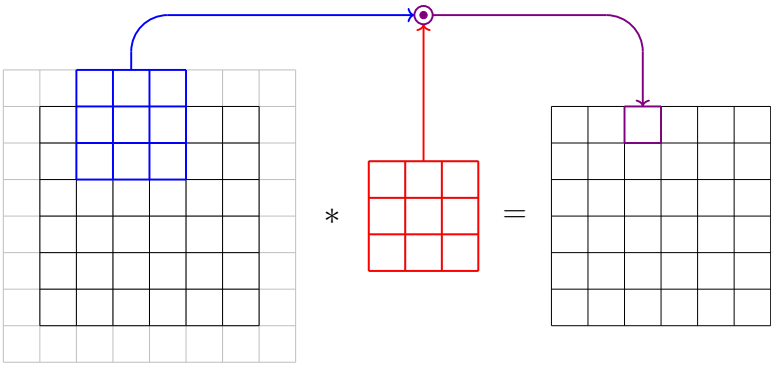


Figure 19.9: A padding scheme that preserves image size.

Usually, the values in this border are all set to zero. This is called *zero padding*. More sophisticated padding schemes have been proposed, including some that attempt to extrapolate appropriate values from the image, but these are almost never seen in practice because zero padding, the simplest possible solution, has proven sufficient for even the most sophisticated processing models.

When filters have odd sizes, the padding can be done in a symmetric fashion. A 3×3 filter needs one pixel of padding all the way around, a 5×5 filter needs two. Even-sized filters are rarely used in practice, in part because they require asymmetric padding.

19.5.3 Multiple Channels

Real-world color images usually have three or four channels, corresponding to red, green, and blue (RGB), or cyan, magneta, yellow, and black (CMYK). If an image has c channels, then the filter must have c channels as well. Channels of the filter get matched up with the corresponding image channels for the product step and everything is then summed, so the filtered image has a single channel, as in Figure 19.10

In convolutional networks, we'll almost always want to convolve an image with many filters. In this case, our output will have one channel per filter. This is demonstrated in Figure 19.11. The input image has four channels, so each filter has four channels. There are three filters, so the output image has three channels. In practice, it's common to see tens or hundreds of channels, but that would make for busy diagrams.

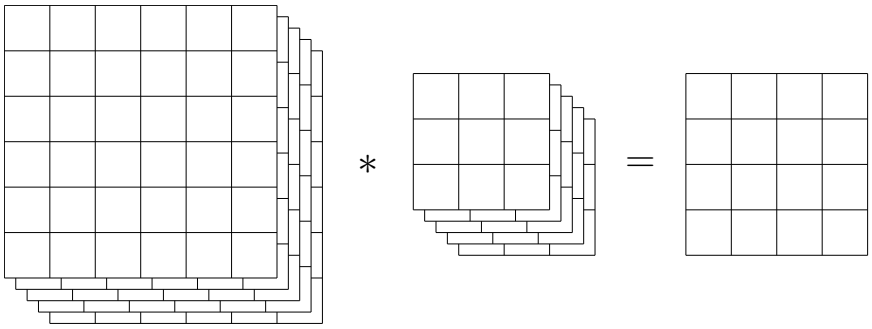


Figure 19.10: A convolution on a multi-channel image.

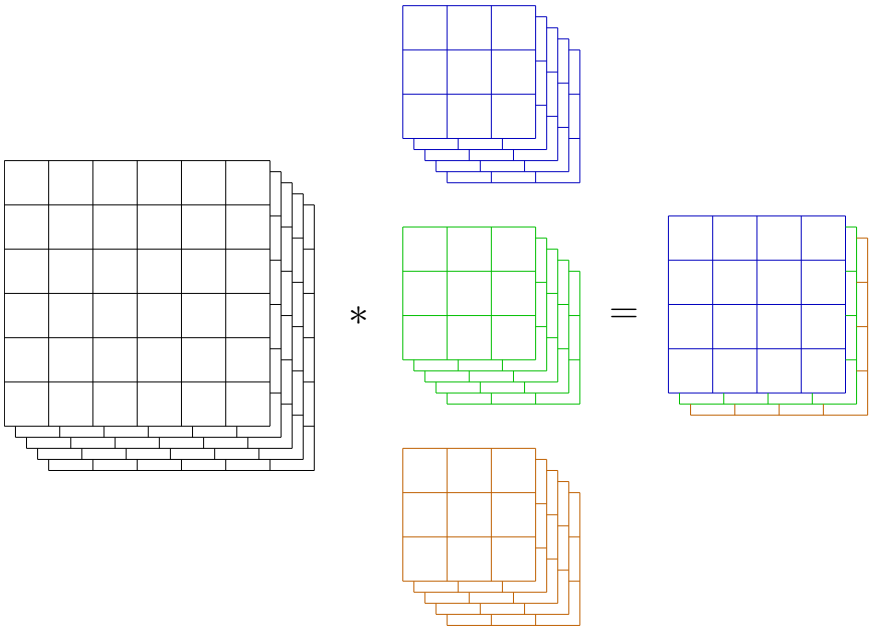


Figure 19.11: A multi-channel convolution on a multi-channel image.

19.5.4 Convolutional Neural Networks

We do not want to choose the filters for convolution by hand. Instead we want to have the network learn which filters are most useful. A convolutional network layer takes an input image with c_{in} channels and outputs an image with c_{out} channels. To do this, it uses c_{out} filters, each of which is $k \times k$ (k is usually 3 or 5) and has c_{in} channels so it can match up with the input images. Each filter also has a bias which is added to each pixel of the corresponding output channel. Every value of the output image is then run separately through the activation function (usually ReLU).

The values in the filters are usually called *weights* so that we can discuss convolutional layers with the same language as standard layers. The weights and biases are trained with SGD or some other optimizer, just as in a standard layer.

It is important to note that a convolutional layer can be described as a standard neural network layer, but where most of the weights are set to zero and many of the remaining weights are forced to be equal to each other. In principle, any function that can be represented by a convolutional network can also be represented by a fully connected network. But with fewer total parameters, the convolutional network can learn its weights faster than a fully connected network can, and a full connected network is not necessarily very likely to learn the special structure of the convolution, even if that special convolutional structure would give the best fit to the data.

Example 19.5.1. Consider one tiny convolutional layer that takes a 4×4 input X, applies a 3×3 convolution F with no padding to get a 2×2 output Z (then adds on a bias, and applies an activation).

This corresponds to a traditional, fully connected layer with a 16-dimensional input consisting of X flattened into

$$\text{flatten}(X) = (x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, \ldots, x_{44}).$$

This layer maps to 4 outputs $\text{flatten}(Z) = (z_{11}, z_{12}, z_{21}, z_{22})$, so its weight matrix W is 16×4. A little thought shows that the weight matrix W is

$$\begin{bmatrix} f_{11} & f_{12} & f_{13} & 0 & f_{21} & f_{22} & f_{23} & 0 & f_{31} & f_{32} & f_{33} & 0 & 0 & 0 & 0 & 0 \\ 0 & f_{11} & f_{12} & f_{13} & 0 & f_{21} & f_{22} & f_{23} & 0 & f_{31} & f_{32} & f_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & f_{11} & f_{12} & f_{13} & 0 & f_{21} & f_{22} & f_{23} & 0 & f_{31} & f_{32} & f_{33} & 0 \\ 0 & 0 & 0 & 0 & 0 & f_{11} & f_{12} & f_{13} & 0 & f_{21} & f_{22} & f_{23} & 0 & f_{31} & f_{32} & f_{33} \end{bmatrix}$$

Even though there are only 9 weights in this convolution, to write it in terms of a traditional weight matrix requires 64 entries.

Nota Bene 19.5.2. Just because a particular function can be represented by a network does not mean that it can be learned. In practice, a fully connected network requires many more parameters to represent the same function that a convolutional network represents. This means that much more memory and computational power are required to learn the correct parameters, and, more importantly, the learning algorithm could easily become stuck in a local minimum that represents a very different function. This is why convolutional networks are generally much more useful than fully connected networks, at least for images and other data where a convolutional filter is meaningful—not because the fully connected network cannot represent the same function, but because it cannot easily learn that function.

19.5.5 Pooling

Convolutional networks usually include a second type of layer called a *pooling* layer. These layers have no parameters, and they serve to shrink images in a simple way. The image is divided into $k \times k$ *pools* (k is usually 2) which are then collapsed down to one pixel; see Figure 19.12. This can be thought of as a form of downsampling. The two most common forms of pooling are *average pooling* and *max pooling*, which take the average and maximum value of the pixels in the pool, respectively. Generally max pooling is more useful than average pooling, for reasons we discuss below. Pooling is applied to each channel separately. When an image dimension isn't divisible by k, padding is used to compensate.

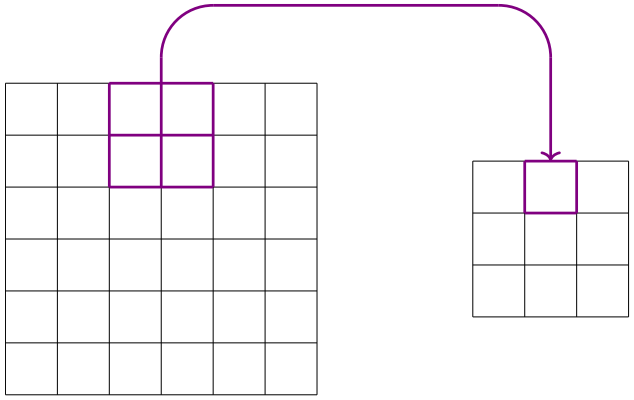


Figure 19.12: A 2×2 pooling operation.

Pooling is done in part for computational convenience. Convolutions are expensive. If images can be downsized without losing too much information, then training the network is sped up considerably. Usually pooling layers are used after several convolutional layers have been used to extract information about local image features. Each convolutional layer tends to have more filters than the previous. So nothing is really lost in pooling. The network takes a large grid with a few channels and turns it into a smaller grid with many channels.

Whereas the original few channels contained low-level information (usually the presence of a color in a pixel), the new channels contain higher level information about the presence of shapes and patterns. Most convolutional networks contain many convolutional layers, interspersed with a few pooling layers. In fact, many convolutional networks end with a global pooling layer, which shrinks the image down to a single pixel (with many channels).

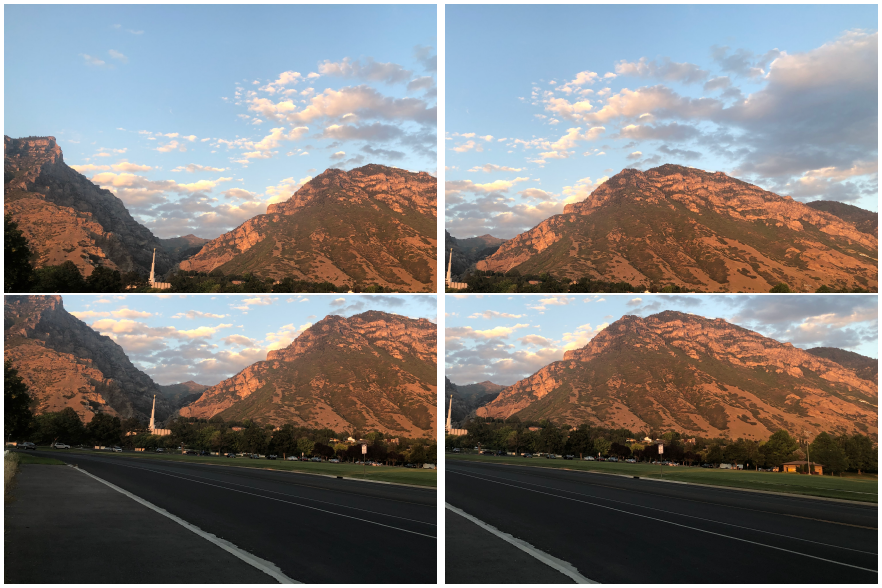


Figure 19.13: Translating (shifting) an image up or down or left or right by a small amount does not change the subject and meaning of the image. Each image here is a translation of the others, and this means that the underlying values at any given pixel may be very different in each image, but all four images represent a landscape with a temple near a mountain and colorful clouds. We say the subject of the image is *invariant* under small translations. One way to try to train a neural network to handle this type of invariance is to augment the training set with duplicate images that are translated in various ways, and train the network on the augmented data set. But this is expensive, both computationally and spatially. A better alternative is to structure the network in a way that reflects the desired translation invariance. Convolutional layers with max pooling do exactly that.

19.5.6 Convolution with Max Pool for Translation Invariance

Images have the important property that translating (sliding) them a little in some direction does not change the subject or the meaning of the image, but it completely changes the values at each pixel in the image. Pixel values in the bottom left corner of three of the panels of Figure 19.13 are dark, while the upper-right panel of that figure the bottom left pixels are white. But the images all depict essentially the same thing, as they are just minor translations of each other. We say the subject of the image is *invariant* under small translations. One way to try to train a neural network to handle this type of invariance is to augment the training set with duplicate images that are translated in various ways, and train the network on the augmented data set. But this is expensive, both computationally and spatially.

A better alternative is to structure the network in a way that reflects the desired translation invariance. Convolutional layers with max pooling do exactly that. To see this, consider a single 3×3 convolutional layer followed by a max pool layer. The convolutional layer learns a 3×3 filter. The filter is applied first in the upper left corner, then is translated by one pixel, and applied again, then translated by one pixel and applied again, and so forth. The same filter is applied at each location, and the output of the convolutional layer has a large value for each location where the image matches the filter well and a small value where it does not. Max pooling the entire output of the convolutional layer gives a large value if the image matched the filter well at any location in the image, and it has a small value if no location in the image matched the filter well. That is, the result of the convolution followed by max pooling is translation-invariant—the output is the same if any location in the image is a good match to a translation of the filter, regardless of where the location is.

Forcing the neural network to have this sort of translation invariance makes it much easier for the network to learn the correct output because it needs only to search among translation-invariant functions, whereas a fully connected network would search in a much larger space of functions, most of which are not translation invariant. This explains why convolutional networks perform so well for images and other translation-invariant problems, compared to fully connected networks, which could, in theory, represent the same function, but have a hard time learning that function because they are looking for it among a huge collection of unsuitable (not invariant) candidates.

19.5.7 Receptive Field

The previous argument suggests that if we want the network to recognize large patterns in an image, we might want to make a convolutional layer that is large, say 20×20 or 100×100. But state-of-the-art image recognition networks often use 3×3 filters exclusively, yet they are able to detect arbitrarily large objects in images. This may seem counterintuitive, but they work because each such filter (convolutional layer) increases the *receptive field*—the size of the area in the original image that affects any one pixel of a given convolutional layer's output.

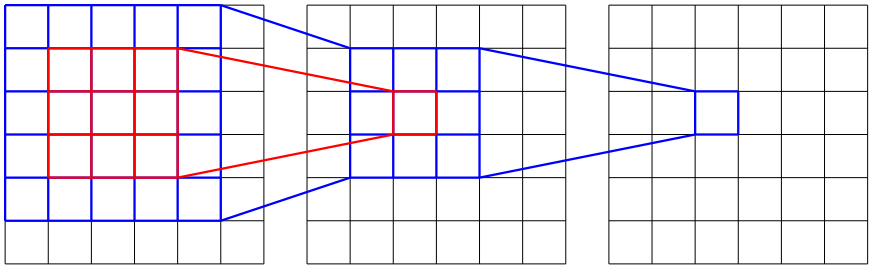


Figure 19.14: After two 3×3 convolutions, a single pixel (on the right) has a 5×5 receptive field in the original image (on the left).

Consider Figure 19.14, in which an original image is convolved with two 3×3 filters consecutively. A pixel in the second image is affected by a 3×3 region of the first, and likewise a pixel in the third image is affected by a 3×3 region of the second. So a pixel in the third image is affected by a 5×5 region of the first image.

The advantage of learning many small filters stacked on top of each other is that each subsequent layer can learn to assemble the small filters of the previous layer into more sophisticated filters, as depicted in Figures 19.15 and 19.16.

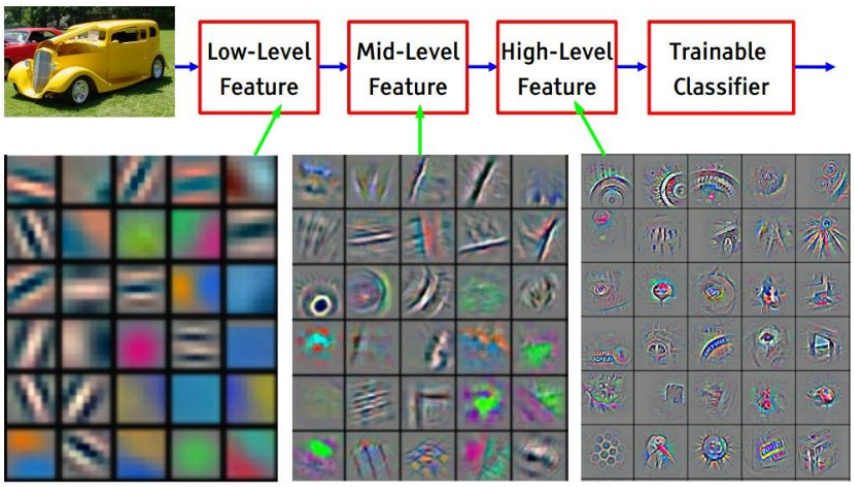


Figure 19.15: This figure depicts learned filters in a multilayer convolutional neural network. The first layers learn only simple filters (lines and blobs), but the intermediate layers can learn simple combinations of the first collection, resulting in more complicated filters, and the later layers can learn combinations of those intermediate filters, resulting in relatively sophisticated filters that can identify thins like a certain type of bird's head, a honeycomb pattern, or a face.

19.5.8 Sequence Processing with 1D Convolutions

Many forms of data have a natural sequence structure and can be expressed as an ordered list of elements. All forms of time series data fit this description, as do text, audio, video, and others. Two main tools exist for exploiting this sequence structure—1D convolutions and recurrent units.

Convolutions in one dimension work just as they do in two. If anything, they're simpler. A filter of length k is convolved with a sequence of length n to produce a filtered sequence of length $n - k + 1$.

If a data point is a sequence of vectors, then each dimension of those vectors can be thought of as a channel. Then filters must match the channel depth of the sequence with which they are convolved, and multiple filters can be used to produce a multi-channel output sequence.

Other convolutional network techniques, such as pooling and padding, also translate easily to the one-dimensional case.

Figure 19.16: Depiction of various layers in a multilayer CNN. The grayish images are, as in the previous figure, depictions of certain learned filters at the corresponding level. The more colorful images are pieces of images in the validation set that match the corresponding filter well. Again the early layers learn simple filters recognizing small elementary pieces, and the later layers assemble several filters from the previous layers into progressively more complicated and intricate filters. Image from [ZF14]

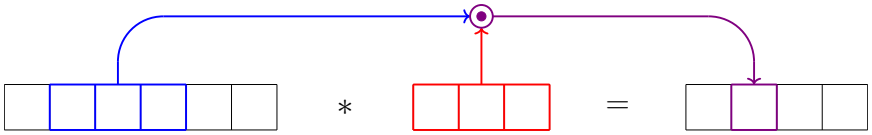


Figure 19.17: A 1D Convolution

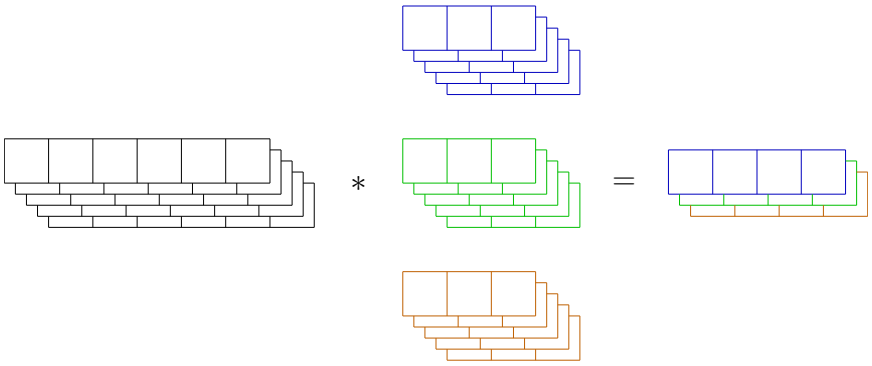


Figure 19.18: A 1D Convolution on Multi-Channeled Input

Exercises

Note to the student: Each section of this chapter has several corresponding exercises, all collected here at the end of the chapter. The exercises between the first and second line are for Section 1, the exercises between the second and third lines are for Section 2, and so forth.

You should **work every exercise** (your instructor may choose to let you skip some of the advanced exercises marked with *). We have carefully selected them, and each is important for your ability to understand subsequent material. Many of the examples and results proved in the exercises are used again later in the text. Exercises marked with ⚠ are especially important and are likely to be used later in this book and beyond. Those marked with † are harder than average, but should still be done.

Although they are gathered together at the end of the chapter, we strongly recommend you do the exercises for each section as soon as you have completed the section, rather than saving them until you have finished the entire chapter.

19.1. Prove that the composition of two affine functions is an affine function.

19.2. The logic gates shown below are the building blocks of all computer programs. Each is a simple function with 2 inputs and 1 output.

	0	1
0	0	0
1	0	1

AND

	0	1
0	0	1
1	1	1

OR

	0	1
0	1	1
1	1	0

NAND

	0	1
0	1	0
1	0	0

NOR

	0	1
0	0	1
1	1	0

XOR

	0	1
0	1	0
1	0	1

XNOR

For each of these functions, explicitly construct a neural network with at most three neurons that reproduces the function exactly. Use the Heaviside step function $h(z)$ as the activation.

19.3. Let $g(x)$ be any continuous function from $[0, 1]$ to $[0, 1]$. Show that a neural network with one hidden layer, using Heaviside activations in the hidden layer and using the identity for the output activation, can approximate $g(x)$ arbitrarily well in the L^1-norm. That is, show that

$$\int_0^1 |f_{\mathbf{n}}(x) - g(x)|\, dx$$

can be made less than ε for any $\varepsilon > 0$. Here $f_{\mathbf{n}}(x)$ is the function defined by the network.

Hint: $h(z) + h(1 - z) - 1$ is a step function, and sums of step functions are really good at approximating continuous functions, as you may recall from Volume 1.

19.4. Prove (19.2) to complete the proof of Theorem 19.1.9.

19.5. Build some neural nets:

(i) Install Keras on your computer (`pip install keras` usually works. If not, it certainly works on Google Colab)

(ii) Starting with the notebook `KerasNeuralNets.ipynb`, experiment with at least three different choices each of

- different layer widths
- different number of layers
- different number of epochs
- different batch sizes
- different learning rate, decay rate, and momentum

For each of these experiments, evaluate the total training time and the accuracy on the validation set, and plot the training history.

Keep experimenting until at least three different models have validation accuracy (not training and and not test) greater than 85% and at least one has training (not validation) accuracy greater than 90%.

For now, limit yourself to dense layers, fully connected, no dropout, no batch norm—just the parameters that are set here. The goal is to get a very accurate classifier with minimal training time.

Nota Bene 19.5.3. WARNING: Whenever you rerun the model with new architecture or parameters, you must first recompile it! If you rerun the model without recompiling, it will continue training the previously compiled model where it left off, as if you had just given it more epochs to train. This is helpful if you want to keep training your old model, but don't be fooled into thinking that your new model is somehow suddenly much better—it is just the old model starting at a better place.

19.6. Among the previous variations, identify the model that:

(i) Had the fastest training time. What was its final accuracy on the test (not validation) set?

(ii) Had the greatest accuracy on the validation set. What was its final accuracy on the test set? How long did it take to train?

(iii) Had the greatest accuracy on the test set. What was its final accuracy on the validation set? How long did it take to train?

(iv) Trained fastest among those that reached at least at least 89% accuracy on the validation set. What was its final accuracy on the test set?

(v) Had the fewest total parameters among those that reached at least at least 89% accuracy on the validation set. What was its final accuracy on the test set?

19.7. When training a neural net, it is common for the validation loss to first decrease for a while and then start going back up, even though training loss keeps going down as training continues. Did that happen with any of your models in Exercise 19.5? How did those models do on the test set?

19.8. Revist your `KerasNeuralNets.ipynb` notebook, used in Exercise 19.5. Using the largest (most total parameters) model you trained in Exercise 19.5, and using the **validation set** to choose among different hyperparameter choices, experiment with

- L^2 weight regularization (try at least 3 different choices of regularization weight, all differing by at least a factor of 10).
- dropout
- batch norm (but don't mix with dropout)

For each of these choices, experiment with at least three different choices of learning rate, decay rate, and momentum. As before, the goal is to get a very accurate classifier with minimal training time.
When you are done, run your classifier with the best combination of hyperparameters on the **test set** to estimate how well it will generalize.
WARNING: Remember, when changing hyperparameters, always recompile before rerunning the model.

19.9. For the function $f(x_1, x_2) = \cos(3x_1x_2 - x_1^2)$ do the following:

(i) Draw the computational graph

(ii) For the point $(x_1, x_2) = (2, 7)$ do the following:

(a) Compute $f(2, 7)$ by computing each of the values α_k of the nodes in the computational graph.

(b) Compute the full derivative $Df(2, 7)$ using the forward mode of automatic differentiation—computing $\dot{\alpha}_k$ for each node.

(c) Compute the full derivative $Df(2, 7)$ using the backward mode of automatic differentiation (back propagation)—computing $\bar{\alpha}_k$ for each node.

(d) Check your work by computing the derivative in the way you learned in calculus. Make sure all three derivative answers match.

(iii) Repeat the previous part (ii) at the point $(1, 0)$, computing the derivative three different ways and comparing the results.

19.10. Do the following using the Jax automatic differentiation package, You'll probably want to start with the commands `import jax.numpy as jnp` and `from jax import jacfwd, jacrev`

(i) Construct a function F that is the composition $F = f_3 \circ f_2 \circ f_1$ of three nonlinear functions, where $f_1 : \mathbb{R}^{5000} \to \mathbb{R}^{1000}$, $f_2 : \mathbb{R}^{1000} \to \mathbb{R}^{30}$, and $f_3 : \mathbb{R}^{30} \to \mathbb{R}$. Generate a random input vector $\mathbf{x} \in \mathbb{R}^{5000}$. Time the computation of the derivative $DF(\mathbf{x})$ computed using forward mode `jacfwd` and time the computation using reverse mode `jacrev`.

(ii) Construct a function G that is the composition $G = g_3 \circ g_2 \circ g_1$ of three nonlinear functions, where $g_1 : \mathbb{R} \to \mathbb{R}^{30}$, $g_2 : \mathbb{R}^{30} \to \mathbb{R}^{1000}$, and $g_3 : \mathbb{R}^{1000} \to \mathbb{R}^{5000}$. Generate a random input $x \in \mathbb{R}$. Time the computation of the derivative $DF(x)$ computed using forward mode `jacfwd` and time the computation using reverse mode `jacrev`.

(iii) Explain why the forward computation is much slower than the reverse in the first case, but much faster than the reverse in the second case.

(iv) Time the computation of the Hessian $D^2F(\mathbf{x})$ computed using each of the following methods: `(jacfwd(jacrev(F))(x)`, `(jacrev(jacrev(F))(x)`. Explain why you don't see the as great a difference between the results as you do in the previous two computations.

(v) Time the computation of the total derivative $DH(\mathbf{x})$ of the composition $H = G \circ F$ using foward mode and reverse mode.

19.11. Write out the full weight matrix (in the style of Example 19.5.1) for a 3×3 convolutional layer applied to a 4×4 input *with one pixel of zero padding* all around the input.

19.12. Stride: instead of moving the filter over by one pixel with each step, we can take a longer *stride* of $s \geq 1$, and move the filter over (or down) by s pixels at each step. What is the size of the output if a $k \times k$ filter is convolved with an $h \times k$ image with stride s with no padding? (Be sure to also figure out what happens with the various edge cases when things don't come out quite even)

19.13. Build and train, using your favorite deep learning tools, a convolutional neural network on a very large, interesting dataset suited to convolutional layers of your choice.

Ensure that your model has at least 2 convolutional layers, at least one max pool layer, at least one dropout or batchnorm layer (not both!), and at least 10^5 total parameters (in keras use `model.count_params()` to check the total number).

Separate your data into test, validation, and training sets and plot the training error and validation error over time on the same graph. Report the accuracy of the final model on the test set.

Hints:

(i) If you run the notebook on CoLab (recommended, unless you have access to another machine with GPUs), it will run much faster if you set the hardware accelerator to GPU instead of CPU. To do this, go to the menu at the top of the page and choose Runtime > Change Runtime Type.

(ii) Convolutions should come before fully connected layers. Convolutions don't make as much sense after a fully connected layer, since most of the spatial structure of the input will be lost in the fully connected layer.

Notes

Our treatment of automatic differentiation is inspired by the excellent article of Baydin-Perlmutter-Radul-Sisskind `https://www.jmlr.org/papers/volume18/17-468/17-468.pdf`

A Important Stuff

A.1 Boolean Algebras and Truth Tables

A.1.1 Brief Primer on Propositional Logic

TODO. Do we want this? Might make sense to discuss equivalence relations and stuff. Anything required to understand truth tables and other deductive things.

A.1.2 Truth Tables

The implication $A \Rightarrow B$ is a shorthand way of saying that whenever A is true, then B is true. That implication itself may or may not hold, so it has a truth value itself. It can be re-expressed using join and negations as $B \vee (\neg A)$. To verify the claim that $(A \Rightarrow B) = B \vee (\neg A)$ we can use a truth table, which just involves enumerating all the possible truth values for the statements A and B on the left, and the listing resulting truth values for $(A \Rightarrow B)$ and $B \vee (\neg A)$ on the right.

A	B	$A \Rightarrow B$	$B \vee (\neg A)$
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

The expressions $A \Rightarrow B$ and $B \vee (\neg A)$ are the same for every combination of truth values for A and B, showing that indeed $(A \Rightarrow B) = (B \vee (\neg A))$. It is sometimes useful to think of expressions like $(A \Rightarrow B)$ and $B \vee (\neg A)$ as functions that accept inputs A and B (taking values in $\{T, F\}$) and that have outputs in $\{T, F\}$. If two functions take the same values at all inputs, then they are the same function; that is, they are equal as functions.

Example A.1.1. The disjunctive syllogism (ii) involves two basic propositions, A and B, and the formulas $A \vee B$ and $\neg A$. The truth table we need in order to examine the syllogism is the following.

A	B	$A \vee B$	$\neg A$
T	T	T	F
T	F	T	F
F	T	T	T
F	F	F	T

The syllogism assumes the premises of $A \vee B$ and $\neg A$, so we look only at the rows of the table where both of those formulas are true. In this case there is only one such row, and in that row the statement B is indeed true, so the syllogism is valid: if $A \vee B$ is true and $\neg A$ is true, then B is also true.

A.1.3 Boolean Algebras

A *Boolean algebra* is a mathematical framework for for working with propositions by joining them together to create new statements using various combinations of the three operations product, join, and negation. As an abstract mathematical object, it consists of a set (of propositions) with the operations product, join, and negation, satisfying the rules of logic earlier in this section, including those of Proposition 1.1.2.

Definition A.1.2. *Given a set $\mathscr{C}$ of propositions, we say that $\mathscr{C}$ is a* Boolean algebra *if it contains the propositions T (True) and F (False) (these are also often denoted 1 and 0, respectively) and is closed under the operations of product, and negation; that is,*

(*i*) *If $A, B \in \mathscr{C}$, then $AB \in \mathscr{C}$.*

(*ii*) *If $A \in \mathscr{C}$, then $\neg A \in \mathscr{C}$.*

Remark A.1.3. A little work shows that if a set $\mathscr{C}$ of propositions is closed under product and negation, it must also be closed under join; that is, if $A, B \in \mathscr{C}$, then $A \vee B \in \mathscr{C}$. This means that in any Boolean algebra $\mathscr{C}$, we can reason with all three operations and the result will always be contained in $\mathscr{C}$—no combination of the operations and propositions will produce a new proposition that is not in $\mathscr{C}$.

Example A.1.4. If $\mathscr{C} = \{T, F\}$, then $\mathscr{C}$ is a Boolean algebra. One way to see that is to check every product and negation: $TT = T \in \mathscr{C}$, $TF = F \in \mathscr{C}$, $FF = F \in \mathscr{C}$, $\neg F = T \in \mathscr{C}$, and $\neg T = F \in \mathscr{C}$.

Definition A.1.5. *Given a set $\mathscr{A}$ of propositions, the Boolean algebra* generated *by $\mathscr{A}$ is the smallest Boolean algebra that contains all the propositions in $\mathscr{A}$.*

Example A.1.6. The Boolean algebra generated by the set $\{A\}$ is $\mathscr{C} = \{A, \neg A, F, T\}$. This can be seen by checking first that any Boolean algebra that contains A must also contain every element of $\mathscr{C}$ and then checking that $\mathscr{C}$ is indeed a Boolean algebra (check the closure requirements).

Remark A.1.7. The reason the Boolean algebra generated by $\mathscr{A}$ is important is that it is the collection of all propositions that we can logically reason about if we know the truth value of all the propositions in $\mathscr{A}$.

A.2 Jensen's Inequality

Jensen's inequality has particular value in the context of probability theory. Here we present three different incarnations, all of which are useful.

Theorem A.2.1 (Jensen's Inequality—Finite Case (Volume 2, Theorem 15.2.4)). *Assume $C \subset V$ is a convex set and $\phi : C \to \mathbb{R}$ a convex function. If $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in C$ and $\lambda_1, \lambda_2, \ldots, \lambda_n \in [0,1]$, with $\sum_{i=1}^n \lambda_i = 1$, then*

$$\phi(\lambda_1 \mathbf{x}_1 + \lambda_2 \mathbf{x}_2 + \cdots + \lambda_n \mathbf{x}_n) \leq \lambda_1 \phi(\mathbf{x}_1) + \lambda_2 \phi(\mathbf{x}_2) + \cdots + \lambda_n \phi(\mathbf{x}_n). \tag{A.1}$$

Example A.2.2. Multiplying both sides of Jensen's inequality by -1 shows that if a function ψ is concave ($-\psi$ is convex), then

$$\psi(\lambda_1 \mathbf{x}_1 + \lambda_2 \mathbf{x}_2 + \cdots + \lambda_n \mathbf{x}_n) \geq \lambda_1 \psi(\mathbf{x}_1) + \lambda_2 \psi(\mathbf{x}_2) + \cdots + \lambda_n \psi(\mathbf{x}_n). \tag{A.2}$$

Let Ω be a finite set, and let $P, Q : 2^\Omega \to [0,1]$ be two probability distributions on Ω. The logarithm function is concave, so using the values $P(\omega)$ in place of the λ_i in the concave form of Jensen's inequality (A.2) gives

$$\begin{aligned} 0 = \log\left(\sum_{\omega\in\Omega} Q(\omega)\right) &\geq \log\left(\sum_{\omega:P(\omega)\neq 0} Q(\omega)\right) = \log\left(\sum_{\omega:P(\omega)\neq 0} P(\omega)\frac{Q(\omega)}{P(\omega)}\right) \\ &\geq \sum_{\omega:P(\omega)\neq 0} P(\omega)\log\left(\frac{Q(\omega)}{P(\omega)}\right) = \sum_{\omega\in\Omega} \left(P(\omega)\log(Q(\omega)) - P(\omega)\log(P(\omega))\right). \end{aligned}$$

The first inequality above holds because log is strictly increasing, so discarding some nonnegative values of $Q(\omega)$ can only make the result smaller. The last equality holds because all the new terms inserted back into the sum are zero (we use the convention that $a \log b = 0$ if a and b are both 0).

This shows that

$$-\sum_{\omega\in\Omega} P(\omega)\log(P(\omega)) \leq -\sum_{\omega\in\Omega} P(\omega)\log(Q(\omega)). \tag{A.3}$$

This is called *Gibbs' inequality* and is a fundamental result in information theory. We revisit this in Theorem **??**.

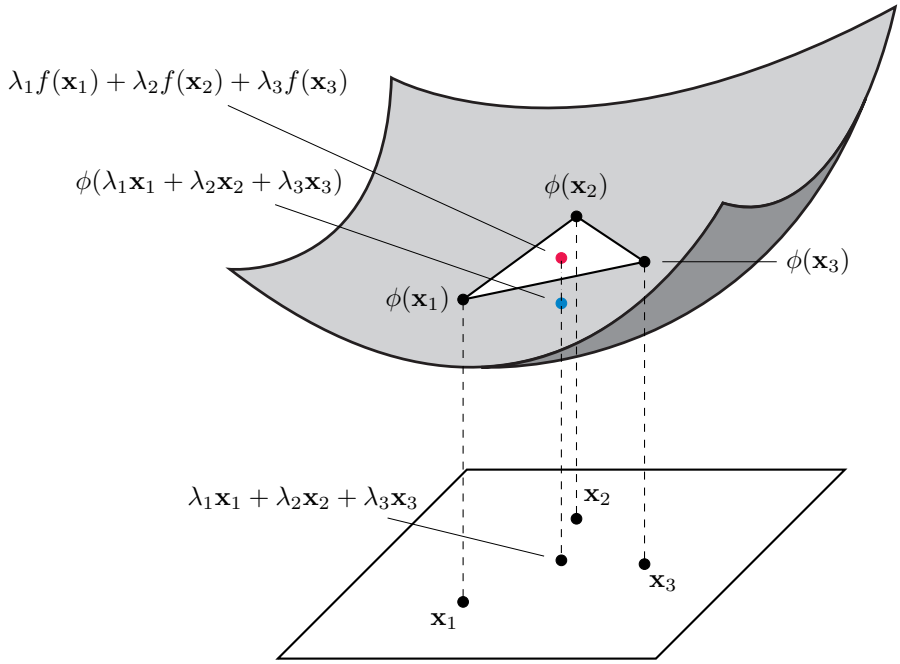


Figure A.1: The graph of a convex function ϕ and three points $\phi(\mathbf{x}_1)$, $\phi(\mathbf{x}_2)$, $\phi(\mathbf{x}_3)$ on the graph. Jensen's inequality guarantees that a convex combination $\lambda_1\phi(\mathbf{x}_1) + \lambda_2\phi(\mathbf{x}_2) + \lambda_3\phi(\mathbf{x}_3)$ (the red point) of the three function values lies above the function evaluated at the convex combination $\phi(\lambda_1\mathbf{x}_1 + \lambda_2\mathbf{x}_2 + \lambda_3\mathbf{x}_3)$ (the blue point); and, in fact, every point in the convex span of these points (the white triangle) lies above the graph.

Theorem A.2.3 (Jensen's Inequality—Integral Form (Volume 2, Theorem 15.2.10)). *Let $C \subset \mathbb{R}^n$ be a convex set and $f : C \to \mathbb{R}$ be an integrable function with $\int_C f(\mathbf{x})\, d\mathbf{x} = 1$. If $\phi : C \to \mathbb{R}$ is a convex function with closed epigraph, then*

$$\phi\left(\int_C \mathbf{x} f(\mathbf{x})\, d\mathbf{x}\right) \leq \int_C \phi(\mathbf{x}) f(\mathbf{x})\, d\mathbf{x}.$$

Theorem A.2.4 (Jensen's Inequality—Probability Form). *Let $(\Omega, \mathscr{F}, P)$ be a probability space and let ϕ be convex (with closed epigraph). If X and $\phi(X)$ each has an expected value, then*

$$\phi(\mathbb{E}[X]) \leq \mathbb{E}[\phi(X)]. \tag{A.4}$$

Example A.2.5. Since the functions $\phi(x) = |x|$ and $\psi(x) = x^2$ are convex, we have

$$|\mathbb{E}[X]| \leq \mathbb{E}[|X|] \qquad \text{and} \qquad \mathbb{E}[X]^2 \leq \mathbb{E}[X^2].$$

for any random variables X.

Corollary A.2.6 (Young's Inequality). *If $a, b \geq 0$ and $\frac{1}{p} + \frac{1}{q} = 1$, where $1 < p, q < \infty$, then*

$$ab \leq \frac{a^p}{p} + \frac{b^q}{q}. \tag{A.5}$$

Proof. Since e^x is convex, we have

$$ab = e^{\log ab} = e^{\frac{1}{p}\log a^p + \frac{1}{q}\log b^q} \leq \frac{1}{p}e^{\log a^p} + \frac{1}{q}e^{\log b^q} = \frac{a^p}{p} + \frac{b^q}{q}. \quad \square$$

Many important inequalities follow easily from Young's inequality, including the arithmetic-geometric mean inequality, Hölder's inequality, and Minkowski's inequality.

Hölder's and Minkowski's Inequalities also play an important role in probability. In the language of expectation, they take the following form:

Theorem A.2.7 (Hölder's Inequality).

$$\mathbb{E}[|XY|] \leq \mathbb{E}[|X|^p]^{1/p}\mathbb{E}[|Y|^q]^{1/q},$$

where $p, q > 1$ with $1/p + 1/q = 1$.

Theorem A.2.8 (Minkowski's Inequality).

$$\mathbb{E}[|X + Y|^p]^{1/p} \leq \mathbb{E}[|X|^p]^{1/p} + \mathbb{E}[|Y|^p]^{1/p},$$

for any $p \geq 1$.

B Summary of Distributions

I think it's great any time someone can be in control of their own distribution.
—Gaby Hoffman

Here is a summary of some of the main properties of the most common distributions.

Distribution	parameters	mean	variance	p.d.f/p.m.f.	m.g.f.	c
Bernoulli	$p \in [0,1]$	$p^x(1-p)^x$	p	$p(1-p)$	$(1-p)+pe^t$	B
Binomial	$p \in [0,1], n \in \mathbb{N}$	$\binom{n}{x}p^x(1-p)^{1-x}$	np	$np(1-p)$	$((1-p)+pe^t)^n$	B
Categorical (multinoulli)						

Bibliography

[ABMM16] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. *CoRR*, abs/1611.01491, 2016. [675]

[BH15] Joseph K. Blitzstein and Jessica Hwang. *Introduction to probability*. Texts in Statistical Science Series. CRC Press, Boca Raton, FL, 2015. [473]

[BKHD15] Johannes Bohannon, Diana Koch, Peter Homm, and Alexander Driehaus. *International Archives of Medicine*, 8(5), 2015. [395]

[Boh15] Johannes Bohannon. I fooled millions into thinking chocolate helps weight loss. here's how. *Gizmodo*, May 2015. [395]

[CC17] Charles K. Chui and Guanrong Chen. *Kalman filtering—with real-time applications*. Springer, Cham, fifth edition, 2017. [554]

[CSS16] Nadav Cohen, Or Sharir, and Amnon Shashua. On the expressive power of deep learning: A tensor analysis. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 698–728, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR. [675]

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. Wiley-Interscience [John Wiley & Sons], Hoboken, NJ, second edition, 2006. [105]

[DK12] J. Durbin and S. J. Koopman. *Time series analysis by state space methods*, volume 38 of *Oxford Statistical Science Series*. Oxford University Press, Oxford, second edition, 2012. [583]

[Dur10] Rick Durrett. *Probability: theory and examples*, volume 31 of *Cambridge Series in Statistical and Probabilistic Mathematics*. Cambridge University Press, Cambridge, fourth edition, 2010. [179]

[ES16] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 907–940, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR. [675]

[Fel71] William Feller. *An introduction to probability theory and its applications. Vol. II.* Second edition. John Wiley & Sons, Inc., New York-London-Sydney, 1971. [200]

[Fra17] Brigham Frandsen. Econ 588 lecture notes. 2017. [330]

[GA15] Mohinder S. Grewal and Angus P. Andrews. *Kalman filtering*. John Wiley & Sons, Inc., Hoboken, NJ, fourth edition, 2015. Theory and practice using MATLAB®. [536, 567]

[Gey11] Charles J. Geyer. Introduction to mcmc. In S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, editors, *Handbook of Markov Chain Monte Carlo*. Chapman and Hall/CRC, 1st edition, 2011. Accessed: 2025-01-03. [473]

[Gif09] Adom Giffin. Maximum entropy: The universal method for inference, 2009. [105]

[Gil14] N. Gillis. The Why and How of Nonnegative Matrix Factorization. *ArXiv e-prints*, January 2014. [619]

[GKV04] Gerd Gigerenzer, Stefan Krauss, and Oliver Vitouch. The null ritual: What you always wanted to know about significance testing but were afraid to ask. In D. Kaplan, editor, *The Sage handbook of quantitative methodology for the social sciences*, pages 391–408. Sage, Thousand Oaks, CA, 2004. [383, 395]

[GL13] Andrew Gelman and Eric Loken. The garden of forking paths. `http://www.stat.columbia.edu/~gelman/research/unpublished/p_hacking.pdf`, 2013. Last accessed 1 September 2019. [383, 388, 395]

[GS01] Geoffrey R. Grimmett and David R. Stirzaker. *Probability and random processes*. Oxford University Press, New York, third edition, 2001. [185]

[Han17] Boris Hanin. Universal function approximation by deep neural nets with bounded width and relu activations. *arXiv preprint arXiv:1708.02691*, 2017. [675]

[HLM17] Nick Harvey, Christopher Liaw, and Abbas Mehrabian. Nearly-tight vc-dimension bounds for piecewise linear neural networks. In Satyen Kale and Ohad Shamir, editors, *Proceedings of the 2017 Conference on Learning Theory*, volume 65 of *Proceedings of Machine Learning Research*, pages 1064–1068, Amsterdam, Netherlands, 07–10 Jul 2017. PMLR. [675]

[Hor93] K. Hornik. Some new results on neural network approximation. *Neural Networks*, 6(8):1069 – 1072, 1993. [674]

[HS17] Boris Hanin and Mark Sellke. Approximating continuous functions by relu nets of minimal width. *arXiv preprint arXiv:1710.11278*, 2017. [675]

[Ioa05] John P. A. Ioannidis. Why most published research findings are false. *PLOS Medicine*, 2(8):e124, 2005. [383]

[Jay63] E. T. Jaynes. Information theory and statistical mechanics. In *Statistical physics (Brandeis Summer Institute, 1962, Vol. 3)*, pages 181–218. W. A. Benjamin, Inc., New York, 1963. [105]

[Jay03] E. T. Jaynes. *Probability theory*. Cambridge University Press, Cambridge, 2003. The logic of science, Edited and with a foreword by G. Larry Bretthorst. [70]

[Kal60] R. E. Kalman. A new approach to linear filtering and prediction problems. *Trans. ASME Ser. D. J. Basic Engrg.*, 82(1):35–45, 1960. [535]

[Kal63] R. E. Kalman. Mathematical description of linear dynamical systems. *J. SIAM Control Ser. A*, 1:152–192 (1963), 1963. [535]

[KB61] R. E. Kalman and R. S. Bucy. New results in linear filtering and prediction theory. *Trans. ASME Ser. D. J. Basic Engrg.*, 83:95–108, 1961. [535, 536]

[KF09] Daphne Koller and Nir Friedman. *Probabilistic graphical models*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2009. Principles and techniques. [485]

[Kle08] Achim Klenke. *Probability theory*. Universitext. Springer-Verlag London, Ltd., London, 2008. A comprehensive course, Translated from the 2006 German original. [200, 201]

[Knu77] Donald E. Knuth. The computer as master mind. *Journal of Recreational Mathematics*, 9:1–6, 1977. [105]

[Kol60] Andrey N. Kolmogorov. *Foundations of the Theory of Probability*. Chelsea Pub Co, 2 edition, June 1960. [179]

[KW96] Robert E. Kass and Larry Wasserman. The selection of prior distributions by formal rules. *Journal of the American Statistical Association*, 91(435):1343–1370, 1996. [246]

[LLW10] John Lafferty, Han Liu, and Larry Wasserman. *Concentration of Measure*. http://www.stat.cmu.edu/ larry/=sml/Concentration.pdf, 2010. [331]

[LPW+17] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. *CoRR*, abs/1709.02540, 2017. [675]

[LS16] Shiyu Liang and R. Srikant. Why deep neural networks? *CoRR*, abs/1610.04161, 2016. [675]

[LTR17] Henry W. Lin, Max Tegmark, and David Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, Sep 2017. [675]

[Lue06] David G. Luenberger. *Information Science*. Princeton University Press, Princeton, NJ, USA, 2006. [105]

[LZ86] H. Linhart and W. Zucchini. *Model selection*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons, Inc., New York, 1986. [330]

[Mac03] David J. C. MacKay. *Information theory, inference and learning algorithms*. Cambridge University Press, New York, 2003. [105]

[MGG+19] Blakeley B. McShane, David Gal, Andrew Gelman, Christian Robert, and Jennifer L. Tackett. Abandon statistical significance. *Amer. Statist.*, 73(suppl. 1):235–245, 2019. [383]

[MLP16] Hrushikesh Mhaskar, Qianli Liao, and Tomaso A. Poggio. Learning real and boolean functions: When is deep better than shallow. *CoRR*, abs/1603.00988, 2016. [675]

[MP16] H. N. Mhaskar and T. Poggio. Deep vs. shallow networks: An approximation theory perspective. *Analysis and Applications*, 14(06):829–848, 2016. [675]

[MPCB14] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2924–2932. Curran Associates, Inc., 2014. [675]

[MRR+53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. [473]

[MRT18] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2018. Second edition of [MR3057769]. [653]

[Ope15] Open Science Collaboration. Estimating the reproducibility of psychological science. *Science*, 349(6251), 2015. [-]

[Pea09] Judea Pearl. *Causality*. Cambridge University Press, Cambridge, second edition, 2009. Models, reasoning, and inference. [489, 490]

[PGEB18] Dmytro Perekrestenko, Philipp Grohs, Dennis Elbrächter, and Helmut Bölcskei. The universal approximation power of finite-width deep relu networks. *CoRR*, abs/1806.01528, 2018. [675]

[PM18] Judea Pearl and Dana Mackenzie. *The book of why*. Basic Books, New York, 2018. The new science of cause and effect. [492, 494]

[Pol54] George Polya. *Mathematics and plausible reasoning. I. Induction and analogy in mathematics. II. Patterns of plausible inference.* Princeton University Press, 1954. [70]

[RPK+17] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl Dickstein. On the expressive power of deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 2847–2854. JMLR.org, 2017. [675]

[RT17] David Rolnick and Max Tegmark. The power of deeper networks for expressing natural functions. *CoRR*, abs/1705.05502, 2017. [675]

[Shi84] A. N. Shiryayev. *Probability*, volume 95 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1984. Translated from the Russian by R. P. Boas. [200]

[SM15] Sho Sonoda and Noboru Murata. Neural network with unbounded activations is universal approximator. *CoRR*, abs/1505.03654, 2015. [675]

[Sta17] Mark Stamp. *Introduction to Machine Learning with Applications in Information Security*. Chapman & Hall/CRC Press, 2017. [504]

[STIM18] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?, 2018. [685]

[Tel15] Matus Telgarsky. Representation benefits of deep feedforward networks. *CoRR*, abs/1509.08101, 2015. [675]

[Tel16] Matus Telgarsky. Benefits of depth in neural networks. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 1517–1539, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR. [675]

[Tod11] Alexis Akira Toda. Unification of maximum entropy and bayesian inference via plausible reasoning. *CoRR*, abs/1103.2411, 2011. [105]

[vdMH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. [616]

[vdV12] Antoine A. J. van de Ven. Modeling the world by minimizing relative entropy. *AIP Conference Proceedings*, 1443(1):80–87, 2012. [105]

[Was04] Larry Wasserman. *All of statistics*. Springer Texts in Statistics. Springer-Verlag, New York, 2004. A concise course in statistical inference. [395]

[WL16] Ronald L. Wasserstein and Nicole A. Lazar. The ASA's statement on p-values: context, process, and purpose [Editorial]. *Amer. Statist.*, 70(2):129–133, 2016. [383]

[WRS+17] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning, 2017. [682]

[Yar17] Dmitry Yarotsky. Error bounds for approximations with deep relu networks. *Neural Networks*, 94:103 – 114, 2017. [675]

[ZF14] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing. [713]

[ZFM+20] Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven Hoi, and Weinan E. Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning. *arXiv e-prints*, page arXiv:2010.05627, October 2020. [682]

[Zil08] Stephen T. Ziliak. Retrospectives: Guinnessometrics: The Economic Foundation of "Student's" t. *Journal of Economic Perspectives*, 22(4):199–216, Fall 2008. [390]